Lecture Notes in Computer Science 1877
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Catuscia Palamidessi (Ed.)

CONCUR 2000 – Concurrency Theory

11th International Conference
University Park, PA, USA, August 22-25, 2000
Proceedings

Springer

Volume Editor

Catuscia Palamidessi
Pennsylvania State University
Department of Computer Science and Engineering
220 Pond Laboratory, University Park, PA 16802-6106, USA
E-mail: catuscia@cse.psu.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Concurrency theory : 11th international conference ; proceedings /
CONCUR 2000, University Park, PA, USA, August 22 - 25, 2000. Catuscia
Palamidessi (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong
Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1877)
ISBN 3-540-67897-2

CR Subject Classification (1998): F3, F.1, D.3, D.1, C.2

ISSN 0302-9743
ISBN 3-540-67897-2 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Da-TeX Gerd Blumenstein
Printed on acid-free paper SPIN 10722387 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000) held in State College, Pennsylvania, USA, during 22-25 August 2000.

The purpose of the CONCUR conferences is to bring together researchers, developers, and students in order to advance the theory of concurrency and promote its applications. Interest in this topic is continuously growing, as a consequence of the importance and ubiquity of concurrent systems and their applications, and of the scientific relevance of their foundations. The scope covers all areas of semantics, logics, and verification techniques for concurrent systems. Topics include concurrency related aspects of: models of computation, semantic domains, process algebras, Petri nets, event structures, real-time systems, hybrid systems, decidability, model-checking, verification techniques, refinement techniques, term and graph rewriting, distributed programming, logic constraint programming, object-oriented programming, typing systems and algorithms, case studies, tools, and environments for programming and verification.

The first two CONCUR conferences were held in Amsterdam (NL) in 1990 and 1991. The following ones in Stony Brook (USA), Hildesheim (D), Uppsala (S), Philadelphia (USA), Pisa (I), Warsaw (PL), Nice (F), and Eindhoven (NL). The proceedings have appeared in Springer LNCS, as Volumes 458, 527, 630, 715, 836, 962, 1119, 1243, 1466, and 1664.

Of the 72 regular papers submitted this year, 34 were accepted for presentation and are included in the present volume. The conference included also talks by four invited speakers: Ed Brinksma (Universiteit Twente, NL), Alberto Sangiovanni-Vincentelli (UC at Berkeley, USA), Natarajan Shankar (SRI International, USA), and Eugene Stark (SUNY at Stony Brook, USA). Additionally, there were four invited tutorials by Rajeev Alur (University of Pennsylvania, USA), Rocco De Nicola (Università di Firenze, I), Philippa Gardner (University of Cambridge, UK), and C. R. Ramakrishnan (SUNY at Stony Brook, USA).

The conference had four satellite events:

- EXPRESS'00 (7th International Workshop on Expressiveness in Concurrency), organized by Luca Aceto and Björn Victor, held on 21 August 2000.
- GETCO 2000 (2nd Workshop on Geometric and Topological Methods in Concurrency Theory), organized by Eric Goubault, Maurice Herlihy, and Martin Raussen, held on 21 August 2000.
- YOO (Why Object-Orientation) organized by Arend Rensink, held on 26 August 2000.
- MTCS 2000 (First Workshop on Models for Time-Critical Systems) organized by Flavio Corradini and Paola Inverardi, held on 26 August 2000.

I would like to thank my conference co-chair, Dale Miller, the members of the program committee and their subreferees, the workshop chair, Uwe Nestmann, and the workshop organizers. I also would like to thank the invited speakers and

invited tutorial speakers, the authors of submitted papers, and all the participants of the conference. Special thanks go to Vladimiro Sassone for providing the software that helped to handle the submissions and the electronic PC meeting. Finally, I would like to thank the Department of Computer Science and Engineering of the Pennsylvania State University for sponsoring the event and providing many facilities.

I also would like to express all my gratitude and love to my husband Dale Miller and to our baby daughter, Nadia Alexandra Miller, for their patience with a wife and mom who was even busier than usual.

June 2000 Catuscia Palamidessi

CONCUR Steering Committee

Jos Baeten (Technische Universiteit Eindhoven, NL, Chair)
Eike Best (Carl von Ossietzky Universität Oldenburg, D)
Kim Larsen (Aalborg Universitet, DK)
Ugo Montanari (Università di Pisa, I)
Scott Smolka (State University of New York at Stony Brook, USA)
Pierre Wolper (Université de Liège, B)

Program Committee

Samson Abramsky (University of Edinburgh, UK)
Jos C.M. Baeten (Technische Universiteit Eindhoven, NL)
Eike Best (Carl von Ossietzky Universität Oldenburg, D)
Michele Boreale (Università di Firenze, I)
Steve Brookes (Carnegie Mellon University, USA)
Luca Cardelli (Microsoft Research Cambridge, UK)
Ilaria Castellani (INRIA Sophia-Antipolis, F)
Philippe Darondeau (INRIA Rennes, F)
Thomas Henzinger (University of California at Berkeley, USA)
Radha Jagadeesan (Loyola University Chicago, USA)
Marta Kwiatkowska (University of Birmingham, UK)
Dale Miller (Penn State University, USA, Co-chair)
Robin Milner (University of Cambridge, UK)
Uwe Nestmann (BRICS, DK)
Catuscia Palamidessi (Penn State University, USA, Co-chair)
Prakash Panangaden (McGill University, CA)
John Reppy (Bell Labs, USA)
Vladimiro Sassone (Università di Catania, I)
Moshe Y. Vardi (Rice University, USA)
Wang Yi (Uppsala Universitet, S)

Referees

Samson Abramsky
Luca Aceto
Luca de Alfaro
Rajeev Alur
Roberto Amadio
Suzana Andova
André Arnold
Thomas Arts
Eric Badouel
Jos Baeten
Paolo Baldan
Marek A. Bednarczyk
Giampaolo Bella
Eike Best
Chiara Bodei
Frank de Boer
Roland Bol
Michele Boreale
Michel Le Borgne
Dragan Bošnački
Ahmed Bouajjani
Amar Bouali
Gérard Boudol
Olivier Bournez
Stephen Brookes
Roberto Bruni
Glenn Bruns
Benoît Caillaud
Luca Cardelli
Ilaria Castellani
Rance Cleaveland
Andrea Corradini
Flavio Corradini
Jean-Michel Couvreur
Pedro R. D'Argenio
Philippe Darondeau
Pierpaolo Degano
Raymond Devillers
Henning Dierks
André Engels
Javier Esparza
Gianluigi Ferrari
Hans Fleischhack
Riccardo Focardi
Cédric Fournet
David de Frutos-Escrig
Fabio Gadducci
Paul Gastin
Simon Gay
Stefania Gnesi
Jan Friso Groote
Vineet Gupta
Dilian Gurov
Keijo Heljanko
Matthew Hennessy
Jesper G. Henriksen
Thomas Henzinger
Holger Hermanns
Thomas T. Hildebrandt
Leszek Holenderski
Kohei Honda
Benjamin Horowitz
Hans Hüttel
Anna Ingólfsdóttir
Purushothaman Iyer
Radha Jagadeesan
Bengt Jonsson
Jan Jürjens
Joost-Pieter Katoen
Josva Kleist
Maciej Koutny
K. Jelling Kristoffersen
Marta Kwiatkowska
Barbara König
Anna Labella
Patrick Lam
Rom Langerak
Diego Latella
Björn Lisper
Michele Loreti
Gavin Lowe
Bas Luttik
Rupak Majumdar
Freddy Y. C. Mang
Will Marrero
Narciso Marti-Oliet
Richard Mayr
Massimo Merro
Kees Middelburg
Dale Miller
Robin Milner
Marius Minea
Madhavan Mukund
Uwe Nestmann
Rocco De Nicola
Mogens Nielsen
Thomas Noll
Gethin Norman
Catuscia Palamidessi
Prakash Panangaden
Dave Parker
Joachim Parrow
Doron Peled
Wojciech Penczek
Antoine Petit
Paul Pettersson
Anna Philippou
Sophie Pinchinat
Alban Ponse
Rosario Pugliese
Michel A. Reniers
Arend Rensink
John Reppy
Elvinia Riccobene
Mark Ryan
Andrei Sabelfeld
Steve Schneider
Phil Scott
Roberto Segala
Robert de Simone
Eugene Stark
Gheorghe Ştefănescu
Maciej Szreter
Jean-Pierre Talpin
P. S. Thiagarajan
Emilio Tuosto
Andrea Valente
Moshe Vardi
Björn Victor
Marc Voorhoeve
Michal Walicki
Heike Wehrheim

Harro Wimmel
Wang Yi
Nobuko Yoshida
Wieslaw Zielonka
Pascal Zimmer

Table of Contents

Combining Theorem Proving and Model Checking through Symbolic Analysis*

Natarajan Shankar

Computer Science Laboratory, SRI International
Menlo Park CA 94025 USA
Phone: +1 (650) 859-5272
shankar@csl.sri.com
http://www.csl.sri.com/~shankar/

Abstract. Automated verification of concurrent systems is hindered by the fact that the state spaces are either infinite or too large for model checking, and the case analysis usually defeats theorem proving. Combinations of the two techniques have been tried with varying degrees of success. We argue for a specific combination where theorem proving is used to reduce verification problems to finite-state form, and model checking is used to explore properties of these reductions. This decomposition of the verification task forms the basis of the Symbolic Analysis Laboratory (SAL), a framework for combining different analysis tools for transition systems via a common intermediate language. We demonstrate how symbolic analysis can be an effective methodology for combining deduction and exploration.[1]

The verification of large-scale concurrent systems poses a difficult challenge in spite of the substantial recent progress in computer-aided verification. Technologies based on model checking [CGP99] can typically handle systems with states that are no larger than about a hundred bits. Techniques such as symmetry and partial-order reductions, partitioned transition relations, infinite-state model checking, represent important advances toward ameliorating state explosion, but they have not dramatically increased the overall effectiveness of automated verification. Model checking does have one advantage: it needs only a modest amount of human guidance in terms of the problem description, possible variable orderings, and manually guided abstractions. Verification based on theorem proving, on the other hand, requires careful human control by way of suitable intermediate assertions, invariants, lemmas, and proofs. Can automated verification ever combine the automation of model checking with the generality of theorem proving?

It has often been argued that model checking and theorem proving could be combined so that the former is applied to control-intensive properties while the

* This work was funded by DARPA Contract No. F30602-96-C-0204 Order No. D855 and NSF Grants No. CCR-9712383 and CCR-9509931.

[1] The SAL project is a collaborative effort between Stanford University, SRI International, and the University of California, Berkeley.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 1–16, 2000.

latter is invoked on data-intensive properties. Achieving an integration of theorem proving and model checking is not hard. Both techniques verify claims that look similar and it is possible to view model checking as a decision procedure for a well-defined fragment of a specification logic [RSS95]. However, most systems contain a rich interaction between control and data so that there is no simple decomposition between data-intensive and control-intensive properties.

For the purpose of this paper, we view model checking as a technique for the verification of temporal properties of a program based on the exhaustive exploration of a transition graph represented in explicit or symbolic form. Model checking methods typically use graph algorithms, automata-theoretic constructions, or finite fixed point computations. Theorem proving is usually based on formalisms such as first-order or higher-order logic, and employs proof techniques such as induction, rewriting, simplification, and the use of decision procedures. Some infinite-state verifiers and semi-decision procedures can be classified as both deductive and model checking techniques, but this ambiguity can be overlooked for the present discussion.

We make several points regarding the use of theorem proving and model checking in the automated verification of concurrent systems:

1. *Correctness is over-rated.* The objective of verification is analysis, i.e., the accretion of useful observations regarding a system. Verifying correctness is an important form of analysis, but correctness is usually a big property of a system that is demonstrated by building on lots of small observations. If these small observations could be cheaply obtained, then the demonstration of larger properties would also be greatly simplified. The main drawback of correctness is its exactitude. The verification of a correctness claim can only either fail or succeed. There is no room for approximate answers or partial information.
2. *Theorem proving is under-rated.* Deduction remains the most appropriate technology for obtaining insightful, general, and reusable automation in the analysis of systems, particularly those that are too complex to be analyzed by a blunt instrument like model checking. Theorem proving can exploit the mathematical properties of the control and data structures underlying an algorithm in their fullest generality and abstractness
3. *Theorem proving and model checking are very similar techniques.* In the verification of transition systems, both techniques employ some representation for program assertions, they compute the image of the transition relation with respect to these assertions, and usually try to compute the least, greatest, or some intermediate fixed point assertion for the transition relation. The difference is that in theorem proving,
   - The image constructions are usually more complicated since they involve quantification in domains where quantifier elimination is either costly or impossible.
   - The least and greatest fixed points can seldom be effectively computed and human guidance is needed to suggest an intermediate fixed point.
   - Showing that one assertion is the consequence of another is typically undecidable and requires the use of lemmas and human insight.

4. *Theorem proving and model checking can be usefully integrated.* Such an integration requires a methodology that decomposes the verification task so that
   - *Deduction is used to construct valid finite-state abstractions of a system.* The construction of a property-preserving abstraction generates simple proof obligations that can be discharged, often fully automatically, using a theorem prover. These are typically assertions of the form: if property p holds in a state s from which there is a transition R to a state s', then property q holds in s'. Similar proof obligations arise during verification (in the form of verification conditions) but these are usually not valid and the assertions have to be strengthened in order to obtain provable verification conditions. While theorem proving is useful for examining the local consequence of properties, it is not very effective at deducing global consequences over a large program or around an iterative loop. Such computations can be extremely inefficient and the computation of fixed points around a loop rarely terminates.
   - *Exploration by means of model checking is used to calculate global properties of such abstractions.* This means that model checking is not used merely to validate or refute putative properties but is actually used to calculate interesting invariants that can be extracted from the reachability predicate or its approximations. Finite-state exploration of large structures can also be inefficient but it is much easier to make finite-state computations converge efficiently.
   - *Deduction is used to propagate the consequences of such properties.* For example, model checking on a finite-state abstraction might reveal an assertion $x > 5$ to hold at a program point simply because it was true initially and none of the intermediate transitions affected the value of x. If the program point has a successor state that can only be reached by a transition that increments x by 2, then we know that this successor state must satisfy the assertion $x > 7$. Such a consequence is easily deduced by theorem proving.

In summary, we advocate a verification methodology where deduction is employed in the local reasoning steps such as validating abstractions and propagating known properties, whereas model checking is used for deriving global consequences. In contrast, early attempts to integrate theorem proving and model checking were directed at using model checking as a decision procedure within a theorem prover. These attempts were not all that successful because it is not common to find finite-state subgoals within an infinite-state deductive verification.

1 Background

We review some of the background and previous work in the combined use of theorem proving and model checking techniques.

1.1 Model Checking as a Decision Procedure

Joyce and Seger combined the theorem prover HOL [GM93] with the symbolic trajectory evaluation tool Voss [JS93] by treating the circuits verified by Voss as uninterpreted constants in HOL. This integration is somewhat *ad hoc* since the definitions of the circuits verified by Voss are not available to HOL. Dingel and Filkorn [DF95] use a model checker to establish assume–guarantee properties of components and a theorem prover to discharge the proof obligations that arise when two components are composed. Rajan, Shankar, and Srivas [RSS95] integrate a mu-calculus [Par76,BCM+92] model checker [Jan93] as a decision procedure for a fragment of the PVS higher-order logic corresponding to a finite mu-calculus. While this integration smoothly incorporates CTL and LTL model checking into PVS, the work needed to reduce a problem into model-checkable form can be substantial. This integration has recently been extended with an algorithm for constructing finite-state abstractions of mu-calculus expressions [SS99].[2]

1.2 Extending Model Checking with Lightweight Theorem Proving

Several alternative approaches to the integration of model checking and theorem proving have emerged in recent years. Some of these have taken the approach of supplementing a model checker with a proof assistant that provides rules for decomposing a verification goal into model-checkable subgoals. McMillan [McM99] in his work with Cadence SMV has extended the SMV model checker with the following decomposition rules that are used to reduce infinite-state systems to model-checkable finite-state ones.

1. *Temporal splitting*: Transforms a goal of the form $\Box(\forall i : A)$ into $\Box v = i \supset A$ for each i.
2. *Symmetry reduction*: Typically, the system being verified and the property are symmetric in the choice of i so that proving $\Box v = i \supset A$ for a single specific value for i is equivalent to proving it for each i. Examples of such symmetric choices include the memory address or the processor in the verification of multiprocessor cache consistency.
3. *Data abstraction*: Large or infinite datatypes can be reduced to small finite datatypes by suitably reinterpreting the operations on these datatypes. For example, with respect to the choice of i in temporal splitting, the remaining values of the datatype can be abstracted by a single value *non-i*.
4. *Compositional verification*: The verification of $P \| Q \models A \wedge B$ is decomposed as $P \models \neg(B \ \mathbf{U} \ \neg A)$ (B fails before A does) and $Q \models \neg(A \ \mathbf{U} \ \neg B)$. This allows different components to be separately verified up to time $t + 1$ by assuming the other components to be correct up to time t.

These and other proof techniques have been used to verify an out-of-order processor, a large cache coherence algorithm, and safety and liveness for a version of Lamport's N-process bakery algorithm for mutual exclusion [MQS00].

[2] These features are part of PVS 2.3 which is accessible at the URL `pvs.csl.sri.com`.

McMillan's approach is substantially deductive. The rules of inference, such as symmetry reduction and compositional verification, are specialized but quite powerful.

Seger [Seg98] has extended the Voss tool for symbolic trajectory evaluation with lightweight theorem proving. Symbolic trajectory evaluation (STE) which is a limited form of linear temporal logic model checking. A few simple proof rules are used to decompose proof obligations on the basis of the logical connectives such as conjunction, disjunction, and implication. These rules can be used to decompose a large model checking problem into smaller ones.

1.3 Abstraction and Model Checking

Abstraction has been studied in the context of model checking as a technique for reducing infinite-state or large finite-state models to finite-state models of manageable size [BBLS92,Kur94,CGL94,LGS+95,Dam96,BLO98].

Some of the work on abstraction is based on *data abstraction* where a variable X over a concrete datatype T is mapped to a variable x over an abstract type t. For example, a variable over the natural numbers could be replaced by a boolean variable representing the parity of its value. Clarke, Grumberg, and Long [CGL94] gave a simple criterion for abstractions that preserve $\forall CTL^{*+}$ properties. Let the concrete transition system be given by $\langle I_C, N_C \rangle$ where I_C is the initialization predicate and N_C is the next-state relation. Then the verification of a concrete judgement $\langle I_C, N_C \rangle \models P_C$ can be reduced by means of the abstraction function α to the verification of an abstract judgement $\langle I_A, N_A \rangle \models P_A$ provided

1. $I_C \sqsubseteq I_A \circ \alpha$
2. $N_C \sqsubseteq N_A \circ \langle \alpha, \alpha \rangle$
3. $P_A \circ \alpha \sqsubseteq P_C$

Data abstraction has the advantage that the abstract description can be statically constructed from the concrete program. The drawback is that many useful abstractions are on relations between variables rather than on individual variables.

Graf and Saïdi [SG97] introduced *predicate abstraction* as a way of replacing predicates or relations over a set of variables by the corresponding boolean variables. For example, given two variables x and y over the integers, and the predicate $x < y$ over these variables, predicate abstraction would replace the variables x and y by a boolean variable b that represents the behavior of the predicate.

The application of predicate abstraction makes significant use of theorem proving. Graf and Saïdi used predicate abstraction to construct an abstract reachability graph for a concrete program by a process of elimination. If a represent an abstract state, a' a putative successor, $\gamma(a)$ the concrete state corresponding to a, and $\gamma(a')$ the concrete state corresponding to a', then if

$$\gamma(a) \supset wp(P)(\neg(\gamma(a')))$$

is provable, the corresponding transition between a and a' can be ruled out.[3] However, if a proof attempt fails, the corresponding successor node can be conservatively included in the abstract reachability graph. Using predicate abstractions with the PVS theorem prover [ORS92], Graf and Saïdi [SG97] were able to verify a variant of the alternating bit protocol called the bounded retransmission protocol [HSV94]. Das, Dill, and Park [DDP99] extended this technique using the SVC decision procedures [BDL96] and were able to verify such impressive examples as the FLASH cache coherence protocol, and a cooperative garbage collector.

Predicate abstraction can also be used to construct an abstract transition relation instead of the abstract reachability graph. It is typically less expensive to construct the abstract transition relation since fewer proof obligations are generated, but it typically results in a coarser abstraction than one that is obtained by directly computing the abstract reachability graph. In the latter construction, information about the current set of abstract reachable states can be used to rule out unreachable successor states. Bensalem, Lakhnech, and Owre [BLO98] describe an abstraction tool called InVeSt that uses the elimination method to construct an abstract transition system from a concrete one in a compositional manner. Colon and Uribe [CU98] give another compositional method for constructing abstractions with the framework of the STeP theorem prover [MtSG95].

All of the above abstraction techniques preserve only $\forall CTL^{*+}$ properties, namely those in the positive fragment of CTL^* with universal path quantification. For more general calculi, criteria for abstractions that preserve CTL* [DGG94] and mu-calculus [LGS+95], but these results are quite technical. Saïdi and Shankar [SS99] gave a simple method for constructing predicate abstractions over the full relational mu-calculus [Par76]. The two key observations in this work are:

1. The operators of the mu-calculus are monotonic with respect to upper and lower approximations.
2. The over-approximation of a literal (an atomic formula or its negation) can be efficiently computed in conjunctive normal form by using a theorem prover as an oracle.

Verification diagrams [MBSU99] can also be seen as a form of predicate abstraction. These diagrams employ graphs whose nodes are labeled by assertions and the edges correspond to program transitions within the diagram. Properties can be directly checked with respect to the verification diagram.

The primary advantage of predicate abstraction is that it is sufficient to guess a relevant predicates without having to guess the exact invariant in these predicates. For n predicates, the construction of the abstract transition system

[3] All programs are assumed to be total as transition system, i.e., the domain of the next-state relation is the set of all states. Thus, $wp(P)(A)$ is the set of states that have no transitions in P to states in $\neg A$. The dual notion $sp(P)(A)$ is the set of states reachable from some state in A by a transition of P.

generates of the order of 2^n proof obligations. The resulting abstract model can also be model checked in time that is exponential in n to yield useful invariants. With deduction, there are 2^{2^n} boolean functions that are candidate invariants in these n predicates so that it is harder to guess suitable invariants.

1.4 Automatic Invariant Generation

Automatic invariant generation has been studied since the 1970s [CH78,GW75,KM76,SI77]. This study has recently been revived through the work of Bjørner, Browne, and Manna [BBM97], and Bensalem, Lakhnech, and Saïdi [BLS96,Saï96,BL99].

The strongest invariant of a transition system P is given by the least fixed point starting from the initial states of P of the strongest postcondition operator for P, $\mu X.I_P \vee sp(P)(X)$. If this computation terminates, it would yield the set of reachable states of P which is its strongest invariant. Unfortunately, the least fixed point computation rarely terminates for infinite-state systems. A program with a single integer variable x that is initially 0 and is repeatedly incremented by one, yields a nonterminating least fixed point computation. Widening techniques [CC77] are needed to accelerate the fixed point computation so that it does terminate with a fixed point that is not necessarily the least one.

A different, more conservative approach to invariant generation is given by the computation of the greatest fixed point of the strongest postcondition $\nu X.sp(P)(X)$. For example, a greatest fixed point computation on a program with a single variable x and a single guarded transition $x \geq 0 \longrightarrow x := x + 1$ would terminate and yield the invariant $x \geq 0$. The greatest fixed point invariant computation also may not terminate and could require *narrowing* as a way of accelerating termination. However, one could stop the greatest fixed point computation after any bounded number of iterations and the resulting predicate would always be a valid invariant.

Dually, a putative invariant p can be strengthened to an inductive one by computing the greatest fixed point with respect to the weakest precondition of the program of the given invariant $\nu X.p \wedge wp(P)(X)$. If this computation terminates, the result is an invariant that is inductive.

Automatic invariant generation is not yet a successful technology. Right now, it is best used for propagating invariants that are computed from other sources by taking the greatest fixed point with respect to the strongest post-condition starting from a known invariant. However, as theorem proving technology becomes more powerful and efficient, invariant generation is likely to be quite a fruitful technique.

2 Symbolic Analysis

Symbolic analysis is simply the computation of fixed point properties of programs through a combination of deductive and explorative techniques. We have already seen the key elements of symbolic analysis as

1. *Automated deduction*, in computing property preserving abstractions and propagating the consequences of known properties.
2. *Model checking*, as a means of computing global properties of by means of systematic symbolic exploration. For this purpose, model checking is used for actually computing fixed points such as the reachable state set, in addition to verifying given temporal properties.
3. *Invariant generation*, as a technique for computing useful properties and propagating known properties.

2.1 SAL: A Symbolic Analysis Laboratory

SAL is a framework for integrating different symbolic analysis techniques including theorem proving and model checking. The core of SAL is a description language for transition systems. The design of this intermediate language has been influenced by SMV [McM93], UNITY [CM88], Murphi [MD93], and Reactive Modules [AH96]. Transition systems described in SAL consist of modules with input, output, global, and local variables. Initializations and transitions can be either specified by definitions of the form *variable* = *expression* or by guarded commands. The assignment part of a guarded command consists of assignments of the form $x' = expression$, meaning the new value of x is the value of the *expression*, as well as selections $x' \in set$, meaning the new value of x is nondeterministically selected from the value of the nonempty set *set*. SAL is a synchronous language in the spirit of Esterel [BG92], Lustre [HCRP91], and Reactive Modules [AH96], in the sense that transitions can depend on latched values as well as current inputs. SAL modules can be composed by means of

1. Binary synchronous composition $P \| Q$ whose transitions consist of lock-step parallel transitions of P and Q.
2. Binary asynchronous composition $P \,[]\, Q$ whose transitions are the interleaving of those of P and Q.
3. N-fold synchronous composition $(\| \ (i) : \ P[i])$
4. N-fold asynchronous composition $([] \ (i) : \ P[i])$

The implementation of SAL is still ongoing. The version to be released some time in 2000 will consist of a parser, typechecker, translators to SMV and PVS, a translator to Java (for animation), and a translator from Verilog, among other tools.

Since the SAL implementation is still incomplete, we informally describe some examples that motivate the need for a symbolic analysis framework integrating abstraction, invariant generation, theorem proving, and model checking.

2.2 Analysis of a Two Process Mutual Exclusion Algorithm

As a first example, we use a simplified 2-process version of Lamport's Bakery algorithm for mutual exclusion [Lam74]. The algorithm consists of two processes P and Q with control variables *pcp* and *pcq*, respectively, and shared variables x

and y. The control states of these processes are either `sleeping`, `trying`, or `critical`. Initially, pcp and pcq are both set to `sleeping` and the control variables satisfy $x = y = 0$. The transitions for P are

$$\begin{array}{rl} & \texttt{pcp} = \mathit{sleeping} \longrightarrow x' = y + 1;\ \mathit{pcp}' = \texttt{trying} \\ [] \ \mathit{pcp} = \texttt{trying} \wedge (y = 0 \vee x < y) & \longrightarrow \mathit{pcp}' = \texttt{critical} \\ [] & \mathit{pcp} = \texttt{critical} \longrightarrow x' = 0;\ \texttt{pcp}' = \texttt{sleeping} \end{array}$$

Similarly, the transitions for Q are

$$\begin{array}{rl} & \mathit{pcq} = \texttt{sleeping} \longrightarrow y' = x + 1;\ \mathit{pcq}' = \texttt{trying} \\ [] \ \mathit{pcq} = \texttt{trying} \wedge (x = 0 \vee y \leq x) & \longrightarrow \mathit{pc}' = \texttt{critical} \\ [] & \mathit{pcq} = \texttt{critical} \longrightarrow y' = 0;\ \mathit{pcq}' = \texttt{sleeping} \end{array}$$

The invariant we wish to establish for $P \,[]\, Q$ is $\neg(\mathit{pcp} = \texttt{critical} \wedge \mathit{pcq} = \texttt{critical}$. Note that $P \,[]\, Q$ is an infinite-state system and in fact the values of the variables x and y can increase without bound. We can therefore attempt to verify the invariant by means of a property-preserving predicate abstraction to a finite-state system.

The abstraction predicates suggest themselves from the initializations, guards, and assignments. We therefore abstract the predicate $x = 0$ with the boolean variable x_0, the predicate $y = 0$ with the boolean variable y_0, and the predicate $x < y$ with the boolean variable xy. The resulting abstract system can be computed as P' and Q', where in the initial state, $x_0 \wedge y_0 \wedge \neg xy$, and the transitions for P' are

$$\begin{array}{rl} & \mathit{pcp} = \mathit{sleeping} \longrightarrow \mathtt{x}_0' = \textbf{false};\ xy' = \textbf{false};\ \mathit{pcp}' = \texttt{trying}; \\ [] \ \mathit{pcp} = \texttt{trying} \wedge (y_0 \vee xy) & \longrightarrow \mathit{pcp}' = \texttt{critical}; \\ [] & \mathit{pcp} = \texttt{critical} \longrightarrow x_0' = \textbf{true}; xy' \in \{\textbf{true}, \textbf{false}\}; \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pcp}' = \texttt{sleeping}; \end{array}$$

The transitions for Q' are

$$\begin{array}{rl} & \mathit{pcq} = \texttt{sleeping} \longrightarrow y_0' = \textbf{false};\ xy' = \textbf{true};\ \mathit{pcq}' = \texttt{trying}; \\ [] \ \mathit{pcq} = \texttt{trying} \wedge (x_0 \vee \neg xy) & \longrightarrow \mathit{pcp}' = \texttt{critical}; \\ [] & \mathit{pcp} = \texttt{critical} \longrightarrow y_0' = \textbf{true}; xy' = \textbf{false};\ \mathit{pcp}' = \texttt{sleeping}; \end{array}$$

Model checking the abstract system $P' \,[]\, Q'$ easily verifies the invariant

$$\neg(\mathit{pcp} = \texttt{critical} \wedge \mathit{pcq} = \texttt{critical}).$$

The theorem proving needed to construct the abstraction is at a trivial level that can be handled automatically by the decision procedures over quantifier-free formulas in a combination of theories [RS00]. Such decision procedures are present in systems like PVS [ORS92], ESC [Det96], SVC [BDL96], and STeP [MtSG95]. The above example can be verified fully automatically by means of the `abstract-and-model-check` command in PVS [SS99].

2.3 Analysis of an N-Process Mutual Exclusion Algorithm

We next examine a fictional example, namely, one that has not been mechanically verified by us. This example is a simplified form of the N-process Bakery algorithm due to Lamport [Lam74]. The description below shows a hand-executed symbolic analysis.

In this version of the Bakery algorithm, there are N processes $P(0)$ to $P(N-1)$, with a shared array x of size N over the natural numbers. The logical variables i, j, and k range over the subrange $0..(N-1)$. The operation $max(x)$ returns the maximal element in the array x. Initially, each $P(i)$ is in the control state `sleeping`, and for each i, $x(i) = 0$. Let $\langle x, i\rangle \leq \langle y, j\rangle$ be defined as the lexicographic ordering $x < y \vee (x = y \wedge i \leq j)$. We abbreviate $y = 0 \vee \langle x, i\rangle \leq \langle y, j\rangle$ as $\langle x, i\rangle \preceq \langle y, j\rangle$.

The transitions of processes $P(i)$ for $0 \leq i < N$ are interleaved and each non-stuttering transition executes one of the following guarded commands.

$$\begin{array}{rl} pc(i) = sleeping \longrightarrow & x'(i) = 1 + max(x); \\ & pc'(i) = \texttt{trying}; \\ [] \quad pc(i) = \texttt{trying} & \longrightarrow pc'(i) = \texttt{critical}; \\ \quad \wedge\ (\forall j : \langle x(i), i\rangle \preceq \langle x(j), j\rangle) & \\ [] \quad pc(i) = critical \longrightarrow & x'(i) = 0; \\ & pc'(i) = \texttt{sleeping}; \end{array}$$

We want to prove the invariance property

$$(\forall i : pc(i) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset i = j)). \tag{1}$$

Invariant generation techniques can be used to generate trivial invariants such as

$$(\forall i : x(i) = 0 \ \textit{iff}\ pc(i) = \texttt{sleeping}). \tag{2}$$

We omit the details of the invariant generation step. The above invariant will prove useful in the next stage of the analysis.

We next skolemize the mutual exclusion statement so as to obtain a correctness goal about a specific but arbitrary i which we call a. The main invariant now becomes

$$pc(a) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset a = j) \tag{3}$$

The goal now is to reduce the N-process protocol to a two process protocol consisting of process a and another process b that is an *existential abstraction* of the remaining $N-1$ processes. By an existential abstraction, we mean one where the $N-1$ processes are represented by a single process b such that a transition by any of the $N-1$ processes is mapped to a corresponding transition of b. In such an abstraction, b is in control state `critical` if any one of the $N-1$ processes is critical. Otherwise, b is in control state `trying` if none of the $N-1$ processes is in the state `critical` and at least one of them is in its `trying` state. If none of the $N-1$ process is either `trying` or `critical`, then b is in its `sleeping` state.

By examining the predicates appearing in the initialization, guards, and the property, we can directly obtain the following abstraction predicates given by the function γ which maps abstract variables to the corresponding concrete predicates:

$$\begin{aligned}
\gamma(pca) &= pc(a) \\
\gamma(pcb) &= \mathit{if} \quad (\exists j : j \neq a \wedge pc(j) = \texttt{critical}) \\
&\quad \mathit{then}\ \texttt{critical} \\
&\quad \mathit{elsif}\ (\exists j : j \neq a \wedge pc(j) = \texttt{trying}) \\
&\quad \mathit{then}\ \texttt{trying} \\
&\quad \mathit{else}\ \texttt{sleeping} \\
\gamma(xa_0) &= (x(a) = 0) \\
\gamma(xb_0) &= (\forall j : j \neq a \supset x(j) = 0) \\
\gamma(ma) &= (\forall j : \langle x(a), a\rangle \preceq \langle x(j), j\rangle) \\
\gamma(mb) &= (\exists j : (\forall k : \langle x(j), j\rangle \preceq \langle x(k), k\rangle) \\
\gamma(ea) &= (\forall j : pc(j) = \texttt{critical} \supset a = j)
\end{aligned}$$

Since mb is only relevant when $pc(j) = \mathit{trying}$ for $j \neq a$, we can use invariant (2) to prove that

$$j \neq a \wedge pc(j) \neq \mathit{sleeping} \supset \gamma(mb) = \gamma(\neg ma)$$

thereby dispensing with mb in the abstraction.

With the above abstraction mapping, the goal invariant (3) becomes

$$pca = \texttt{critical} \supset ea.$$

and the resulting abstracted transition system is one where initially

$$pca = \mathit{sleeping} \wedge pcb = \mathit{sleeping} \wedge xa_0 \wedge xb_0 \wedge ma \wedge ea$$

Each non-stuttering step in the computation of the abstract program executes one of the guarded commands shown in Figure 1.

Model checking the abstract protocol fails to verify the invariant

$$pca = \mathit{critical} \supset ea$$

as the model checker could generate the following counterexample sequence of transitions:

transition	*pca*	*xa*	*ma*	*ea*	*pcb*	*xb*
initially	*sleeping*	*true*	*true*	*true*	*sleeping*	*true*
3	*sleeping*	*true*	*false*	*true*	*trying*	*false*
4	*sleeping*	*true*	*false*	*false*	*critical*	*false*
1	*trying*	*false*	*false*	*false*	*critical*	*false*
8	*trying*	*false*	*true*	*false*	*critical*	*false*
2	*critical*	*false*	*true*	*false*	*critical*	*false*

$$
\begin{array}{rll}
 & pca = \texttt{sleeping} \longrightarrow & xa' = \textbf{false}; \\
 & & ma' = xb; \\
 & & pca' = \texttt{trying}; \\
[] & pca = trying \wedge ma \longrightarrow & pca' = \texttt{critical}; \\
[] & pca = \texttt{critical} \longrightarrow & pca' = \texttt{sleeping}; \\
 & & ma' = xb; \\
 & & ea' = \neg(pcb = \texttt{critical}); \\
 & & xa' = \textbf{true}; \\
[] & pcb = \texttt{sleeping} \longrightarrow & pcb' = \texttt{trying};\ xb' = \textbf{false}; \\
 & & ma' = \neg xa \\
[] & pcb = \texttt{trying} \wedge \neg ma \longrightarrow & pcb' = \texttt{critical};\ ea' = \textbf{false}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{sleeping}; \\
 & & ea' = \textbf{true}; \\
 & & ma' = \textbf{true}; \\
 & & xb' = \textbf{true}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{trying}; \\
 & & ea' = true; \\
 & & ma' \in \{\textbf{true}, ma\}; \\
[] & pcb = \texttt{critical} \longrightarrow & ma' \in \{\textbf{true}, ma\};
\end{array}
$$

Fig. 1. Abstract transitions for the N-process Bakery Algorithm

An inspection of the counterexample and the abstract model confirms that the mutual exclusion invariant would follow if the invariant $\neg xa \wedge ma \supset ea$ were to hold. Mapped back in the concrete domain, this corresponds to

$$\forall i : x(i) \neq 0 \wedge (\forall j : x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle) \supset (\forall j : pc(j) = \texttt{critical} \supset i = j).$$

This goal can be generalized as

$$(\forall i, j : x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle) \leq \langle x(j), j\rangle \supset (pc(j) = \texttt{critical} \supset i = j)).$$

and further rearranged as

$$(\forall i, j : pc(j) = \texttt{critical} \supset (x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle)) \supset i = j).$$

By the invariant (2), we can eliminate the subformula $x(j) = 0$ and simplify the goal to the equivalent formula

$$(\forall i, j : pc(j) = critical \supset x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle).$$

This can be rearranged as

$$(\forall j : pc(j) = critical \supset (\forall i : x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle)).$$

But this is the just the invariant $pca = critical \supset ma$ which is already implied by the abstract model.

The safety property is thus verified by using a judicious combination of a small amount of theorem proving and model checking. The abstractions were suggested by the predicates in the text of the program. Simple invariant generation methods were adequate for generating trivial invariants. Theorem proving in the context of these invariants could be used to discharge the proof obligations needed to construct an accurate abstraction of the N-process protocol. Abstraction mappings of this sort are quite standard and work for many mutual exclusion and cache consistency algorithms [Sha97]. The abstract model did not discharge the main safety invariant but it was easy to extract the minimal condition needed to verify the invariant from the abstract model. A reachability analysis of the abstract model delivered enough useful invariants so that a small amount of theorem proving could discharge this condition. Neither the model checking nor the theorem proving used here is especially difficult. While some guidance is needed in selecting lemmas and conjectures, the proofs of these can be carried out with substantial automation.

3 Conclusion

We have argued that verification technology is best employed as an analysis technique to generate properties of specifications and programs rather than as a method for establishing the correctness of specific properties. Such a symbolic analysis framework can employ both theorem proving and model checking as appropriate to generate useful abstractions and automatically derive system properties.

Many ideas remain to be explored within the symbolic analysis framework. The construction of the symbolic analysis laboratory SAL as an open framework will support the exploration of ideas at the interface of theorem proving and model checking.

Acknowledgments

Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari. John Rushby read an earlier draft of the paper and suggested numerous improvements.

References

AH96. Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press. 8

BBLS92. Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Computer-Aided Verification, CAV '92*, volume 630 of *Lecture Notes in Computer Science*, pages 260–273, Montréal, Canada, June 1992. Springer-Verlag. Extended version available with title "Property Preserving Abstractions.". 5

BBM97. Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997. 7

BCM+92. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992. 4

BDL96. Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag. 6, 9

BG92. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 8

BL99. Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999. 7

BLO98. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331. 5, 6

BLS96. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/August 1996. Springer-Verlag. 7

CC77. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis. In *4th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1977. 7

CGL94. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. 5

CGP99. E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999. 1

CH78. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978. 7

CM88. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988. 8

CU98. M. A. Col on and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [HV98], pages 293–304. 6

Dam96. Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996. 5

DDP99. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99], pages 160–171. 6

Det96. David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, San Diego, CA, January 1996. Association for Computing Machinery. 9

DF95. Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Computer-Aided Verification 95*, 1995. This volume. 4

DGG94. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994. 6

GM93. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993. 4

GW75. S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975. 7

HCRP91. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 8

HP99. Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag. 14, 16

HSV94. L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, March 1994. 6

HV98. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag. 14

Jan93. G. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993. 4

JS93. Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. Association for Computing Machinery, 1993. 4

KM76. S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976. 7

Kur94. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994. 5

Lam74. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974. 8, 10

LGS+95. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995. 5, 6

MBSU99. Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41, Amazonia, Brazil, January 1999. Springer-Verlag. 6

McM93. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993. 8

McM99. K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, number 1703 in Lecture Notes in Computer Science, pages 219–233. Springer Verlag, September 1999. 4

MD93. Ralph Melton and David L. Dill. *Murφ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993. 8

MQS00. K. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, 2000. To appear. 4

MtSG95. Z. Manna and the STeP Group. STeP: The Stanford Temporal Prover. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 793–794, Aarhus, Denmark, May 1995. Springer Verlag. 6, 9

ORS92. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artifical Intelligence*, pages 748–752. Springer Verlag, June 1992. 6, 9

Par76. David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976. 4, 6

RS00. H. Rueß and N. Shankar. Deconstructing Shostak. Available from http://www.csl.sri.com/shankar/shostak2000.ps.gz., January 2000. 9

RSS95. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag. 2, 4

Saï96. Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996. Springer-Verlag. 7

Seg98. Carl-Johan H. Seger. Formal methods in CAD from an industrial perspective. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag. 5

SG97. Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag. 5, 6

Sha97. N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997. 13

SI77. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977. 7

SS99. Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [HP99], pages 443–454. 4, 6, 9

Verification Is Experimentation!

Ed Brinksma

Chair of Formal Methods and Tools
Faculty of Computer Science, University of Twente
PO Box 217, 7500AE Enschede, Netherlands
brinksma@cs.utwente.nl
http:/home.cs.utwente.nl/~/brinksma

Abstract. The formal verification of concurrent systems is usually seen as an example par excellence of the application of mathematical methods to computer science. Although the practical application of such verification methods will always be limited by the underlying forms of combinatorial explosion, recent years have shown remarkable progress in computer aided formal verification. They are making formal verification a practical proposition for a growing class of real-life applications, and have put formal methods on the agenda of industry, in particular in the areas where correctness is critical in one sense or another. Paradoxically, the results of this progress provide evidence that successful applications of formal verification have significant elements that do not fit the paradigm of pure mathematical reasoning. In this essay we argue that verification is part of an experimental paradigm in at least two senses. We submit that this observation has consequences for the ways in which we should research and apply formal methods.

1 A Little History

The roots of formal verification lie in the observation, which broke ground in the nineteen-sixties and -seventies, that software programs can be seen as *formal*, i.e. *mathematical* objects. This led to extensive studies of formal semantical models for programming languages, and the development of (academic) programming languages for which nice formal models could be guaranteed to exist. This activity generated a deeper understanding of the structure of computer programs and provided the foundations for many of the mathematical tools and models that are known as formal methods today.

In addition to the development of mathematical models that would help to give a scientific foundation to programming, this period was also marked by a strong methodological interpretation of these achievements that promoted the view of programming as an essentially mathematical activity. If programs are mathematical objects and specifications of their intended functionality properly formalised, then their correctness can be demonstrated by mathematical means. Much effort was directed towards the development of programming calculi in which the development of programs can be seen as a form of *equation solving*.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 17–24, 2000.

The beauty and strength of this vision were so compelling that they dominated the scientific research agenda for programming for many years. The movement as such acquired, perhaps inescapably, also some ideological traits, in the sense that it was less forgiving of practical programming as practised in industry, and the program validation methods used there, most notably testing. The cure for the diagnosed *software crisis* was thought to be found in the education of a new generation of academically trained programmers that would introduce the mathematical methods of programming into industry.

Although the mathematical school of programming has thoroughly influenced the the study of programming, programming languages, specification, etc., it is now generally acknowledged that the mathematical theory of programming cannot be applied as was originally envisaged. Many arguments have been put forward to explain why it did not or could not work (see e.g. [7]), including circumstantial technical, economical, sociological and educational reasons that we will not consider here. An intrinsic reason for failure that was initially overlooked, is the retrospectively almost obvious fact that in general a correctness proof has a complexity proportional to that of the program involved. This causes direct problems in scaling up the method to deal with complicated software systems, which was, of course, the ultimate goal. Proofs or derivations of such systems would be very complicated and therefore susceptible to errors themselves, directly undermining the essential contribution of the method.

The law of conservation of misery has made sure that there are no easy solutions to the problem. The term *formal methods* became commonplace somewhere in the nineteen-eighties to indicate the assorted formal notations, theories and models that had been developed to help specify, implement amd analyse software systems. As by then not only sequential, but also concurrent and reactive systems could be mathematically described and analysed in terms of elegant mathematical models, a second wave of methodological optimism swept through academia and parts of industry. Having at our disposal methods for the formal description and manipulation of algorithms, concurrent interaction and data, it was believed that the design and the development of a proof of correctness of complex systems could be a *shared* activity, known as *correctness by design*. Moreover, these integral *broad spectrum* formalisms would be supported by powerful software environments to support the design and verification with the required precision, thus solving the problem of controlling the precision of complicated (or perhaps better: lengthy) formal manipulations. The failure of this second formalist attack on the software crisis was again due to many, diverse causes. Again there was one important intrinsic reason: the formal objects that corresponded to descriptions of (parts of) the systems designs were so large that they could no longer be manipulated effectively, not even by software tools. This phenomenon became known as the *combinatorial explosion*, or in the paticular case of the explicit manipulation of system states as the *state space explosion*.

2 Computer Aided Verification

During the past few years formal methods, and in particular formal verification, are again drawing the attention of the research community and (parts of) industry. This time it is not the result of a methodological movement, but the result of technological advances and research in the field of computer aided verification. For the first time it has proved possible to formally verify parts of nontrivial systems with practical consequences. This success in scaling up verification methods from toy examples to small-size real-life systems is the first hard evidence that the use of formal methods can, under circumstances, be made consistent with the requirements of industrial engineering.

Broadly speaking, computer aided verification can be categorised in two main streams: the *theorem proving* approach, which uses tools to produce completely formal proofs of correctness, and the *model checking* approach, which is basically a brute force approach to enumerate and check all reachable states of the system under verification. Both approaches can only work on the basis of a *model* of the system under verification. In theorem proving the model is a *logical theory* characterising the (relevant) properties of the system. In model checking the model must be operational so that systems states can be systematically produced, and usually takes the form an abstract program describing some sort of transition system.

Some reasons for the growing success of computer aided verification are:

1. The technological improvement of the necessary computing equipment. Because of the ever increasing performance of systems in terms of speed and available memory, computations that were far beyond the possible ten years ago are a matter of routine today.
2. The availability of serious tool environments. Much of the work on such environments has needed a considerable time to come to fruition, and effective tools start becoming available now. The development of good tools require sustained efforts over many years to develop stable architectures and profit from accumulating improvements.
3. The development of techniques to contain the effects of the combinatorial explosion. Abstraction techniques are used to strip away information in the model that is not relevant for the verification at hand, and lead to a simplification of verification models. Modular and compositional techniques apply a divide and conquer strategy to handle complexity by making formal manipulation local to well-defined parts of the model that are significantly smaller. In model checking substantial progress has also been achieved by the use of clever data types that allow for compactification, such as BDDs and the use hashing techniques.

Computer aided verification tends to have a rather pragmatic approach. What can be achieved depends more on the capacities and limitations of the tools that are being used, and less on methodological considerations. Because verification problems in their entirety are too large to handle, the process is

eclectic and concentrates on the essentials. This means that only crucial parts and properties of the system and its requirements, respectively, are formalised and verified. In practice this means that computer aided verification process is subject to a process of trial and error to determine how much of a system can be verified with the available resources.

As we are still in the early days of computer aided verification the knowledge about the effective scope of applications of the different tools and techniques is still very limited. Moreover, it appears to be difficult to generalise successful applications as we are sometimes confronted be *chaotic* behaviour, in the sense that small changes to a given problem may have big effects on the effectiveness of a particular verification technique.

Because of the substantial investments that must be made for computer aided verification for industrial applications, typical examples concern systems whose correct functioning is critical in a certain sense. This is not restricted to the so-called safety-critical systems, but applies more generally to systems for which the abstract or real cost of their malfunctioning is too high. Highly replicated systems (partly) implemented in hardware are a case in point. Embedded systems in consumer electronics and the automotive industry are good examples, as well as the verification of hardware chip designs.

Summarising, we can say that computer aided verification takes place in a context of experimentation. Different techniques are experimented with to increase the performance of the tools. Models and specifications are experimented with to see how much of a given problem can be verified. Typical examples of verification are found outside the world of pure software in interaction with more traditional engineering disciplines for which experimentation and measurement is an established method of quality control. In addition to this general experimental atmosphere that surrounds practical verification, there is another and more essential link between experimentation and verification.

3 Verification Needs Experimentation

Verification needs as its basis a formal model of the system that must be verified, the so-called *verification model*. As it plays a crucial role in the verification process, it can be said that a verification is as good as its underlying model. Obtaining valid verification models, i.e. models that faithfully represent the relevant properties of the objects they represent, therefore is a cornerstone of the verification process.

One of the strengths of the original paradigm of programs as mathematical objects is that a program text is (through its formal semantics) its own formal definition. The question of the validity of the formal model w.r.t. the reality of the physically executed program can be dealt with as a correctness requirement for the compiler (or interpreter) of the programming language. This has the advantage that the problem can be adressed and solved in generic terms, and the cost of producing a correct compiler can be amortised over all the programs that will be compiled.

As already indicated earlier, the situation for actual computer aided verifications can be radically different. If a complete, formal definition of the system under verification is available then it will usually be too big to serve as an effective verification model. This means that additional efforts are required to obtain smaller models that are valid for the verification task at hand.

A way to approach this question is to transform the original model into a smaller one, and demonstrate the validity of the result by proof or by construction. Indeed, the use of proof checkers for this purpose, in combination with model checkers for the verification proper, has been suggested as an elegant way to combine the strength of these two verification methods. Although this can be useful approach for specific classes of systems, it is less likely to be a generic solution to the problem, as it generally will bring us right back to the combinatorial explosion that we want to avoid.

In practice verification models are not formally derived or proved, but constructed on the basis of a combination of insight, heuristics, and sometimes formally well-defined abstractions. This principal loss of a formal link between the formal definition of a system and its effective verification model may be lamented, but it has a positive side to it. The availability of complete formal specifications of systems that we want to verify may help, but is no longer an absolute requirement. This is important as for complex systems such specifications are as a rule not available, and the cost of producing them is often prohibitively expensive. Smaller, more abstract specifications that suffice for the verification of some crucial correctness properties, however, could help to increase confidence in the correctness of a system in a more realistic price-performance ratio.

Also, it should be realised that the relation between formal specifications of complex systems and their realisations is more problematic than that between programs and their inplementations. Complex systems generally cannot be produced by just using reliable compiler(s), and often need elaborate engineering involving both hardware and software, requiring solutions that are unique to the given system. This implies that if we want to assess not only the correctness of a formal design, but actually want to analyse properties of the resulting system, its formal specification may not be the only relevant source of information.

We thus find ourselves in a situation in which the validity of many of our verification models cannot be demonstrated by formal means. This means that if we want to assess their validity, and we must if we take our job seriously, we can only use experimental methods. Moreover, this experimental validation is not just a phase born out of temporary necessity, but that it constitutes an essential methodological ingredient for the verification of real-life systems. As in physics, it is the tool to bridge the orders of magnitude that lie between a complete description of a system and an effective theory of its properties (cf. an extremely large set of molecules vs a volume of gas). It is worthwhile mentioning that in physics one can also quantify the consequences of such abstractions and show that the errors incurred are *sufficiently small*. In this respect it is interesting to note that in *performance analysis* often great simplifications of the evaluation models can be obtained without significant loss of precision. Such approximative

abstractions are not feasible if evaluation is restricted to evaluation in the binary system of classical logic. Stochastic interpretations of behaviour can perhaps provide a way forward in this respect: they would allow abstracting away from behaviours that are sufficiently unlikely.

Because the experimental paradigm is foreign to many who are active in the field of formal methods, its role is seldom explicitly addressed and if acknowledged, it is usually delegated to the engineers of "real" systems and the testing community. But the engineering of verification models is a task that requires intimate knowledge of the formal methods that are used, and therefore should concern all who are interested in the application of verification.

What is needed is agreement on what constitutes good verification practice. If we take our inspiration from the established experimental sciences, we should follow a protocol that includes the following ingredients:

- *Problem statement*: clearly defines the problem that is adressed. It answers questions like:
  - What is the system or design that must be verified?
  - What are the properties that must be verified?
  - What assumptions are made?
- *Verification set-up*: describes the ingredients of the verification, their use and relation accurately, so that all will be *repeatable* by others. Related questions are:
  - What verification model is used?
  - How are the properties formalised?
  - What tools and computing equipment are used? (versions, relevant system parameters)
  - What procedure was followed?
- *Measurements:* gives all the relevant data that were obtained.
- *Error discussion:* evaluates systematically all sources of errors that could have influenced the measurements. In particular, this section should address the quality of the verification model that was used.
- *Conclusion:* presents the final outcome of the verification. Generally, this is not a simple yes/no-answer, but an qualitative and/or quantitative interpretation of the measurements in the context of the error discussion.

Current verification practice is often opaque, not because it does not include activities to validate the verification model, but because it does not make them explicit and does not relate them via an error discussion to the results. What complicates matters is that the debugging of a verification model is often interleaved with the verification itself, when during verification unexpected properties are encountered that are not related to errors in the original system, but to errors in the model. This can lead to a continuous improvement of the verification model itself, and requires precise bookkeeping of model versions and properties verified. The quality of this process has decisive influence on the quality of the verification procedure as a whole.

So far, we have looked at experimentation as a way to improve the practical applicability of formal verification. In the next section we want to present yet

another angle that links verification to empirical methods, viz. in the scientific evaluation of formal methods.

4 Verification and the Methodology of Computer Science

From time to time the question concerning the nature of computer science among the sciences is raised. Hartmanis addressed this question in his 1993 Turing Award Lecture, which led to a subsequent discussion published in the ACM Computing Surveys [3,1]. Hartmanis' own conclusion is not very precise: he qualified computer science as a new species among the known sciences for which "a haunting question remains about analogies with the development of physics". Even so, he reports on the coexistence of science and engineering aspects, and remarks:"Somewhat facetiously, but with a grain of truth in it, we can say that computer science is the engineering of mathematics (or mathematical processes)". The authors in [1] take various positions, some defending the experimental science point of view, others emphasising the engineering aspects, and yet others argue for both. The overall impression is that at that time there was no general agreement.

Interestingly enough, already in 1986 Robin Milner gave an account of the experimental nature of a good part of computer science in [5]. He distinguishes between the hard core of computing theory, consisting of recursion and complexity theory, dealing with the characterisation and classification of what is computable, and mathematical theories "in the service of design", i.e. theories that help making and analysing computational artifacts. He proposes that such theories must be evaluated experimentally, by using them in prototype methodologies that are tried out in practice.[1] This position is nicely reminiscent of the point of view taken by Herbert Simon in his *Sciences of the Artificial* [6]. He argues that in general the engineering of artifacts is based on an empirical science of implementation and realisation methods. Experimental design is used to determine the scope of effectiveness of the different methods: under which conditions and circumstances can they be applied successfully, and how to they influence the quality of the resulting product?

In this context (computer aided) verification can be seen as experiments in the sense of Milner and Simon to determine the effectiveness of formal methods. Of course, there are many other experiments that one can think of, dealing with qualities other than correctness. But verification is a useful class of experiments because it can be tool supported and seems to lend itself better for purposes of comparison than, for example, entire system designs.

The role of verification as experimention with formal methods suggests that we should also develop richer evaluation criteria for such experiments than seem to be in current use. Computer aided verification is strongly focussed on the performance (time, memory usage) of the software tools that are used. This is understandable from the existing drive towards faster and better verification

[1] A similar point of view was elaborated by the author in [2]

tools. Nevertheless, it is very important to know to what extent other aspects of verification are (not) supported in different formal frameworks, such as the relative ease of validating a verification model, of obtaining a verification model, of selecting and formalising correctness criteria, etc. These observations imply that it is important to repeat and compare verifications using different formalisms and tools. The results of such repeated experiments should be publishable. This should also hold for failed attempts, provided that interesting lessons can be distilled from such failures.

Publications like [4] suggest that computer science compares badly with other branches of science, in the sense that relatively few papers are published with experimentally validated results. Every real opportunity to validate our methods should therefore exploited, and we should strive for a culture that is comparable to that in the other sciences, viz. that in the long run there is no place for formal methods that have not been validated by serious experimentation.

References

1. Computing surveys symposium on computational complexity and the nature of computer science. *ACM Computing Surveys*, 27(1):5–61, 1995. 23
2. E. Brinksma. What is the method in formal methods? In *Formal description Techniques, IV*, IFIP Transactions **C-2**, pages 33–50. North-Holland, 1992. 23
3. J. Hartmanis. Turing award lecture: On computational complexity and the nature of computer science. *ACM Computing Surveys*, 27(1):7–16, 1995. 23
4. P. Lukowicz, E. A. Heinz, L. Prechelt, and W. F. Tichy. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, January 1995. 24
5. Robin Milner. Is computing an experimental science. Technical Report ECS-LFCS-86-1, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1986. 23
6. H. A. Simon. *Sciences of the Artificial*. MIT Press, 1981. 23
7. W. Turski. Essay on software engineering at the turn of the century. In *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2000. 18

Compositional Performance Analysis Using Probabilistic I/O Automata*

Eugene W. Stark

Department of Computer Science, State University of New York at Stony Brook
Stony Brook, NY 11794 USA
stark@cs.sunysb.edu

In this talk, I give an overview of some recent work, by my colleagues and me at Stony Brook, concerning compositional specification and performance analysis techniques for finite-state reactive systems in which probability and timing play a significant role.

By *reactive systems*, I mean concurrent systems whose components maintain an ongoing interaction with their environments. The complexity of such systems typically resides not in the calculations to be performed by individual system components, but rather in timing and other details of the interactions between components. Examples of reactive systems incude process control software, telecommunication security and e-commerce protocols, and embedded systems. By *compositional specifications* I mean those in which complex systems are described in a hierarchical and modular way as compositions of simpler components. By *compositional performance analysis* I mean performance analysis techniques that exploit the modular structure of a system description and work in a component-by-component fashion to calculate performance measures.

In our work [WSS97,SS98,SP99], we have introduced *Probabilistic I/O Automata* (PIOA) as a model for reactive systems in which probability and timing are of importance. The PIOA model is a probabilistic adaptation of the I/O automata model developed by Nancy Lynch and her students [Lyn96]. A key feature of the I/O automata model is the operation of *composition*, by which a "compatible" collection of I/O automata can be combined into a single, larger automaton. The notion of composition depends in an essential way on a distinction made in the I/O automata model between *input actions*, which are stimuli applied to an automaton by its environment, *output actions*, which are responses made by an automaton to its environment, and *internal actions*, which represent internal steps in which the automaton does not interact with its environment. Output and internal actions are called *locally controlled*, because their occurrence is under the control of the automaton, whereas input actions are under the control of the environment, with the automaton unable to exert any influence over their occurrence.

The PIOA model integrates probability and timing into the I/O automata model, while carrying over in a natural way its essential features of asynchrony and compositionality. To the original I/O automata model, two kinds of probability-related information are added. First, probability distributions are

* Research supported in part by AFOSR Grant F49620-96-1-0087.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 25–28, 2000.

associated with each state q of an automaton, in such a way that for each input action a, there is one probability distribution covering all transitions for action a from state q, and such that there is one additional probability distribution covering all transitions for locally controlled actions from state q. The second type of probabilistic information included in the PIOA model consists of a *rate* associated with each state. The rate is a nonnegative real number, which we interpret as the parameter of an exponentially distributed random variable that describes the amount of time spent by an automaton in a state before it executes its next locally controlled action. Rates enable us to "probabilize" the scheduling of locally controlled transitions taken by a system of component PIOAs, according to the following *race criterion*: upon entering a state, each component PIOA chooses randomly, according to the exponential distribution whose parameter is the rate of that state, a nonnegative real number that represents the amount of time that PIOA will remain in this state before executing the next locally controlled action. The times chosen by each of the component PIOAs are compared, the PIOA that has chosen the smallest time is declared "the winner," and it is allowed to perform the next locally controlled action at the time it has chosen.

A PIOA with an empty set of input actions is called *closed*. Closed PIOAs determine continuous-time Markov chains (CTMCs) [How71], which are widely used in modeling and performance analysis. An important reason for the widespread use of CTMCs is that they can, in fact, be analyzed—algorithms exist to "solve" the CTMC to compute performance measures that predict the behavior of the real system. For example, the *steady-state probabilities* for a CTMC describe the fraction of time spent by the system in each state over the long run, and these probabilities can be obtained by solving a system of linear "balance equations" associated with the CTMC. From the steady-state probabilities, one can compute the *mean recurrence times*, which describe, on the average, how long it will take for a system to return to a state it has just left. The reciprocals of the mean recurrence times can be interpreted as *throughputs*, which are often useful performance parameters that can actually be measured for a real system.

The PIOA model is closely related to stochastic process algebras such as EMPA [BDG98], PEPA [Hil96], and TIPP [HR94], whose stochastic semantics are also described in terms of an underlying CTMC. Significant differences between the PIOA model and these other stochastic process algebras are: (1) the PIOA model does not attempt to treat any form of non-probabilistic internal choice, and (2) in the PIOA model, a syntactic distinction is drawn between output ("active") actions and input ("passive") actions, and at most one component can be active in any synchronized action. These restrictions enable the PIOA model to avoid semantic problems associated with assigning probabilities to synchronized actions.

Traditional techniques [Ste94] for performance analysis of a CTMC system model typically proceed by first compiling the system description into a matrix representation of a CTMC. An iterative numerical method, such as Gauss-Seidel iteration or successive overrelaxation, is then used to solve the system to obtain,

for example, the steady-state probabilities. Although the matrices arising in practice from CTMC descriptions are generally sparse, and require storage space which is linear in the number of states of the CTMC, if no steps are taken to reduce the size of the CTMC representation, the amount of memory needed to store it limits the maximum size of models that can be solved. Typical maximum sizes of CTMCs that can be treated by standard methods are somewhat in excess of 10^6 states. This sounds like a lot of states, until one realizes that it only takes six components, with each component having ten states, to describe a system with this many global states. This severely limits the applicability of standard techniques to even relatively simple real-world systems.

The PIOA-based techniques we have developed can be used to give a description of a CTMC as the composition of a collection of PIOA components, and then to exploit the compositional structure of the system description in the calculation of performance measures, thereby avoiding the construction of the full CTMC state space. Our techniques are applied by starting with a representation of a particular performance measure to be calculated, and then treating the system one component at a time. As each component is treated, state-space reduction is performed before proceeding to the next component. This reduction step eliminates information that is not relevant to the computation of the performance measure of interest. The overall idea is similar to the compositional aggregation technique of Hillston [Hil95], except that we use a very different state-space reduction algorithm and we are able to use information about the performance measure to enhance the reduction.

Our techniques most naturally apply to the calculation of transient performance measures that can be expressed as expectations of a certain kind of function over the probability space of system executions. Some examples of performance measures that can be expressed in this way are: (1) the probability of a total failure of all components, in a system in which components fail and are repaired after some time; (2) the expected time until a total failure in such a system, assuming that such a failure will occur with probability one. We have also shown that our techniques can be applied to compute steady-state performance measures, by viewing them as limits of transient measures.

Our methods for compositional analysis of systems of PIOAs are based on a kind of abstract semantics for PIOAs that retains just the information required for performance analysis. The main notions in the theory are: *rated traces*, which are alternating sequences of rates and actions that form a kind of abstraction of PIOA executions; *observables*, which are measurable functions from rated traces to real numbers, and whose expectations correspond to performance measures; and *PIOA behaviors*, which are functions from observables to observables that capture the way in which system performance is modified by the incorporation of an additional PIOA as a system component. A *compositionality theorem* states that PIOA behaviors compose: the behavior of a composite system can be obtained by composing (as functions) the behaviors of the individual system components.

Our algorithms for compositional performance analysis work for *representable observables*, which are those that have a *linear representation* as a certain kind of automaton with states in a finite-dimensional vector space. Linear representations of observables are a generalization of the classical notion of linear representations for formal power series [BR84]. We have shown [SS98] that algorithms exist for: (1) calculating the result of applying the behavior of a finite-state PIOA to a representable observable; (2) finding, given a linear representation of an observable, a minimum-dimension linear representation for that same observable; and (3) calculating the expectation of an observable to obtain a performance measure. We have implemented our algorithms in the very high-level functional programming language Standard ML. A description of the implementation and the results of applying it to some examples can be found in [SP99].

References

BDG98. M. Bernardo, L. Donatiello, and R. Gorrieri. A formal approach to the integration of performance aspects in the modeling and analysis of concurrent systems. *Information and Computation*, 144:83–154, August 1998. 26

BR84. J. Berstel and C. Reutenauer. *Rational Series and Their Languages*, volume 12 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984. 28

Hil95. J. Hillston. Compositional Markovian modelling using a process algebra. In W. Stewart, editor, *Proceedings of the Second International Workshop on Numerical Solution of Markov Chains: Computations with Markov Chains*, Raleigh, North Carolina, January 1995. Kluwer Academic Press. 27

Hil96. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996. 26

How71. R. A. Howard. *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes*, volume 2 of *Series in Decision and Control*. John Wiley & Sons, 1971. 26

HR94. H. Hermanns and M. Rettelbach. Syntax, semantics, equivalences, and axioms for MTIPP. In U. Herzog and M. Rettlebach, editors, *Proc. of 2nd Workshop on Process Algebras and Performance Modelling*, Erlangen-Regensberg, Germany, July 1994. Universität Erlangen. 26

Lyn96. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. 25

SP99. E. Stark and G. Pemmasani. Implementation of a compositional performance analysis algorithm for probabilistic I/O automata. In *Proceedings of 1999 Workshop on Process Algebra and Performance Modeling (PAPM99)*. Prensas Universitarias de Zaragoza, September 1999. 25, 28

SS98. E. W. Stark and S. Smolka. Compositional analysis of expected delays in networks of probabilistic I/O automata. In *Proc. 13th Annual Symposium on Logic in Computer Science*, pages 466–477, Indianapolis, IN, June 1998. IEEE Computer Society Press. 25, 28

Ste94. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994. 26

WSS97. S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1-2):1–38, 1997. 25

Formal Models for Communication-Based Design

Alberto Sangiovanni-Vincentelli[1], Marco Sgroi[1], and Luciano Lavagno[2]

[1] University of California at Berkeley
EECS Department, Berkeley, CA 94720
[2] Cadence Berkeley Labs
2001 Addison Street, Berkeley, CA 94704-1144

Abstract. Concurrency is an essential element of abstract models for embedded systems. Correctness and efficiency of the design depend critically on the way concurrency is formalized and implemented. Concurrency is about communicating processes. We introduce an abstract formal way of representing communication among processes and we show how to refine this representation towards implementation. To this end, we present a formal model, Abstract Co-design Finite State Machines (ACFSM), and its refinement, Extended Co-design Finite State Machines (ECFSM), developed to capture abstract behavior of concurrent processes and derived from a model (Co-design Finite State Machine (CFSM)) we have used in POLIS, a system for the design and verification of embedded systems. The design of communication protocols is presented as an example of the use of these formal models.

1 Introduction

By the year 2002, it is estimated that more information appliances will be sold to consumers than PCs (see *Business Week*, March 1999). This new market includes small, mobile, and ergonomic devices that provide information, entertainment, and communications capabilities to consumer electronics, industrial automation, retail automation, and medical markets. These devices require complex electronic design and system integration, delivered in the short time frames of consumer electronics. The system design challenge of at least the next decade is the dramatic expansion of this spectrum of diversity and the shorter and shorter time-to-market window. Given the complexity and the constraints imposed upon design time and cost, the challenge faced by the electronics industry is insurmountable unless a new design paradigm is developed and deployed that focuses on:

- design re-use at all levels of abstraction;
- "correct-by-construction" transformations.

An essential component of a new system design paradigm is the *orthogonalization of concerns*, i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions. The pillars of the design methodology that we have proposed over the years are:

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 29–47, 2000.

– the separation between function (what the system is supposed to do) and architecture (how it does it);
– the separation between computation and communication.

1.1 Function-Architecture Co-design

The mapping of function to architecture is an essential step from conception to implementation. In the recent past, there has been a significant attention in the research and industrial community to the topic of Hardware-Software Co-design. The problem to be solved here is coordinating the design of the parts of the system to be implemented as software and the parts to be implemented as hardware, avoiding the HW/SW integration problem that has marred the electronics system industry for so long. We actually believe that worrying about hardware-software boundaries without considering higher levels of abstraction is the wrong approach. HW/SW design and verification happens after some essential decisions have been already made, thus making the verification and the synthesis problem so hard. SW is really the form that a given piece of functionality takes if it is "mapped" onto a programmable microprocessor or DSP. The origin of HW and SW is in behavior that the system must implement. The choice of an "architecture", i.e. of a collection of components that can be either software programmable, re-configurable or customized, is the other important step in design.

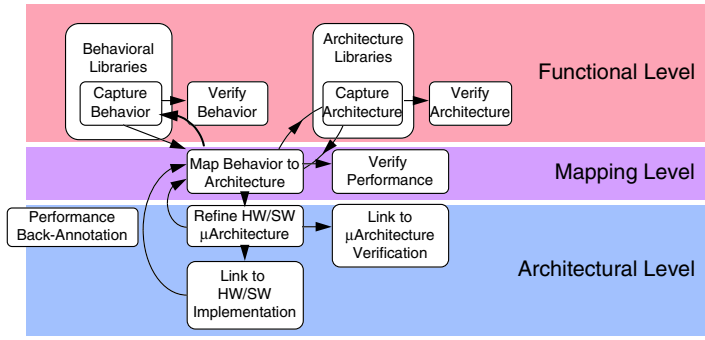


Fig. 1. Proposed design strategy

1.2 Communication-Based Design

The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of concurrent systems is essential. In any large-scale embedded systems design methodology, concurrency must be considered as a first class citizen at all levels of abstraction and in both hardware and software.

Concurrency implies communication among components of the design. Communication is too often intertwined with the behavior of the components of the

design so that it is very difficult to separate out the two domains. Separating communication and behavior is essential to dominate system design complexity. In particular, if in a design component behaviors and communications are intertwined, it is very difficult to re-use components since their behavior is tightly dependent on the communication mechanisms with other components of the original design. In addition, communication can be described at various levels of abstraction, thus exposing the potential of implementing communication behavior in many different forms according to the available resources. Today this freedom is often not exploited.

1.3 Formal Models

We have promoted the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology [1]. Further, the concept itself of synthesis can be applied only if the precise mathematical meaning of a description of the design is applied. It is then important to start the design process from a high-level abstraction that can be implemented in a wide variety of ways. The implementation process is a sequence of steps that remove freedom and choice from the formal model. In other words, the abstract representation of the design should "contain" all the correct implementations in the sense that the behavior of the implementation should be consistent with the abstract model behavior. Whenever a behavior is not specified in the abstract model, the implicit assumption is that such behavior is a "don't-care" in the implementation space. In other words, the abstract model is a source of non-deterministic behavior, and the implementation process progresses towards a deterministic system. It is important to underline that way too often system design starts with a system specification that is burdened by unnecessary references to implementations resulting in over-determined representations with respect to designer intent that obviously yield under-optimized designs.

In the domain of formal model of system behavior, it is common to find the term "Model of Computation" (MOC), an informal concept that has its roots in language theory. This term refers more appropriately to mathematical models that specify the semantics of computation and of concurrency. In fact, concurrency models are the most important differentiating factors among models of computation. Ed Lee [2] has very well stressed the importance of allowing one to express designs making use of all models of computation, or at least of the principal ones, thus yielding a so-called heterogeneous environment for system design. In his approach to *simulation and verification*, assembling a system description out of modules represented in different models of computation reduces to the problem of arbitrating communication among the different models. However, the concept of communication among different models of computation still needs to be carefully explored and understood from a *synthesis and refinement* viewpoint.

This difficulty has actually motivated our approach to communication-based design where communication takes the driver seat in system design [7]. In this

approach, communication can be specified somewhat independently of the modules that compose the design. In fact, two approaches can be applied here. In the first case, we are interested in communication mechanisms that "work" in any environment, i.e., independently of the formal models and specifications of the behavior of the components. This is a very appealing approach if we look at the ease of component assembly. It is however rather obvious that we may end up with an implementation that is overly wasteful, especially for embedded systems where production cost is very important. A more optimal (but less modular) approach is to specify the communication behavior, and then to refine *jointly* one side of the communication protocol and the behavior that uses it, in order to exploit knowledge of both to improve the efficiency of the implementation. Here, a synthesis approach is most appealing since it reduces the risk of making mistakes and it may use powerful optimization techniques to reduce design cost and time.

Communication and time representation in a Model Of Computation are strictly intertwined. In fact, in a synchronous system, communication can take place only at precise "instants of time" thus reducing the risk of unpredictable behavior. Synchronous systems are notoriously more expensive to implement and often less performing thus opening the door to asynchronous implementations. In this latter case, that is often the choice for large system design, particular care has to be exercised to avoid undesired and unexpected behaviors. The balance between synchronous and asynchronous implementations is possibly the most challenging aspect of system design. We argue that globally asynchronous locally synchronous (GALS) communication mechanisms are probably a good compromise in the implementation space [1]. The research of our group in the last few years has addressed the above problems and allowed us to define a full design methodology and a design framework, called Polis [1], for embedded systems. The methodology that we have proposed is based on the use of a formal and implementation-independent MOC, called CFSMs (Co-design Finite State Machines). CFSMs are Finite State Machines extended with arithmetic data paths, and communicating asynchronously over signals. Signals carry events, which are buffered at the receiving end, and are inherently uni-directional and potentially lossy (in case the receiver is not fast enough to keep up with the sender's speed). The CFSMs model is Globally Asynchronous Locally Synchronous (GALS), since every CFSM locally behaves synchronously following the FSMs semantics while the interaction among CFSMs is asynchronous from a system perspective.

However, the view of communication in CFSMs is still at a level of abstraction that is too low. We would like to be able to specify abstract communication patterns with high-level constraints that are not implying yet a particular model of communication. For example, it is our opinion that an essential aspect of communication is the guarantee of reception of all the information that has been sent. We argue that there must exist a level of abstraction that is high enough to require that communication takes place without loss. The synchronous-asynchronous mechanism, the protocols used and so on, are just implementation choices that either guarantee no loss or that have a good chance

of ensuring that no data is lost where it matters (but that need extensive verification to make sure that this is indeed the case). For example, Kahn process networks [4] guarantee no loss at the highest level of abstraction by assuming an ideal buffering scheme that has unbounded buffer size. Clearly the unbounded buffer size is a "non-implementable" way of guaranteeing no loss. When moving towards implementable designs, this assumption has to be removed. A buffer can be provided to store temporarily data that are exchanged among processes but it must be of finite size. The choice of the size of the buffer is crucial. Unfortunately deciding whether a finite buffer implementation exists that guarantees no loss is not theoretically feasible in the general case, but there are cases for which an optimal buffer size can be found [2]. In other cases, one has to hope for the best for buffer overwrite not to occur or has to provide additional mechanisms that, when composed with the finite buffer implementation, still guarantee that no loss takes place (for example, a request/acknowledge protocol). Note that in this case the refinement process is quite complex and involves the use of composite processes. Today, there is little that is known about a general approach to communication design that has some of the feature that we have exposed.

An essential step to develop communication-based design is the understanding of "communication" semantics. We believe that communication in formal models has not been treated at the correct level of abstraction.

1.4 Outline of the Paper

In Section 2, we present a model for communication semantics and use it to describe the FIFO communication mechanism. This mechanism is used in the two models introduced in Section 3 obtained by "abstracting" Co-design Finite State Machines (CFSMs) to deal with the problems posed by communication-based design. Finally, in Section 4 we show how to use these models to design an application example: a wireless communication protocol.

2 Communication

Large distributed systems are composed of a set of concurrent and interacting components [1].

A prerequisite for the interaction between distinct components is the existence of a *connection* between an output port of one component, called the *sender*, and an input port of another component, called the *receiver*. A connection can be modeled as a process whose function is the identity between input and output signals. With respect to the interaction between sender and receiver, a connection imposes the equality of the input signal of the receiver with the output signal of the sender (Figure 2).

[1] Following the Tagged Signal Model formalism [3], system components are modeled as functional processes, whose set of behaviors is defined by a mapping from a set of input signals I to a set of output signals O, $F : I \Rightarrow O$. Unless a definition is explicitly given, terms like process, signal, behavior are used in this paper as in [3].

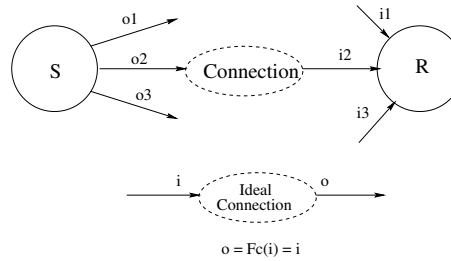


Fig. 2. Connection Process

Consider two components, the *sender* modeled by process S ($F_s : I_s \Rightarrow O_s$) and the *receiver* by process R ($F_r : I_r \Rightarrow O_r$). Connecting S and R as shown in Figure 3 implies that only a signal that is an output of S may be an input of R. As a result, the input space of R is *restricted* to the intersection $O_s \cap I_r$ of its domain I_r with the output space of S (Figure 3). When the domain of R includes a signal $i_r \notin O_s$, the connection results in a restriction of the set of possible behaviors of R that can be optimized by removing the behaviors that correspond to the input signals $i_r \in \overline{O_s} \cap I_r$.

2.1 Adapters

If the set $O_s \cap \overline{I_r}$ is not empty, we say that the behaviors of S and R are not adapted, since the behavior of R is not defined for inputs $o_s^* \in O_s \cap \overline{I_r}$. This mismatch [2] can be solved in one of the following ways:

1. R discards inputs o_s^* and treats them as errors,
2. outputs o_s^* of S and the behaviors originating them are removed from S,
3. signals o_s^* are mapped into signals that can be accepted by R.

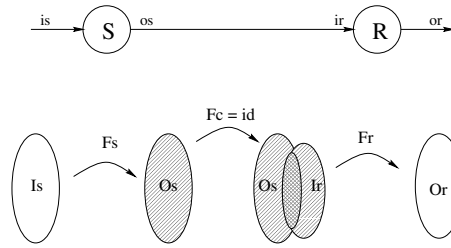


Fig. 3. Behavior mismatch

[2] An example is when the sender is an analog system and the receiver is digital: an A/D converter is needed to allow the receiver to understand the messages of the sender.

In 1) the set of behaviors of R is extended to R' to include the error handling of the undesired input signals that may be received. In 2) the behavior of the sender is optimized and restricted to S' to exclude the production of output signals incompatible with R. In 3) an interface (represented in Figure 4 as a process with function F_{beh_int}) is used to map the signal emitted by S into a signal that belongs to the domain of R. Such an interface [8] can be usually split into two processes ($F_{beh\_int} = F_{beh\_sint} \circ F_{beh\_rint}$)), which encapsulate S and R ($F_{s'} = F_s \circ F_{beh\_sint}$ and $F_{r'} = F_{beh\_rint} \circ F_r$) and permit communication between the modified behaviors S' and R' over a connection. We call this type of interface *Behavior Adapter (BA)* (Figure 4).

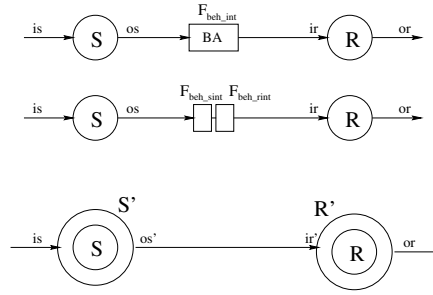


Fig. 4. Behavior Adapter

2.2 Channels

Connections are implemented using physical channels (Figure 5) [3] , whose function ($F_c : I_c \Rightarrow O_c$) in general differs from the identity, e.g. due to noise or interference. As a result, even if the behaviors S' and R' were perfectly adapted, the received signal $F_c(i_c) = F_c(o_s)$ might be out of the domain of R' or trigger an incorrect behavior of R. Therefore, for a safe and correct interaction among system components it is key to select a channel whose behavior approximates that of the ideal connection.

Quality of Service (QoS) requirements [4] partition the set of behaviors F into two classes, the class of those that satisfy them and the class of those that do not satisfy them. Let us introduce a relation $\sim \subseteq F \times F$, such that

$$\sim = \{(f, f') | f \in F, f' \in F, \text{both f and f' satisfy the quality requirements}\}.$$

Two processes f and f' such that $f \sim f'$ are said to be QoS-equivalent. Given a connection and a set of requirements on the quality of the received signal, the set of the behaviors $F_s \circ F_c \circ F_r$, where F_c is the channel function,

[3] A connection establishes a relation between signals. A channel is a set of physical objects that implement a connection.

[4] Quality of Service requirements include maximum delay, minimum throughput, maximum number of errors.

is partitioned into two classes: the class of *valid* channels and the class of the *invalid* ones. The former includes all the channels that guarantee satisfaction of the requirements on the quality of the received signal [5]. Since the ideal connection (identity function *id*) by definition satisfies the quality requirements, the set of valid channels can be defined as:

$$ValidChannels = \{F_c : F_s \circ F_c \circ F_r \sim F_s \circ F_r\}$$

The first step in designing a valid channel is to select a physical channel whose function is F_c. If the channel is invalid due to its physical limitations, it is necessary to introduce an interface between the behaviors and the channel that matches the undesired effects. We call this type of interface *Channel Adapter (CA)* [6]. A channel adapter interface is usually symmetric to the channel and is defined by two functions, (F_{cs_int}) implementing the sender-channel and (F_{cr_int}) the channel-receiver interfaces (Figure 5). If $F_s \circ F_{cs\_int} \circ F_c \circ F_{cr\_int} \circ F_r \sim F_s \circ F_r$ the interface successfully adapts the channel F_c, otherwise it is necessary to iterate the adaptation process and look for two other functions $F_{cs'\_int}$ and $F_{cr'\_int}$ such that $F_s \circ F_{cs\_int} \circ F_{cs'\_int} \circ F_c \circ F_{cr'\_int} \circ F_{cr\_int} \circ F_r \sim F_s \circ F_r$ [7]. Note also that if channel adapters introduce some mismatch between the range of $F_{cs\_int} \circ F_c$ and the domain of F_{cs_int} a behavior adapter is needed.

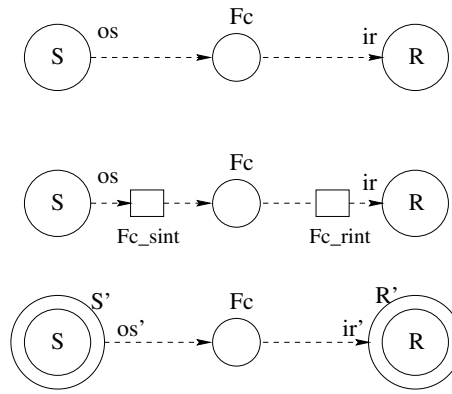


Fig. 5. Channel Adapter

[5] No losses is one of the possible QoS requirements for a communication. For such requirement valid channels are called lossless, invalid channels lossy.

[6] Example of Channel Adapters functions are: error correction, flow control, medium access control.

[7] The channel adapter interface often consist of several layers, each adapting the channel at a different level of abstraction.

2.3 Communication and Protocols

We define communication as a composition of processes that is QoS equivalent to the identity process. A protocol is the sequence of behavior and channel adapters that implement a communication.

Figure 6 describes the flow we propose for designing protocols. Given two components S and R, they are first connected and their behaviors are compared. If there is a need for a behavior adapter BA, this is introduced. The next step is the selection of channel CH. If there is no valid channel available, a channel adapter composed of the two interfaces CRA (channel-receiver adapter) and SCA (sender-channel adapter) is introduced to overcome the limitations of the invalid channel selected. If the behavior of the sender composed with the channel and the behavior of the receiver are not adapted, another behavior adapter is needed. This procedure is iterated until the behaviors no longer need to be adapted and a valid channel is defined.

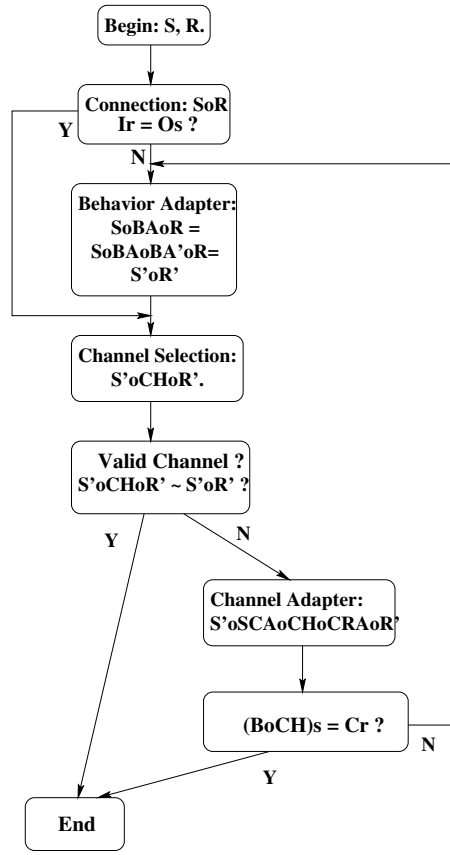


Fig. 6. Design Flow

2.4 Communication over FIFO Channels

A FIFO channel can be seen as an adapter between behaviors that operate at a different rate. If the sender produces outputs at a faster rate than the receiver, a FIFO allows to handle their rate difference and prevent from losses.

An unbounded queue is an ideal adapter, since it allows to accumulate an infinite number of tokens and therefore match any rate difference. However, communication has to be implemented using channels that have finite queues. Unfortunately, bounded FIFO channels do not prevent from overflow, i.e. losses, when the FIFO is full. For this reason, a bounded FIFO channel may not be a valid channel, especially for systems where the rate difference between sender and receiver is large. In this case, it is necessary to introduce a channel adapter interface that may take the form either of a scheduling policy or an explicit Request/Acknowledgment protocol that blocks the sender when the FIFO is full. In particular the Req/Ack protocol restricts the behavior of S to S' excluding all the behaviors, considered illegal, where the number of consecutive output events exceeds the capacity of the queue (Figure 7).

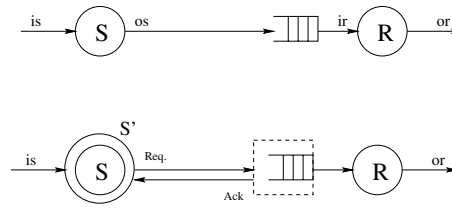


Fig. 7. Fifo Channel

3 Abstract and Extended Codesign Finite State Machines

A network of Co-design Finite State Machines was introduced in [1] to model formally the behavior of embedded systems. It consists of a network of FSMs that communicate asynchronously by means of events (that at the abstract level only denote partial ordering, not time) over signals with FIFO semantics. A network of CFSMs is a Globally Asynchronous Locally Synchronous (GALS) model:

- the local behavior is synchronous (*from its own perspective*, like the "atomic firing" of Dataflow actors), because each CFSM executes a transition by producing an output reaction based on a snap-shot set of inputs in zero time,
- the global behavior is asynchronous (*as seen by the rest of the system*) since each CFSM detects inputs, executes a transition, and emits outputs in an unbounded but finite amount of time.

The asynchronous communication among CFSMs over a FIFO channel, where the FIFO had a length of one, supports a (possibly very non-deterministic) specification where the execution delay of each CFSM is unknown a priori and, therefore, is not biased towards a specific hardware/software implementation.

The FIFO channel of length one was appropriate for some control-dominated application but it lacked flexibility to model other applications especially in the communication and consumer electronics domains. In those cases, we noted that there was value in decoupling the behavior of each CFSM from the communication with other CFSMs and in allowing an ideal communication mechanism where the FIFO channel is unbounded. This is the origin of the network of Abstract Co-design Finite State Machines (ACFSM) ACFSM. The communication can then be designed by refinement independently from the functional specification of the ACFSMs to yield a network of Extended Co-design Finite State Machines ECFSM with finite (and hence implementable) FIFO channels[8].

3.1 Single A(E)CFSM Behavior

A single ACFSM (ECFSM)[9] describes a finite state control operating on a data flow. It is an extended FSM, where the extensions add support for data handling and asynchronous communication. An ACFSM transition can be executed when a pre-condition on the number of present input events and a boolean expression over the values of those input events is satisfied. During a transition execution, an ACFSM first *atomically* detects and consumes some of the input events, then performs a computation by emitting output events with the value determined by expressions over detected input events. A key feature of ACFSMs is that transitions in general consume multiple events from the same input signal, and produce multiple events to the same output signal (*multi-rate transitions*). We formally define an ACFSM as follows:

Definition 1. *An ACFSM is a triple* $A = (I, O, T)$*:*

- $I = \{I_1, I_2, \ldots I_N\}$ *is a finite set of inputs. Let* i_i^j *indicate the event that at a certain instant occupies the j-th position in the FIFO at input* I_i*.*
- $O = \{O_1, O_2, \ldots O_M\}$ *is a finite set of outputs. Let* o_i^j *indicate the j-th event emitted by a transition on output* O_i*.*
- $T \subseteq \{(IR, IB, CR, OR, OB)\}$ *is the transition relation, where:*
  - *IR is the input enabling rate,*
    $IR = \{(I_1, ir_1), (I_2, ir_2), \ldots, (I_N, ir_N) \mid$
    $1 \leq n \leq N, I_n \in I, ir_n \in \mathbb{N}\}$
    i.e., ir_n *is the number of input events from each input* I_n *that are required to trigger the transition.*

[8] Note that a network of CFSMs is a particular case of network of ECFSM.

[9] From now on, in this subsection we refer to both ACFSM and ECFSM behavior. Hence ACFSM can be substituted with ECFSM to obtain the behavior of a single ECFSM.

- *IB is a boolean-valued expression over the values of the events* $\{i_i^j\}, 1 \leq i \leq N, 1 \leq j \leq ir_i$ *that enable the transition.*
- *CR is the input consumption rate,* $CR = \{(I_1, cr_1), (I_2, cr_2), \ldots, (I_N, cr_N) \mid 1 \leq n \leq N, I_n \in I, cr_n \in \mathbb{N}, cr_n \leq ir_n \vee cr_n = I_n^{ALL}\}$ *i.e.,* cr_n *is the number of input events consumed from each input*[10].
- *OR is the output production rate,* $OR = \{(O_1, or_1), (O_2, or_2), \ldots, (O_M, or_M) \mid 1 \leq m \leq M, O_m \in O, or_m \in \mathbb{N}\}$ *i.e.,* or_n *is the number of output events produced on each output* O_n *during a transition execution.*
- *OB is a set of vectors of expressions that determines the values of the output events, one vector per output with* $or_n > 0$ *and one element per emitted event.* $\{o_i^j\}, 1 \leq i \leq N, 1 \leq j \leq or_i$

Note that signals that are at the same time input and output, and for which a single event is produced and consumed at each transition act as *implicit* state variables of the ACFSM.

If several transitions can be executed in a given configuration (events and values) of the input signals, the ACFSM is non-deterministic and can execute any one of the matching transitions.

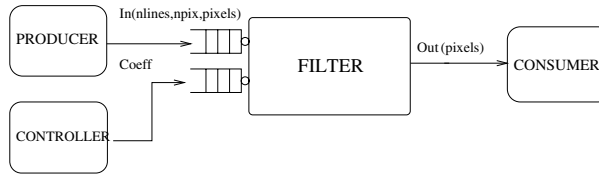
ACFSMs differ from Dataflow networks in that there is no blocking read requirement, i.e. ACFSMs transitions, unlike firings in DF networks, can be conditioned to the *absence* of an event over a signal. Hence, ACFSMs are also not continuous in Kahn's sense [4] (the arrival of two events in different orders may change the behavior). Another difference from DF models is that ACFSMs can "flush" an input signal, in order to model exception handling and reaction to disruptive events (e.g., errors and re-initializations). A DF actor cannot do so, since when the input signal has been emptied, the actor is blocked waiting on the signal, rather than proceeding to execute the recovery action.

A difference from most FSM-based models (e.g., SDL) is the possibility to use multi-rate transitions to represent production and consumption of events over the same signal at different rates (e.g. images produced line-by-line and consumed pixel-by-pixel).

As an example, consider the filter shown in Figure 8. It filters a sequence of frames, by multiplying all the pixels of a frame by a coefficient. At the beginning of each iteration it receives the number of lines per frame and pixels per line

[10] The number of events that is consumed should be not greater than the number of events that enabled the transition. It is also possible to specify, by saying that $cr_n = I_n^{ALL}$, that a transition *resets* a given input, i.e., it consumes all the events in the corresponding signal (that must be at least as many as those enabling the transition).

from the input signal *in* and the initial filtering coefficient (used to multiply the pixels) from the input signal *coef*. Then, it receives a frame, i.e. a sequence of lines of pixels, from *in*, possibly interleaved with new coefficient values (if it must be updated) from *coef*. The filtered frame is produced on the output signal *out*. The primitives *read(in,n)* and *write(out,n)* consume and produce n events from signal *in* and *out*, respectively, and *present(coef,n)* returns true if n events are available on signal *coef*.




```
module filter;
input byte in, coef;
output byte out;
int nlines, npix, line, pix;
byte k;
int buffer[];
forever {
  (nlines, npix) = read (in, 2);
  k = read (coef, 1);
  for (line = 1; line <= nlines; line++) {
    if (present(coef, 1)) k = read (coef, 1);
    buffer = read (in, npix);
    for (pix = 1; pix <= npix; pix++)
      buffer[pix] = buffer[pix] * k;
    write (out, buffer, npix);
  }
}
```

Fig. 8. Filter example

Let *prod(v,s,n)* denote the multiplication of a vector v of n values by a scalar s, and "$\Leftarrow$" a shorthand to represent writing to state feedback signals, which are also omitted for simplicity from IR and OR, where they always have rate 1. The filter can be modeled by an ACFSM as follows:

- $IR = CR = \{(\text{in}, 2), (\text{coef}, 1)\}$, $IB = (\text{state} = 1)$,
  $OR = \{\}$, $OB = \{(\text{nlines}, \text{npix}) \Leftarrow \text{read}(\text{in}, 2),$
  $\text{line} \Leftarrow 1, k \Leftarrow \text{read}(\text{coef}, 1), \text{state} \Leftarrow 1\}$
- $IR = CR = \{(\text{in}, \text{npix})\}$,
  $IB = (\text{state} = 2 \wedge \text{line} < \text{nlines})$,
  $OR = \{(\text{out}, \text{npix})\}$,

$OB = \{\text{write}(\text{out}, \text{prod}(\text{read}(\text{in}, \text{npix}), k, \text{npix}), \text{npix}),$
$\text{state} \Leftarrow 2, \text{line} \Leftarrow \text{line} + 1\}$
- $IR = CR = \{(\text{in}, \text{npix})\}$,
  $IB = (\text{state} = 2 \wedge \text{line} = \text{nlines})$, $OR = \{(\text{out}, \text{npix})\}$,
  $OB = \{\text{write}(\text{out}, \text{prod}(\text{read}(\text{in}, \text{npix}), k, \text{npix}), \text{npix}),$
  $\text{state} \Leftarrow 1\}$
- $IR = CR = \{(\text{coef}, 1)\}$, $IB = (\text{state} = 2)$,
  $OR = \{\}$, $OB = \{k \Leftarrow \text{read}(\text{coef}, 1), \text{state} \Leftarrow 2\}$

3.2 Network of ACFSMs and ECFSMs

An ACFSMs (ECFSM) *network* is a set of ACFSMs (ECFSMs) and signals. The behavior of the network depends on both the individual behavior of each ACFSM (ECFSM), and that of the global system. In the mathematical model, the system is composed of ACFSMs (ECFSMs) and a scheduling mechanism coordinating them. The scheduler operates by continually deciding which ACFSMs (ECFSMs) can be run, and calling them to be executed. Each ACFSM (ECFSM) is either *idle* (waiting for input events), or *ready* (waiting to be run by the scheduler), or *executing* a single transition from its transition relation.

The topology of the network, as for dataflow networks, simply specifies a partial order on the execution of the ACFSMs (ECFSMs). Initially the time required by an ACFSM (ECFSM) to perform a state transition is not specified, hence each ACFSM (ECFSM) captures *all its possible hardware* (with or without resource sharing constraints) *and software* (generally with CPU sharing constraints) *implementations*. The ACFSM model is fully implementation-independent but, as a consequence, it is highly non-deterministic.

3.3 Refining ACFSMs

The non-determinism present in ACFSMs is resolved only after an *architectural mapping* (implementation) is chosen. An architectural mapping in this context means:

- a set of architectural and communication resources (CPUs, ASICs, busses, ...),
- a mapping from ACFSMs to architectural resources and from signals to communication resources,
- a scheduling policy for shared resources.

Once an architectural mapping is selected for a network of ACFSMs, the computation delay for each ACFSM transition can be estimated with a variety of methods, depending on how the trade-off between speed and accuracy is solved [1].

Annotating the ACFSM transitions with such delay estimates defines a global order of execution of the ACFSM network that has now been "refined" into a Discrete Event semantics. At this level the designer can verify, by using a Discrete Event simulator [1], if the mapped ACFSM network satisfies not only functional but also performance and cost requirements.

Communication Refinement Abstract CFSMs communicate asynchronously by means of events, transmitting and receiving data over unbounded communication channels with FIFO semantics. This abstract specification defines for each channel only a partial order between the emission and the reception of events and therefore must be refined to be implemented with a finite amount of resources. To implement an abstract communication it is necessary to design a protocol that satisfies the functional and performance requirements of the communication, and can be implemented at a minimum cost in terms of power, area, delay and throughput.

The amount of data that should be *correctly* received is one requirement. A communication mechanism is called *lossless*, if no data (in the form of events) is lost over the communication channel during any system execution, and *lossy* otherwise. The specification requirements dictate if the communication must be lossless or lossy and, if lossy, how much loss is acceptable. Data can be lost either because of the poor quality of the physical channel, e.g. due to noise or interference, or because of the limited amount of resources in the implementation, e.g. the receiver is slow and does not have enough memory to store the incoming data. In the first case the problem is usually overcome by the definition of a robust protocol that (probabilistically) guarantees correctness of received data by means of re-transmissions or coding techniques for error correction. In the second case the solution, as discussed below, is to use a sufficient amount of resources or an appropriate protocol to meet the requirements. The throughput and the latency in the arrival of the data to the destination are key requirements especially in the design of protocols for real-time applications, e.g. real-time video or audio, that require that incoming video frames and audio samples are received and processed at regular intervals.

Communication protocols in our approach are refined towards implementation through several levels of abstraction, by applying a sequence of refinement steps, such that each step preserves the original behavior and constraints are propagated in a *top-down* fashion. At the same time the use of architectural units, e.g. the physical channel, and functional libraries, e.g. communication primitives, captures also the *bottom-up* aspect of the design process. Therefore, we can say that the overall methodology we propose is really a mix of top-down and bottom-up.

Extended Co-design Finite State Machines At the Abstract CFSMs level, the specification includes the topology of the network and the functional behavior of each module, while the protocols that implement the communication requirements are still undefined. To optimally design a protocol for each communication channel it is necessary to use a model, or set of models, that allow to capture different algorithmic solutions and evaluate their implementation costs. For this reason we introduce the Extended Co-design Finite State Machines (ECFSMs) model that "implements" the ACFSMs model. ECFSMs are obtained from the ACFSMs simply by refining infinite-size queues to implementable finite-size queues. The event-based communication semantics, as well as the rules for the execution of ECFSMs transitions, are the same as in the ACFSMs model.

ECFSMs have input queues of finite size, and write operations can be either blocking or non-blocking (on a channel-by-channel basis). In the latter case, every time a queue is full and new data arrives over the channel, the data previously stored is lost (overwritten). To avoid this scenario it is often sufficient to use a longer queue. Sometimes, instead, e.g. when the average production rate of the sender is greater than the average consumption rate of the receiver, there exists no finite-queue solution guaranteeing no loss. The problem of checking if there exists a lossless implementation with bounded queues, that has been solved using static or quasi-static scheduling algorithms for Synchronous Data Flow networks [5] and Free-Choice Petri Nets [6], is undecidable in general for the ACFSMs model. Therefore, to implement lossless communication in a given ECFSM network, it is necessary to define an additional mechanism that, when a queue is full, blocks the sender until the queue has again enough space for new incoming data. This can be achieved by means of either a handshake protocol, consisting of explicit events carrying the sender request for an emission and the receiver acknowledgment that new data can be accommodated in the queue, or scheduling constraints that ensure that the sender is scheduled for execution only when the queue at the receiver has enough space. Depending on the requirements on data losses, ACFSM queues are refined into either *lossy* ones, that are eventually overwritten when they are full, or *lossless* ones, where overwriting is prevented by a blocking write protocol or scheduling constraints.

The actual size of the queues should be determined by evaluating the cost of different implementations that satisfy the communication requirements. For example, large-size queues mean high throughput due to the higher level of pipelining that can be achieved by sender and receiver, but are also expensive in terms of area. Small-size queues reduce the area, but decrease the throughput, as the sender is blocked more often, and increase power consumption (more frequent request and acknowledgment messages).

4 Example: A Wireless Protocol

Intercom [9] is a single-cell wireless network supporting voice communication among a number of mobile terminals. The network operation is coordinated by a unit, called base station, that handles user service requests (e.g. request to establish a connection), and solves the shared wireless medium access problem using a TDMA (Time Division Multiple Access) policy and assigning the slots to communicating users.

Figure 9 describes the Intercom protocol stack, that is composed of the following layers. The *User Interface Layer* interacts with the users and forwards to lower layers user service requests and (logarithmically quantized) voice samples. The *Transport Layer* takes care of message retransmission until acknowledgment reception. The *MAC Layer* implements the TDMA scheme, keeping within internal tables the information on which action (transmit, receive or standby) the terminal should take at each slot. The *Error Control Layer* applies the Cyclic Redundancy Check algorithm to the incoming and outgoing streams of data and,

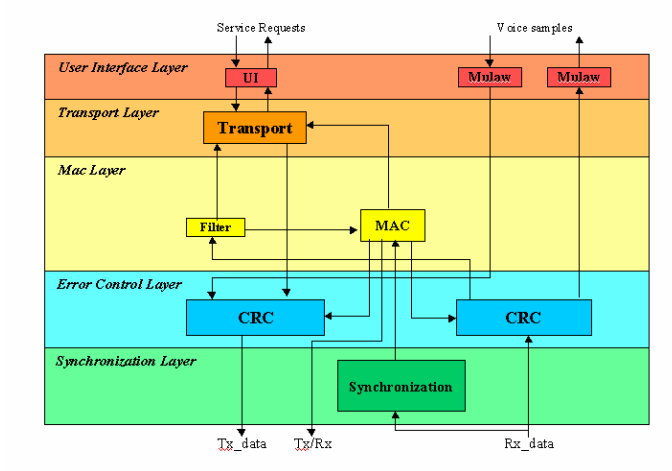


Fig. 9. Intercom protocol stack

if detects an error, discards the incorrect packet. The *Synchronization Layer* extracts from the received bit stream the frame and slot synchronization patterns, and notifies the MAC of the beginning of a new slot. The *Physical Layer* includes computation-intensive bit-level data processing functions, e.g. modulation, timing recovery. The Intercom protocol specification includes both data processing and control functions. It includes computation intensive functions (e.g. error control, logarithmic quantization) that are applied to the flows of (a) data samples and (b) service request/acknowledgment messages both in transmission and reception [11] . Control functions include time-dependent and data-dependent control. The first type occurs at the MAC layer, where the processing and the transmission (or reception) of data flows is enabled using a time-based (TDMA) mechanism. Error control is instead a data-dependent control function since it enables some actions (e.g. discarding a packet) depending on the result of data computation.

This mix of control-intensive and data-intensive aspects, often in the same functional layer, means that no design approach that separates between the two will provide (1) clear and concise specification of the functionality and (2) synthesis and optimization capabilities.

To model the Intercom protocol specification we have initially used ACFSMs. Then, starting from the communication requirements and taking into account the assumptions on the behavior of the environment, e.g. on the rates and patterns of the input events, we have refined the ACFSMs into implementable ECFSMs

[11] Service request and acknowledgment messages are part of the control function at the network level, as they contribute to the network coordination. However, at the node level, request messages are still data units processed and transmitted over the wireless channel.

and selected a communication protocol for each channel (currently this step is manual, but we are exploring ways of automating it). Since high-quality voice communication requires that no data is lost within the protocol stack due to buffer overflow, we have refined ACFSMs into *lossless* ECFSMs. Additional timing requirement are imposed by the TDMA policy: data transmission can occur only within TDMA slots. Environmental assumptions concern the occurrence of the following input signals: *Voice samples* is periodic at 64 kbps, *Rx_data* is a periodic bit stream at 1.6 Mbps. Let us consider in detail the behavior of a protocol stack fragment. Incoming voice samples are first processed by the Mulaw and sent to the CRC module that at each transmission slot is enabled by the MAC. The CRC input FIFO shapes the data stream to make it fit the TDMA slot allocation pattern and, since the read and write operations from this queue occur at regular times, in this case, the FIFO size can be simply computed from input/output rates and slotsize, and chosen to be 500 bytes. A smaller size would require to block the Mulaw module and introduce another queue at the input of Mulaw. A greater size does not give any advantage since 500 bytes is the maximum amount of data arriving during a frame.

If we used the classical CFSMs model instead of ACFSMs, we would be forced to represent each FIFO as a CFSM. A CFSM of type FIFO calls, upon occurrence of external events, internal read and write functions that access an "internal" memory in FIFO manner. This implies that the CFSM at the receiver end of the channel (in this case CRC) has to explicitly make a request (in the form of an event) for new data as soon as it can process it. This results in an unnecessary overhead that is not present in the ACFSM model where FIFOs are part of the model and can be directly accessed by simple communication primitives.

Finally, we also compared the design of the Intercom using our ACFSM-based methodology with a previous design [9] of the same specification done following an SDL-based approach. The overall design process turned out to be easier and more effective using ACFSMs due to their capability of modeling control and dataflow components independently from the final implementation. The previous approach, instead, due to SDL's purely asynchronous nature, required to partition into hardware and software components from the beginning and model the hardware components directly in VHDL. This greatly reduced the design space and prevented from some system optimizations.

5 Conclusions

In this paper, we presented a new way of formalizing communication among processes, and refining and implementing it using physical channels and protocol stack layers. We then introduced a new Model Of Computation called Abstract Codesign Finite State Machines, in which a finite state control coordinates multi-rate transitions and data-intensive computations. We described how a specification using ACFSMs can be refined, by queue sizing and static scheduling, until communication becomes implementable. We used a real-life example,

a wireless communication protocol, to illustrate how the abstract communication can be refined to a concrete, implementable one, and we discussed the main trade-offs involved.

Acknowledgements

We gratefully acknowledge the discussions about Models of Computation with Ken McMillan and Roberto Passerone and about communication protocols with Jan Rabaey. This research is sponsored in part by the Giga-scale Silicon Research Center (GSRC) and the Consiglio Nazionale delle Ricerche, Progetto MADESSII.

References

1. F. Balarin and al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997. 31, 32, 38, 42
2. J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Special issue on Simulation Software Development, Jan. 1990. 31, 33
3. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–29, December 1998. 33
4. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings IFIP Congress*, Aug. 1974. 33, 40
5. E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, Sept. 1987. 44
6. M.Sgroi, L.Lavagno, Y.Watanabe, and A.Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings of the Design Automation Conference*, June 1999. 44
7. J. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the Design Automation Conference*, pages 178–183, 1997. 31
8. R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the Design Automation Conference*, pages 8–13, June 1998. 35
9. M. Sgroi, J. da Silva jr., F. D. Bernardinis, F. Burghardt, A. Sangiovanni-Vincentelli, and J. Rabaey. Designing Wireless Protocols: Methodology and Applications. In *Proceedings of the ICASSP Conference*, June 2000. 44, 46

Programming Access Control: The KLAIM Experience

Rocco De Nicola[1], GianLuigi Ferrari[2], and Rosario Pugliese[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
{denicola,pugliese}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa
giangi@di.unipi.it

Abstract. In the design of programming languages for highly distributed systems where processes can migrate and execute on new hosts, the integration of security mechanisms is a major challenge. In this paper, we report our experience in the design of an experimental programming language, called KLAIM, which provides mechanisms to customize access control policies. KLAIM security architecture exploits a capability-based type system to provide mechanisms for specifying and enforcing policies that control uses of resources and authorize migration and execution of processes. By means of a few programming examples, we illustrate the flexibility of the KLAIM approach to support the specification of control policies and to guarantee their enforcement.

1 Introduction

Highly distributed systems and networks have now become a common platform for large scale distributed programming. Network applications distinguish themselves from traditional applications for *scalability* (huge number of users and nodes), *connectivity* (both availability and bandwidth), *heterogeneity* (operating systems and application software) and *autonomy* (of administration domains having strong control of their resources). These features call for programming languages that support mobility of code and of computations, and for effective infrastructures that support coordination of dynamically loaded (often untrusted) software modules. Current software computing environments exploit so called *security architectures* to monitor the execution of mobile code and protect hosts from external attacks. We refer the readers to [18] for a collection of papers dedicated to tackling these issues.

Recently, the possibility has been explored of considering security issues at the level of language design, aiming at embedding dynamic linking of code and protection mechanisms in programming languages. The challenge is that of designing the language together with its secure kernel and to implement the corresponding secure abstract machine. In other words, security issues are taken into account when designing the programming language and not later for addition to existing infrastructures. This provides the users with suitable programming

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 48–65, 2000.

mechanisms for defining their own security policies. A partial example of this approach is given by the *Java 1.2 Security architecture* [24].

In this paper we report our experience in the design of a language which supports programming of security policies. We describe the security mechanisms of KLAIM (*a Kernel Language for Agents Interaction and Mobility*) [13], an experimental programming language specifically designed for programming network applications. KLAIM provides direct support for expressing and enforcing access control policies to resources and for authorizing migration and execution of mobile processes. KLAIM exploits a capability-based type system to specify and enforce access control policies.

KLAIM consists of core Linda [11,10,5] with multiple located tuple spaces. A KLAIM program, called a net, is structured as a collection of nodes. Each node has a name, and consists of a process component and a tuple space component. Sites are the *concrete* names of the nodes and are the main linguistic constructs to provide network references to nodes. Processes may access tuple spaces through explicit naming: operations over tuple spaces are indexed with their *locality*. Localities are the *symbolic* names of nodes and programmers need not to know the concrete network references, i.e. the precise association of localities with sites. The net primitives are designed to handle all issues related to physical distribution, scoping and mobility of processes: the visibility of localities, the allocation policies of tuple spaces, the scoping disciplines of mobile agents, etc. In other words, KLAIM nets provide the distributed infrastructure to coordinate processes that access and share resources over a highly distributed system.

KLAIM exploits a capability-based type system to specify and enforce access control policies. Capabilities provide information about the intentions/rights of processes: reading/consuming tuples, producing tuples, activating processes, and creating new nodes. Capabilities have a hierarchical structure; for example we assume that a processes authorized to consume a tuple has also the right of reading it. The hierarchy of capabilities is reflected in the subtype relation over access types. The underlying idea is that if a process `P` has access type `ac1` and this is a subtype of `ac2` then `P` could be granted access type `ac2` as well. In other words, `P` can be safely used in all contexts where a process of type `ac2` is expected. The (access) type of a process specifies the operations the process intends to perform at the localities of a net. The (access) type of a node specifies the access control policy of that node with respect to the nodes of the nets.

Enforcement of access control policies is performed by the *type checker* which controls that the loaded software modules match the access policies of the nodes. Only processes (stationary or mobile) that have successfully passed the type checking phase can be executed.

The clear distinction between specification and enforcement of access control policies is a key design element of the type system. Indeed, the development of KLAIM applications proceeds in two phases.

In the first phase, processes are programmed while ignoring the precise physical allocations of tuple spaces and the access rights of processes. By exploiting type annotations and mechanisms for code inspection, a type inference system

assigns types to processes and checks whether processes behave consistently with their type declarations.

In the second phase, processes are allocated over the nodes of the net. It is in this phase that access control policies are set up as a result of a coordination activity among the nodes of the net.

This paper illustrates the KLAIM access control model and shows how a capability-based type system is a useful and effective tool to write secure network applications. The main features of the approach are summarized below.

- Access rights are explicitly recorded in type specification, thereby providing declarative specifications of access control policies.
- Subtyping of access rights: alternative access policies can be defined by replacing the default policy with a new, more restrictive policy that is a subtype of the default policy.
- Mobile processes are typed by their access control requirements; these are automatically generated by the type inference procedure.
- Processes access control requirements are clearly separated from nodes access control policies: the node access control policy is formulated by the authority (the owner) of the node and is dynamically enforceable at run-time.
- KLAIM type system is sufficiently powerful to express access patterns of policies for customizing and confining the route of process migration.

Figure 1 below, illustrates the basic ingredients of our approach.

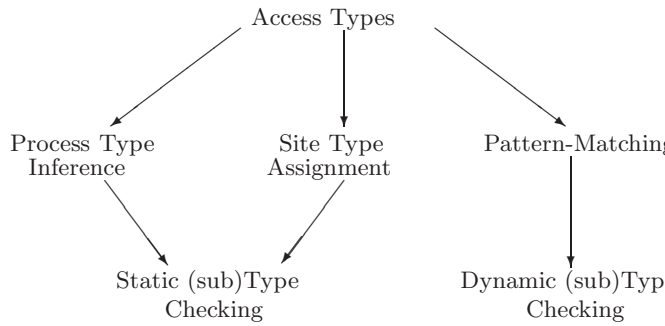


Fig. 1. KLAIM type system: main ingredients

The language and the design philosophy underlying KLAIM are presented in [13]. The mathematical foundations (decidability and soundness) of the kernel of KLAIM type system can be found in [15] (a preliminary presentation appeared in [12]). The prototype implementation of the language is described in [2].

The rest of the paper is organized as follows. In the next section we briefly review KLAIM and its capability-based type system. Section 3 illustrates how KLAIM primitives allow programming of complex interaction protocols; more

specifically, we consider three well-known paradigms for distributed computing, namely, mobile agent, code-on-demand and remote evaluation. We also show how the capability-based type system can be exploited to address the security needs of the three paradigms. Section 4 shows how the run-time type-checking mechanisms of KLAIM can be exploited to impose access control policies. In particular, we show how to impose restrictions on mobile processes concerning:

- the access to tuples from one or more localities;
- the exchange of tuples among groups of localities;
- the trajectory of process migrations.

The final section summarizes our approach and suggests future extensions.

2 The KLAIM Programming Model

We begin this section by summarizing the Linda programming model. Then, we introduce the main features of KLAIM by means of simple examples. The complete syntax of the language is reported in the Appendix A.

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called Tuple Space (TS). A tuple space is a multiset of tuples, that are sequences of actual fields (expressions or values) and formal fields (variables). *Pattern-matching* is used to select tuples in a TS. Linda provides four main primitives for handling tuples: two (non-blocking) operations add tuples to a TS, two (possibly blocking) operations read/withdraw tuples from the TS. The Linda asynchronous communication model allows programmers to explicitly control interactions among processes via shared data and to use the same set of primitives both for data manipulation and for process synchronization.

2.1 Nets and Processes

We now introduce the KLAIM primitives without considering typing issues. KLAIM programs are structured around the notions of *localities*, *tuples*, *tuple spaces*, *nets* and *processes*.

Localities (`l, l', ...`) can be thought of as the symbolic names for sites. Sites (`s, s', ...`) are the addresses (network references) of nodes. The bindings between localities and sites are stored in the *allocation environment* of each node of the net. Existence of a distinguished locality `self` is assumed: the `self` locality is used by processes to refer to their current execution site.

Tuples contain information items; their fields can be actual fields (i.e. expressions, localities, processes) and formal fields (i.e. variables). Syntactically, a formal field takes the form `!ide`, where `ide` is an identifier. For instance, the sequence `(''foo'', 25, !u)` is a tuple with three fields. The first two fields are basic values (a string value and an integer value) while the third field is a locality variable. Similarly, `(Agent<Itinerary, RetLloc>, !List, Site)` is a tuple whose first field is a process (with two parameters).

Tuple spaces are collections (multisets) of tuples. *Pattern-matching* is used to select elements from a tuple space. Two tuples match if they have the same number of fields and corresponding fields have matching values or variables. Variables match any value of the same type, and two values match only if they are identical.

KLAIM *nets* provide the infrastructure to coordinate users accessing and sharing a set of resources over a configurable distributed system. Nets are sets of nodes; each node consists of a site `s`, an allocation environment `e`, a set of running processes `P` and a tuple space `T`. Sites stand for physical places with a boundary (e.g. host machines, firewalls, administration domain).

A node (an active KLAIM abstract machine) embodies both the active computational units (processes) and the resources (tuples). KLAIM communication paradigm is anonymous (tuples have no name) and associative (tuples are content addressable). This form of communication mechanism has been recognized as a suitable tool to program network services where the set of available services (coded as tuples in the tuple spaces) at any given moment may dynamically change (see [1,23]).

Technically, tuple spaces are modelled as processes emitting tuples, namely processes of the form **out**`(t)` where `t` is an *evaluated tuple*, i.e. a tuple where all fields are ground (they do not include variables). For instance, **out**`(''foo'', 1)` defines the tuple `(''foo'', 1)`.

The allocation environment `e` constraints network connectivity: it basically behaves as a proxy mechanism for the processes allocated at a certain node.

The following code

```
node s :: e {P | T}
```

defines a node, where `s` is the site, `e` the allocation environment, `P` the set of running processes and `T` the tuple space. Similarly, the following code

```
node s1 :: e1 {P1 | T1} ||
node s2 :: e2 {P2 | T2} ||
node s3 :: e3 {P3 | T3} .
```

defines a net with 3 nodes.

Processes are the active computational units. They can perform five different basic operations, called *actions*, that permit reading from a tuple space, withdrawing from a tuple space, writing in a tuple space, activating new threads of execution and creating new nodes.

The operation for retrieving information from a node has two variants: **in**`(t)@l` and **read**`(t)@l`. Action **in**`(t)@l` evaluates the tuple `t` and looks for a matching tuple `t'` in the tuple space located at `l` (`l` is the logical address of the tuple space). Whenever the matching tuple `t'` is found, it is removed from the tuple space. The corresponding values of `t'` are assigned to the variables in the formal fields of `t` and the operation terminates; the new bindings are used by the continuation of the process that has executed **in**`(t)@l`. If no matching tuple is found, the operation is suspended until one becomes available.

Action **read**(t)@l differs from **in**(t)@l only because the tuple t' selected by pattern-matching is not removed from the tuple space.

The operation for placing information on a node has, again, two variants: **out**(t)@l and **eval**(P)@l. The operation **out**(t)@l adds the tuple resulting from the evaluation of t to the tuple space located at l. The operation **eval**(P)@l spawns a process (whose code is given by P) at the node located at l.

The operation for creating new nodes is **newloc** which takes as arguments a list of locality variables. For instance, **newloc**(u1, u2, u3) dynamically creates 3 distinct new sites that can only be accessed via locality variables u1, u2, u3.

Notice that variables occurring in KLAIM processes can be bound by actions. More precisely, the actions **in**(t)@l and **read**(t)@l act as binders for variables in the formal fields of the tuple t. The action **newloc** binds the locality variables taken as arguments.

We now provide some simple examples of KLAIM programs; more advanced programming examples will be presented later.

KLAIM provides two mechanisms of mobility. The action **eval** moves a process code to a remote site "cutting" the links to the local resources, i.e. by adopting a dynamic scoping discipline. The action **out** can move a piece of code to a remote site while maintaining the links to the resources allocated at the original site, i.e. by adopting a static scoping discipline.

Our first example illustrates a process that moves along the nodes of a net with a fixed binding of localities to sites. We consider a net consisting of two sites s1 and s2. A client process Client is allocated at site s1 and a server process Server is allocated at site s2. The server process can accept clients for execution. The client process sends process Q to the server. This is implemented by the following code:

```
def Client = out(Q)@l1 ; nil
def Q = in(''foo'', !x)@self; out(''foo'', x+1)@self; nil
def Server = in(!X)@self; eval(X)@self; nil
```

The behaviour of the above processes depends on the meaning of l1 and self. It is the allocation environment that establishes the links between localities and sites. Here, we assume that the allocation environment of node s1, namely e1, associates self to s1 and l1 to s2, while the allocation environment of site s2, e2, associates self to s2. Finally, we assume that the tuple spaces located at s1 and s2 both contain the tuple (''foo'', 1). The following KLAIM code defines the net outlined above:

```
node s1 :: e1 { Client | out(''foo'', 1)} ||
node s2 :: e2 { Server | out(''foo'', 1)}.
```

The client process Client sends process Q for execution at the server node (locality l1 is bound to s2 in the allocation environment e1). After the execution of the action **out**(Q)@l1, the tuple space at site s2 contains a tuple where the code of process Q is stored. Indeed, the process stored in the tuple is

```
Q' = in(''foo'', !x)@s1; out(''foo'', x+1)@s1 ; nil
```

because the localities occurring in `Q` are evaluated using the environment at `s1` where the action **out** has been executed. Hence, when executed at the server's site the mobile code increases the tuple at the client's site. Fig. 2 gives a pictorial representation of this alternative.

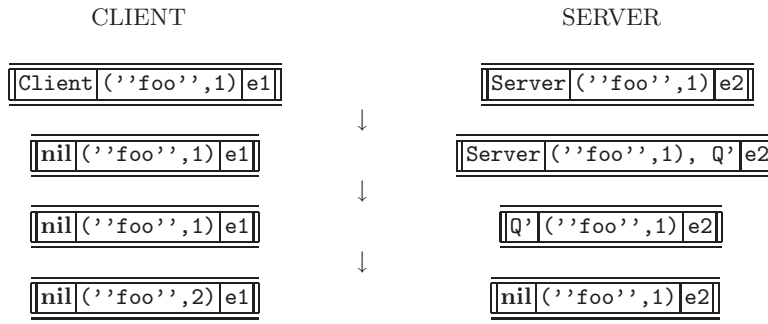


Fig. 2. Agent Mobility: Static Scoping

Our second example illustrates how mobile agents migrate with a *dynamic scoping* strategy. In this case the client process `Client` is defined by the code **eval**`(Q)@l1` **;** **nil**. When action **eval**`(Q)@l1` is executed, the code of the process `Q` is spawned at the remote node *without* evaluating its localities according to the allocation environment `e1`. Thus, the execution of `Q` will depend only on the allocation environment `e2` and, hence, `Q` will increase the tuple at the server's site. Fig. 3 illustrates this alternative.

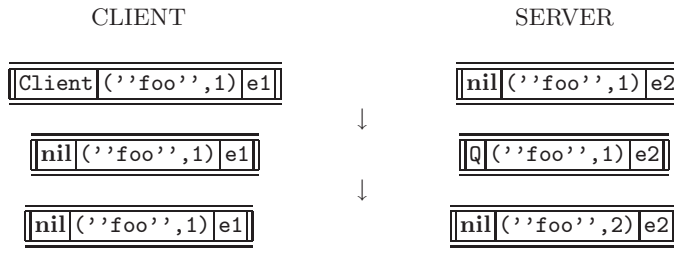


Fig. 3. Agent Mobility: Dynamic Scoping

2.2 Types for Access Control

KLAIM makes use of a capability-based type system to express and enforce access control policies. Capability-based types provide information about permissions of processes: downloading/consuming tuples, producing tuples, activating processes, and creating new nodes.

We use { `r, i, o, e, n` } to indicate the set of permissions, where each symbol stands for the operation whose name begins with it (e.g. `r` denotes the permission of executing a **read** action).

The type `Bottom` is used to express no requirement (no action) by processes. Conversely, the type `Top` denotes the intention of performing any kind of operations. Moreover, **atype** and **ttype** denote access types and tuples types, respectively.

A type of the form `l --> r[ttype] --> Bottom`, an *arrow type*, describes the permission/intension of performing, at location `l`, the action of reading a tuple of type **ttype** without imposing any further constraint on the remaining process (the continuation). The arrow type `l --> e --> atype` describes the permission of executing of process of type **atype** at location `l`. We often adopt the name *capability* to indicate the action permission of arrow types. For instance, `r[int, string]` is the capability of the arrow type `l --> r[int, string] --> Bottom`.

The union type "**atype, atype**" is used to join permissions. Recursive types are used for typing migrating recursive processes (we often use `'a` to denote type variables).

In general, permissions have a hierarchical structure: a process authorized to read a tuple with a `real` value has also the right of reading a tuple with an `int` value. Similarly, a process authorized to perform an **in** action may also perform a **read** action. The hierarchy of access rights is reflected in the subtype relation. The underlying idea is that if a process `P` has type `ac1` and `ac1` is a subtype of `ac2` then `P` could be considered as having type `ac2` as well. In other words, `P` can be safely used whenever a process of type `ac2` is expected.

The subtype relation $\preceq$ formalizes this intuition. The type `Bottom` semantically corresponds to the smallest type, and the type `Top` denotes the greatest type. Several typing judgments formally characterize the subtype relation. For instance, the following clause

`l --> r[int] --> Bottom` $\preceq$ `l --> r[real] --> Bottom`

states that a process looking for a tuple with an integer field can also be viewed as a process looking for a tuple with a real field. Motivations and formal properties of the subtype relation $\preceq$ can be found in [15].

KLAIM tuples are typed. Tuple types basically are record types. The distinguished tuple type **any** is used to have a form of *genericity* of tuples.

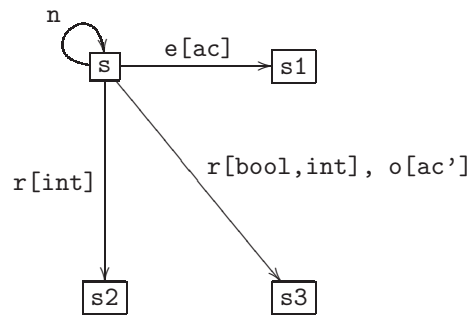
To express user-based access policies, localities and locality variables may be annotated with type specifications which are pairs of the form $\langle \lambda, \mathtt{ac} \rangle$ called *access lists* that are lists of bindings from localities to capabilities. The distinguished locality **all** is used to denote all nodes of a net. Access lists are exploited for different purposes:

- To restrict the kind of actions that a process can perform at (the site corresponding to) a specific locality that is transmitted in a tuple.
- To specify the access rights of newly created nodes with respect to the nodes of the net, and the vice versa.
- To specify the permissions of a process on actual arguments of process definitions.

Access types are also used to define the access control policy of nodes. For instance, the following access type, where `ac` and `ac'` are access types and tuple types respectively, describes the access policy of node `s`, i.e. the permissions of processes located at `s`

```
s --> n --> Bottom,
s1 --> e --> ac,
s2 --> r[int] --> Bottom,
s3 --> r[bool,int] --> Bottom,
s3 --> o[ac'] --> Bottom
```

For example, a process located at `s` is allowed to spawn a process of type `ac` at site `s1` and it is not allowed to perform any other action at site `s1`. Nodes access types have a natural representation as labelled graphs. The following graph represents the access type described above.



We now give a description of the use of types for access control. Let us consider a system consisting of a process `Server` and two identical `Client` processes. The server process

```
def Server = out(l:<void, Top>)@self; nil
```

adds a tuple, containing locality `l`, to its local tuple space and evolves to the terminated process **nil**. In our example the server process does not take advantage of the possibility of imposing further restrictions on the access policy of `l`: the pair `<void, Top>` states that the access restrictions at `l` are left unchanged.

The client process

```
def Client = read(!u:<[self--> e], ac>)@l-S ;
             eval(P)@u ; nil.
```

first accesses the tuple space located at `l-S` to read an address. Then, it assigns the value read to the locality variable `u` and, sends process `P` for execution at `u`. The pair `<[self--> e], ac>` specifies that `Client` needs a locality where it is possible to send for execution a process with (access) type `ac` from the site where `Client` is running[1].

Let us now consider a net where `Server` is allocated at site `s` and the two, identical, `Client` processes are at sites `s1` and `s2`, where `l-S` is bound to `s` to allow clients to interact with the server.

Under the assumption that code `P` does not violate the access requirements specified by the access type `ac` (i.e. the type inferred for `P` is a subtype of `ac`), the inference system determines the following type `c` for the process `Client`

```
c = l-S --> r[<[self--> e], ac>] --> Bottom
```

This type `c` is an abstraction of `Client` behaviour stating that the process aims at reading the address of a node where a code of type `ac` is allowed to run.

Notice that an important part of the programming task consists of adding type annotations to the code in order to specify the access policies. Our inference system inspects the annotated code to automatically determine the access policy requirements of the code.

Let us consider our running example and assume the following types for the permissions associated to the sites `s1` and `s2` where `Client` is allocated:

```
ac-s1 = s --> r[<[self--> e], ac1>] --> Bottom
ac-s2 = s --> r[<[self--> e], ac2>] --> Bottom
```

Both permissions state that a process can be executed at the address read at site `s`. However, in the case of site `s1` processes with privileges smaller that `ac1` can be executed at the read address; while processes with type smaller than `ac2` can be executed in the case of site `s2`.

The enforcement is performed by the *type checker* that verifies that the types of the processes loaded on a node are valid instances of the type which specifies the access control policy of the node. Processes are allocated and executed only if they are accepted by the type checker. In the type checking phase, the type of the process is obtained by evaluating its localities according to the allocation environment of the site where the process is allocated. Hence, access requests for sites which are not "visible" from the node lead to type failures.

In our running example assume that `c1` and `c2` are the instances of the type of the `Client` process in sites `s1` and `s2`, respectively. Further, assume that `c1` is a subtype of `ac-s1` and that `c2` requires more privileges than `ac-s2` then, only the `Client` process at site `s1` successfully passes the type checking phase and has the right of reading tuples at the server's site and, consequently, of sending processes for execution.

[1] In KLAIM `self` is a reserved keyword that indicates the current execution site

3 Paradigms for Mobility

Mobile Code Applications are applications running over a network whose distinctive feature is the exploitation of forms of "mobility". According to the classification proposed in [9], we can single out three *paradigms* that, together with the traditional *client-server* paradigm, are largely used to build mobile code applications. These paradigms are: *Mobile Agent*, *Code-On-Demand* and *Remote Evaluation*.

The first paradigm can be easily implemented in KLAIM by using the actions **eval** or **out**, in dependence on the chosen binding strategy (dynamic or static, respectively) adopted for localities. We did comment on this at the end of Section 3. In the rest of this section, we analyze the two remaining paradigms with specific attention devoted to the security aspects.

Code-on-Demand A component of an application running over a network at a given node, can dynamically download some code from a remote node to perform a given task.

To download and execute code stored in the tuple space located at `l` with certain access privileges (specified by the access type `ac`), we can use the process

read`(!X:ac)@l ; eval(X)@self;` **nil**.

The downloaded code is checked for access violations at run-time by the pattern-matching mechanism that checks the types of processes to ensure that they satisfy the type annotations specified by the programmer. Hence, process codes are checked before being downloaded. In other words, the pattern-matching operation performs a *run-time* type checking of incoming codes.

Remote Evaluation A component of a network application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.

To transmit both code `P` and data `v` of type `bt` at the locality `l` of the server, we can use the code

out`(`**in**`(!y:bt)@l ; A<y>, v)@l`

where `def A(x) = P`.

Here, we assume that the server adopts the following (code-on-demand) protocol

in`(!X:ac, !x:bt)@self;` **out**`(x)@self;` **eval**`(X)@self;` **nil**

To prevent "damages" from `P`, the server loads and executes code `P` only if the instance of the type of the code is a subtype of `ac`. Again, dynamic type checking is extensively used to ensure that dynamically executable codes adhere to the access policy of the node.

Type `ac` may give only minimal access permissions on the server's site, for instance, only the capability of reading a tuple consisting of a string and an integer value, and giving back the results of the execution. In this case, the access type is of the form

```
s --> r[string, int] --> Bottom,
s0 --> o[string, int] --> Bottom,
s1 --> o[Top] --> Bottom
```

where `s` is the server's site and `s0` is the site which invokes server's facilities and `s1` is another site.

Notice, however, that this does not prevent `P` from visiting other sites. In particular, code `P` may be programmed in such a way that it transmits some code `Q` with access type `ac-Q` at `l1`:

```
def P = read(!y:string, !x:int)@l ;
        out(op2(y), op1(x))@self;
        out(Q)@l1; nil
```

It is immediate to see that the instance of the type of code `P` at the server site is

```
s --> r[string, int] --> Bottom,
s0 --> o[string, int] --> Bottom,
s1 --> o[ac-Q] --> Bottom
```

Hence, the instance of the code `P` satisfies the access policy of node `s0`. However, code `Q` is only stored in the tuple space at `s1`, no new thread of execution is activated at that site. Before being executed code `Q` must be read and verified (dynamic type checking). Therefore, process `P` cannot silently activate a *Trojan horse* at the remote site `s1`.

4 Controlling Use of Resources

In the previous section, we examined the support offered by the KLAIM capability-based type system for access control by considering well-known paradigms of mobility. In this section, we show how to exploit KLAIM capability-based types to restrict the resources usage within a net. In particular, we show that KLAIM types are powerful enough to program and enforce policies which are usually handled by *low-level* techniques (e.g. sand-boxing and firewall). Moreover, the explicit typing of sites can be also used to define policies which limit mobile processes to specific route.

Restricting Interactions KLAIM action primitives operate on the whole net not just at the process current site. From the point of view of security mechanisms, communications among different sites of the net (i.e. remote communications) could be controlled and regulated. This corresponds to place over a node a form of *sandboxing* which treats all processes as potentially suspicious.

For instance, to make sure that a process running on a certain site `s` gets only local information, it suffices to constrain the type specifying the access control policy of the node to allow local communications only.

To this purpose, it suffice to state that for no site `s'` different from `s`, the type `s' --> r[any] --> Bottom` (or any of its subtypes) is a subtype of `ac-s`,

the access type of site `s`. Hence, a process `P` allocated at site `s` that is willing to perform remote **read**/**in** operations violates the access rights. To access tuples at a remote tuple space, a well-typed process must first move (if it has the required rights) to the remote site. Using a similar strategy also output actions can be forced to be local.

Firewall We now show how to specify a firewall by means of suitable typing. The idea is that the type of the site where the firewall is allocated specifies the access policy of network services. Assume that the firewall is set up at site `s` to protect sites `s-1`, ..., `s-n` from sites `S-1`, ..., `S-m`.

A first requirement is that processes located at site `S-j` cannot directly interact and ask for services at any site `s-i`. To this purpose we impose that for each site `S-j` it cannot be the case that `s-i --> cap --> Bottom`, where `cap` is any capability, is a subtype of the access type of the node (`ac-S-j`).

On the other hand, type `ac-S-j` may have subtypes of the form

```
s --> o[any] --> Bottom,
```

since each site `S-j` may send service requests to the firewall. Here we assume that the service request is stored in a tuple and we do not impose any restriction on the type of the tuple. This setting guarantees that all connections from sites `S-1`, ..., `S-m` to sites `s-1`, ..., `s-n` have to pass through site `s`, the firewall.

For instance, process

```
def P = out(eval(Q)@s-i)@s ; nil
```

complies with the access control policy, while process

```
def P' = eval(Q)@s-k ; nil
```

violates the access control policy of site `s-k`.

The firewall can be programmed to handle the requests according to certain policies. Typically, the firewall handles a service request according to the following pattern

```
read(!Request:ac-r) ; <RequestHandler>
```

where type `ac-r` specifies the access policy that the service request must satisfy. For instance, if

```
ac-r = s-1 --> i[any] --> Bottom,
       s-2 --> e --> (s-2 --> r[any] --> Bottom,
                      S-k --> o[any] --> Bottom)
```

then the two service requests

```
eval(read(!x:int)@self; out(op(x))@S-k ; nil)@s-2 ; nil
read(!x:int)@s-1 ; nil
```

both satisfy the type requirements and will be accepted. The service request

```
read(!x:int)@s-1 ; out(op(x))@S-k ; nil
```

violates the access policy and will be rejected.

The examples above do not specify any constraint on the tuples read. More refined access policies can be obtained by specifying the type of tuples. For instance, one can fix a policy where the read operations are constrained to get from the tuple space only 'plain' data tuples, namely tuples without localities or process codes.

KLAIM applications might need to define their own firewall, i.e. a user node which acts as a filter for the network traffic, without changing the default policies of the nodes of the net. Let us consider, for instance, the following user process:

```
def P = newloc( u-1: <al-1, ac-1>,
                u-2: <al-2, ac-2>,
                u-F: <al-F, ac-F>) ;
      <Firewall Handler>
```

Now, assume that both access lists `al-1` and `al-2` are of the form

```
[u-F --> cap]
```

for a suitable capability `cap`.

Process `P` creates a subnet whose sites are associated to locality variables `u-1` and `u-2` and the type annotations ensure that the subnet can be reached only through site `u-F` (the firewall). The user process specifies in the type specification of locality `u-F` the access policy to the firewall. For instance, setting `al-F = [self--> o[any]]` means that the firewall site can be accessed only from the site where the user process is running.

Fares and Tickets A primary access control policy consists of controlling the route of a mobile agent traveling in the net. For instance, if one has to configure a set of sites with new software, a mobile agent can be programmed to travel among the sites to install the new release of the software.

If the starting site of the trip is site `s-0` and sites `s-1`, `s-2`, ..., `s-n` have to be visited visited before getting back to the starting site, the following type can be used to specify the access policy of the trip:

```
ac-0 = ac, s-1 --> e --> ac-1
ac-1 = ac, s-2 --> e --> ac-2
                ⋮
ac-n-1 = ac, s-n --> e --> ac'
ac' = s-0 --> o[string] --> Bottom
```

The idea is that at each site type `ac` specifies the allowed operations (e.g. installing the new release of a software package); the remaining type information specifies the structure of the trip (which is the next site of the trip). When it reaches the last site of the trip, the agent has the rights of returning to the

original site the results of the trip (e.g. the notification that the installation was successful).

The type discussed above can be properly interpreted as the *fare* of the trip: an agent `M` can perform the trip provided that its type matches the fare, namely its instance at site `s-0` is a subtype of the type of the trip. Notice that this ensures that a malicious agent cannot modify the itinerary of the trip to visit sites other than those listed in its ticket.

5 Concluding Remarks

We have described the security architecture of the experimental network programming language KLAIM. This language provides users with a programmable support to configure application-specific access control policies by exploiting a capability-based type system. Although, the type system we designed is tailored to the KLAIM language, the general spirit of the approach can be applied to define types for access control of programming languages for highly distributed systems. We summarize below the main points:

- Declarative specification of access control policies via type annotations and type inference.
- Structured hierarchy of access rights based on subtyping.
- Clear separation of process requirements from node requirements.
- Static and dynamic type checking.
- Programmability and customization of access control policies.

The prototype implementation of KLAIM [2] is built on top of Java and Java security architecture (the sandbox model). The implementation of KLAIM access control models (in Java version 1.2) is in progress, however preliminary implementations have been already exploited to validate some design choices of the type system.

We plan to extend KLAIM access types to handle history dependent access control. In history dependent access control access requested are granted on the basis of what happened in the past of the ongoing computation. Moreover, the development of network applications raises other issues related to security. We plan to integrate in KLAIM other security mechanisms (both at the foundation level and at the implementation level). These include mechanisms for secure communication and authentication, and agent code security. As for related work, we do refer the interested reader to [14] where specific comments on [3,4,8,6,7,16,17,19,20,21,22] can be found.

Acknowledgments

The authors would like to thank Lorenzo Bettini, Michele Loreti and Betti Venneri for interesting discussions and comments. This work has been partially supported by Esprit working groups *CONFER2*, and *COORDINA*, and by MURST projects TOSCA and SALADIN.

A KLAIM Syntax

Basic Types
btype ::= `int` | `real` | `string` | `bool` | ...

Field Types
ftype := **btype** | **<alist, atype>** | **atype**

Tuple Types
ttype :: = **ftype** | **any** | **ftype, ttype** | **ttype, ftype**

Access Lists
alist ::= **void** | `[l -->e]` | `[l -->n]` | `[l --> r[ttype]]` |
`[l --> i[ttype]]` | `[l --> o[ttype]]` |
`[all -->e]` | `[all -->n]` | `[all --> r[ttype]]` |
`[all --> i[ttype]]` | `[all --> o[ttype]]` | **alist,alist**

Access Types
atype ::= `Bottom` | `Top` | `l --> r[ttype] --> Bottom` |
`l --> i[ttype] --> Bottom` | `l --> o[ttype] --> Bottom` |
`l --> n --> Bottom` | `l --> e --> atype`|
atype, atype| ‘a | `rec ‘a atype`

Locality Patterns
lpattern ::= `u:`**<alist, atype>** | `u:`**<alist, atype>, lpattern**

Actions
`Act` :: = **out**`(t)@L` | **in**`(t)@L` | **read**`(t)@L` | **eval**`(P)@L` |
newloc`(`**lpattern**`)`

Processes
`P` :: = **nil** | `Act ; P` | `P | P` | `A<P, L, v>`

Agent Definitions
`A ::= def A(P,L,v) = P`

Evaluated Tuples
`T :=` **out**`(t)` | `T | T`

Nets
`N ::= node s ::` `e { P | T }` **where atype** | `N || N`

Tuples
`t ::= f | (f , t)`

Fields
`f ::= v | P | l:`**<alist, atype>** |
!x:btype | **!X:atype** | `!u:`**<alist, atype>**

References

1. K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, B. O'Sullivan. *The Jini specification*. Addison Wesley, 1999. 52
2. L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese. Interactive Mobile Agents in X-Klaim. *IEEE Seventh International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, *Proceedings* (P. Ciancarini, R. Tolksdorf, Eds.), IEEE Computer Society Press, 1998. 50, 62
3. C. Bodei, P. Degano, F. Nielson, H. R. Nielson. Control Flow Analysis for the π-calculus. *Concurrency Theory* (CONCUR'98), *Proceedings* (D. Sangiorgi, R.de Simone, Eds.), *LNCS* 1466, pp.611-638, Springer, 1998. 62
4. G. Boudol. Typing the use of resources in a Concurrent Calculus. *Advances in Computing Science* (ASIAN'97), *Proceedings* (R. K. Shyamasundar, K. Ueda, Eds.), *LNCS* 1345, pp.239-253, Springer, 1997. 62
5. N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, 1989. 49
6. L. Cardelli, A. Gordon, Mobile Ambients. *Foundations of Software Science and Computation Structures* (FoSSaCS'98), *Proceedings* (M. Nivat, Ed.), *LNCS* 1378, pp.140-155, Springer, 1998. 62
7. L. Cardelli, A. Gordon, Types for Mobile Ambients. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1999. 62
8. L. Cardelli, G. Ghelli, and A. Gordon, Mobile Types for Mobile Ambients To appear *ICALP'99*, *LNCS* , Springer, 1999. 62
9. A. Fuggetta, G. P. Picco, G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, Vol.24(5):342-361, IEEE Computer Society Press, 1998. 58
10. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, ACM Press, 1985. 49
11. D. Gelernter, N. Carriero, S. Chandran, et al. Parallel Programming in Linda. *Proc. of the IEEE International Conference on Parallel Programming*, pp. 255-263, IEEE Computer Society Press, 1985. 49
12. R. De Nicola, G. Ferrari, R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. *Coordination Languages and Models* (COORDINATION'97), *Proceedings* (D. Garlan, D. Le Metayer, Eds.), *LNCS* 1282, pp. 220-237, Springer, 1997. 50
13. R. De Nicola, G. Ferrari, R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, Vol.24(5):315-330, IEEE Computer Society Press, 1998. 49, 50
14. R. De Nicola, G.-L. Ferrari, R. Pugliese. "Types as Specifications of Access Policies", In J. Vitek, C. Jensen (Eds) [18], pp.117-146. 62
15. R. De Nicola, G. Ferrari, R. Pugliese, B. Venneri. Types for Access Control. To appear, *Theoretical Computer Science*, 2000. Available at `http://rap.dsi.unifi.it/papers.html`. 50, 55
16. G. Necula. Proof-carrying code. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1997. 62
17. J. Riely, M. Hennessy. Trust and Partial Typing in Open Systems of Mobile Agents. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1999. 62
18. J. Vitek, C. Jensen (Eds). *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS State-Of-The-Art-Survey, *LNCS* 1603, Springer, 1999. 48, 64

19. R. Stata, M. Abadi. A Type System for Java Bytecode Verifier. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1998. 62
20. J. Vitek, G. Castagna. A Calculus of Secure Mobile Computations. *Proc. of Workshop on Internet Programming Languages*, Chicago, 1998. 62
21. D. Volpano, G. Smith. A typed-based approach to program security. *Theory and Practice of Software Development* (TAPSOFT'97), *Proceeding* (M. Bidoit, M. Dauchet, Eds.), *LNCS* 1214, pp.607-621, Springer, 1997. 62
22. D. Volpano, G. Smith. Secure Information Flow in a Multi-threaded Imperative Language. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1998. 62
23. Sun Microsystems. The JavaSpace Specifications. `http://java.sun.com/products/javaspaces`, 1997. 52
24. Li Gong. The Java Security Architecture (JDK 1.2). `http://java.sun.com/`, 1998. 49

Exploiting Hierarchical Structure for Efficient Formal Verification

Rajeev Alur

Department of Computer and Information Science
University of Pennsylvania
http://www.cis.upenn.edu/~alur

Model checking is emerging as a practical tool for automated debugging of reactive systems such as embedded controllers and network protocols [9,11]. Since model checking is based on exhaustive state-space exploration, and the size of the state space of a design grows exponentially with the size of the description, scalability remains a challenge. The goal of our recent research has been to develop techniques for exploiting the modular design structure during model checking. Software design notations (e.g. STATECHARTS [10], UML) provide rich constructs to facilitate modular descriptions. Instead of destroying this structure by compiling the descriptions into flat state-transition graphs, we are developing algorithms that are optimized for modular descriptions. In this talk, we summarize research concerning two different notions of hierarchy: architectural and behavioral.

Architectural Hierarchy

The hierarchical decomposition of a system architecture into subcomponents can be well described in the modeling language of *Reactive Modules* [4]. In this language, processes communicate via shared variables, and can be combined using the operations of parallel composition, instantiation, and variable hiding. The theory of reactive modules supports modular refinement, and the language is supported by the model checker MOCHA (see `www.cis.upenn.edu/~mocha`). We consider three ways of exploiting the modular structure for automated verification:

Hierarchic reduction. We have developed an on-the-fly reduction algorithm that exploits the visibility information about transitions in a recursive manner based on the architectural hierarchy [7].

Assume guarantee reasoning. Refinement checking corresponds to trace containment between implementation and specification modules. Compositional and assume-guarantee proof rules allow for decomposing the refinement check into subproblems, and are integrated in the model checker.

Games. The requirements are expressed as formulas of *Alternating Temporal Logic* [5], which allows the formal specification of requirements that refer to collaborative as well as adversarial relationships between modules. The game semantics of ATL allows the construction of the most general environments of individual components, which can be exploited to automate modular verification [1].

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 66–68, 2000.

Behavioral Hierarchy

The control structure for the behavior of an atomic component can be described hierarchically in the modeling language of *Hierarchical Modules* [2]. The language has powerful features such as nested modes, mode reuse, exceptions, group transitions, history, and conjunctive modes, and is being implemented in the model checker HeRMes. We survey three related research directions:

Complexity and expressiveness. Due to sharing of modes, flattening of a hierarchical state machine causes an exponential blow-up. Recent research has results on succinctness of hierarchical construct, the complexity of operations such as emptiness and equivalence, and algorithms for model checking linear-time and branching-time requirements [8,6].

Modular refinement. While in the style of Statecharts in spirit, Hierarchical Modules have well-typed entry- and exit-points, and communication via variables with standard scoping rules. These features allow for a compositional trace semantics for modes that supports a refinement calculus with assume-guarantee proof rules [2].

Model checking. We have developed heuristics for enumerative as well as symbolic reachability analysis of hierarchical modules that exploit the scoping of variables and stack-like structure of individual states [3].

Acknowledgments

Mocha is a joint project with University of California at Berkeley. Research on Mocha and HeRMes is supported by Bell Labs, and by grants from the NSF, DARPA, and SRC.

References

1. R. Alur, L. de Alfaro, T. Henzinger, and F. Mang. Automating modular verification. In *Proc. of 10th CONCUR*, LNCS 1664, pages 82–97, 1999. 66
2. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proc. of 27th POPL*, pages 390–402, 2000. 67
3. R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Proc. of 12th CAV*, 2000. 67
4. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. 66
5. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proc. of 38th FOCS*, pages 100–109, 1997. 66
6. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Proc. 26th ICALP*, pages 169–178. 1999. 67
7. R. Alur and B.-Y. Wang. "Next" heuristic for on-the-fly model checking. In *Proc. of 10th CONCUR*, LNCS 1664, pages 98–113. Springer-Verlag, 1999. 66
8. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proc. of 6th FSE*, pages 175–188. 1998. 67

9. E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996. 66
10. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. 66
11. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997. 66

From Process Calculi to Process Frameworks

Philippa Gardner*

Computer Laboratory, University of Cambridge
pag20@cl.cam.ac.uk

Abstract. We present two process frameworks: the *action calculi* of Milner, and the *fusion systems* of Gardner and Wischik. The action calculus framework is based on process constructs arising from the π-calculus. We give a non-standard presentation of the π-calculus, to emphasise the similarities between the calculus and the framework. The fusion system framework generalises a new process calculus called the π_F*-calculus*. We describe the π_F-calculus, which is based on different process constructs to those of the π-calculus, and show that the generalisation from the calculus to the framework is simple. We compare the frameworks by studying examples.

1 Introduction

Our aim in studying process frameworks is to integrate ways of reasoning about related interactive behaviour. The purpose is not to provide a general, all-encompassing system for describing interaction—that would miss the point that there are many kinds of interactive behaviour worth exploring. Rather, the aim is to study interaction based on some fixed underlying process structure. A process framework consists of a structural congruence, which establishes the common process constructs used in the framework, and a reaction relation which describes the interactive behaviour of particular processes specified by a signature. The higher-order term rewriting systems have a similar format; the structural congruence describes the higher-order features given by the λ-terms with β-equality, and a rewriting relation describes the reaction between terms generated from a signature.

In this paper, we describe two process frameworks: the *action calculi* of Milner [Mil96], and the *fusion systems* of Gardner and Wischik [GW99]. The action calculus framework is based on process constructs arising from the π-calculus. We give a non-standard presentation of the π-calculus, to emphasise the similarities between the calculus and the framework. We also present the fusion system framework which generalises a new process calculus, called the π_F*-calculus* [GW00], in much the same way as the action calculus framework generalises the π-calculus. The π_F-calculus is similar to the π-calculus in that its interactive behaviour is based on input and output processes, and different in that its underlying process structure is not the same. We describe the π_F-calculus

* The author acknowledges support of an EPSRC Advanced Fellowship.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 69–88, 2000.

and the fusion system framework, and illustrate that the generalisation from the calculus to the framework is simple. We also compare the two frameworks by studying specific examples.

1.1 From the π-Calculus to Action Calculi

We describe the transition from the π-calculus to the action calculus framework. By adapting Milner's presentation of the π-calculus in [Mil99], we show that this transition is comparatively simple. Instead of the input and output processes specific to the π-calculus, an action calculus consists of more flexible *control processes*. The interactive behaviour of such control processes is given by a reaction relation specific to the particular action calculus under consideration. The other constructs of an action calculus are essentially the same as those of our π-calculus, except that there is no restriction. Instead the π-process $(\nu x)P$ is interpreted by an action process $\mathsf{new} \cdot (x)P$, where the control process new blocks the access to the name-abstraction. We explore two examples of action calculi: the π action calculus corresponding to the π-calculus, and the λ_v action calculus corresponding to a call-by-value λ-calculus. These examples illustrate that the basic process constructs for action calculi are indeed natural. We also refer to recent results of Leifer and Milner [LM00], who provide general bisimulation congruences for (simple) reactive systems and aim to extend their results to action calculi.

1.2 The π_F-Calculus

The π_F-calculus is similar to the π-calculus in that it consists of input and output processes that interact with each other. It is different from the π-calculus in the way that these processes interact. In a π-reaction, the intuition is that names are sent along a channel name to replace the abstracted names at the other end. In contrast, the π_F-calculus does not have abstraction. Instead, a π_F-reaction is directionless with names being *fused* together during reaction. This fusion mechanism is described using a *explicit fusion* $\langle z = y \rangle$, which is a process declaring that two names can be used interchangeably. A natural interpretation is that names refer to addresses, with explicit fusions declaring that two names refer to the same address.

The π_F-calculus is similar in many respects to the fusion calculus of Parrow and Victor [PV98], and the χ-calculus of Fu [Fu97]. The key difference is how the name-fusions have effect. In the π_F-calculus, fusions are explicitly recorded and have effect though the structural congruence. This has the consequence that the π_F-reaction is a simple local reaction between input and output processes. In the fusion calculus on the other hand, fusions occur implicitly within the reaction relation. This outcome of this is a non-local reaction relation.

We provide a natural bisimulation congruence for the π_F-calculus. We also have simple embeddings for the π-calculus, the fusion calculus and a call-by-value λ-calculus in the π_F-calculus. Intriguingly, the λ-embedding is purely compositional, in contrast with the analogous embeddings in the π-calculus and fusion

calculus. Further details can be found in [GW00]. We summarise the results in this paper, using the examples of fusion systems described below.

1.3 Fusion Systems

The generalisation from the π_F-calculus to the fusion system framework is simple. The basic process constructs and the definition of the structural congruence are the same. As with action calculi, fusion systems have the flexibility to describe other forms of interactive behaviour by adapting the input and output processes of the π_F-calculus to more general control processes. The interactive behaviour of the control processes is again given by a reaction relation, which is specific to the particular fusion system under consideration. We explore the π_F fusion system, which corresponds exactly to the π_F-calculus, and the λ_v fusion system which describes a call-by-value λ-calculus. The λ_v fusion system is a subsystem of the π_F fusion system. This simple relationship depends on the explicit fusions. It means that the bisimulation congruences for the π_F-system easily transfer to the λ_v-system. We are currently exploring a link between a bisimulation congruence for the λ_v-system and a corresponding behavioural congruence for the call-by-value λ-calculus. It remains further work to provide general bisimulation results for fusion systems. However, this is not our primary concern. First, we would like to explore specific examples of fusion systems to illustrate the expressiveness of explicit fusions.

Acknowledgements The work on the π_F-calculus and fusion systems is joint with Lucian Wischik.

2 From the π Calculus to Action Calculi

In [Mil99], Milner presents the π-calculus using a structural congruence which describes the underlying process structure, and a reaction relation which describes the interaction between input and output processes. In particular, he builds processes using abstractions and concretions as interim constructs, and generates the reaction relation by the axiom

$$\overline{x}.P \,|\, x.Q \searrow P \cdot Q,$$

where the operator $\cdot$ is a derived operator expressing the application of a concretion to an abstraction. In contrast, we treat all the constructs in the same way. We consider the application operator as a primitive operator. Although it can be derived in the π-calculus, it cannot be derived in the action calculus framework. We regard abstractions as processes, but instead of the concretion $(\nu z)\langle y\rangle P$ for example, we have the process $(\nu z)(\langle y\rangle \,|\, P)$ where the process $\langle y\rangle$ is called a *datum*. The reason for focusing on datums rather than concretions is that variables of the λ-calculus translate to datums in the corresponding action calculus. The datums therefore seem more fundamental.

In order to define reaction precisely, we require the correct number of datums and abstractions to match. This information is given by the *arity* of the process. In general, the arity information can include quite complex typing information, such as the sorting discipline for the π-calculus given in [Mil99]. In this paper we keep the arity structure simple, and just use it to count abstractions and datums.

There are two natural approaches for defining simple arities for π-processes. One approach keeps the abstractions and datums separate, so that we can form $(x)P$ and $\langle x\rangle \mid P$, but not $(x)(\langle x\rangle \mid P)$. In this case, arities are integers with the positive numbers counting the abstractions and the negative numbers the datums. The resulting process algebra corresponds to the π-calculus presented in Milner's book [Mil99]. The second approach mixes abstractions and datums, so that we have for example the process $(x)(\langle x\rangle \mid P)$. Arities have the form (m, n) for $m, n, \geq 0$, where m counts the abstractions and n the datums. We choose this second approach here, since it gives a smooth link to the action calculus framework. It remains further work however to fully justify the use of abstracted datums for the π-calculus. They are justified in the action calculus framework.

Following our presentation of the π-calculus, the description of the action calculus framework is comparatively simple. An action calculus essentially consists of the same underlying process constructs as our π-calculus, except that it does not have restriction. The definition of the structural congruence has to be modified however to account for the more general behaviour of application in the framework. Instead of input and output processes, an action calculus consists of general *control processes* of the form $K(P_1, \ldots, P_r)$ where the *control* K denotes some sort of boundary (or firewall) containing the processes P_i. The interactive behaviour of the control processes is described by a reaction relation.

For example, the input and output processes of the π-calculus correspond to action processes of the form $\langle x\rangle \cdot \mathsf{in}(P)$ and $\langle x\rangle \cdot \mathsf{out}(P)$ respectively. The controls in and out are specific to the π action calculus, and provide a boundary inside which reaction does not occur. Restriction is not primitive in the framework. Instead the restriction $(\nu x)P$ of the π-calculus is interpreted by $\mathsf{new} \cdot (x)P$ where new is a control which blocks the access to the abstraction.

2.1 The π-Calculus

Throughout this paper, we assume that we have an infinite set N of names ranged over by $u, \ldots, z$, and use the notation $\vec{z}$ to denote a sequence of names and $|\vec{z}|$ to denote the length of the sequence.

Definition 1
The set $\mathcal{P}'_\pi$ of *pre-processes* of the π-calculus is defined by the grammar

$P ::=$	nil	Nil process
	$P \mid P$	Parallel composition
	$P \cdot P$	Application
	$\langle x\rangle$	Datum
	$(x)P$	Abstraction

$(\nu x)P$	Restriction
$x.P$	Input Process
$\overline{x}.P$	Output process

The definitions of *free* and *bound* names are standard: the abstraction $(x)P$ and the restriction $(\nu x)P$ bind x in P, and x is free in the processes $\langle x \rangle$, $x.P$ and $\overline{x}.P$. We write $\mathsf{fn}(P)$ to denote the set of free names in P.

The application operator is used to apply the contents of input and output processes during reaction:

$$\overline{x}.P \mid x.Q \searrow P \cdot Q.$$

To define reaction properly, we require the correct number of datums in P and abstractions in Q to match. This information is given by the *arity* of a pre-process. We write $P : (m, n)$ to declare that a pre-process has arity (m, n), where m and n are natural numbers counting the abstractions and datums respectively.

Definition 2
The set $\mathcal{P}_\pi$ of *π-processes* of arity (m, n) is defined inductively by the rules

$$\mathsf{nil} : (0,0) \qquad \langle x \rangle : (0,1)$$

$$\frac{P : (m,n) \qquad Q : (k,l)}{P \mid Q : (m+k, n+l)} \qquad \frac{P : (m,k) \qquad Q : (k,n)}{P \cdot Q : (m,n)}$$

$$\frac{P : (m,n)}{(x)P : (m+1,n)} \qquad \frac{P : (m,n)}{(\nu x)P : (m,n)}$$

$$\frac{P : (m,0)}{x.P : (0,0)} \qquad \frac{P : (0,m)}{\overline{x}.P : (0,0)}$$

Definition 3
The *structural congruence* between π-processes of the same arity, written $\equiv$, is the smallest congruence satisfying the axioms given in figure 1, and which is closed with respect to $\_ \mid \_$, $\_ \cdot \_$, $(x)\_$, $(\nu x)\_$, $x.\_$ and $\overline{x}.\_$.

Notice the side-condition associated with the axiom for the commutativity of parallel composition. It results in the order of abstractions and datums always being preserved, as one would expect. The other rules are self-explanatory. The application axioms are enough to show that application can be derived from the other operators.

One property of the structural congruence is that every π-process is congruent to a *standard form*. Standard forms are processes with the shape

$$(\vec{x})(\nu \vec{z})(\langle \vec{y} \rangle \mid P),$$

Standard axioms for nil, $\_|\_$ and $(\nu x)\_$:

$$\begin{aligned} P \mid \mathsf{nil} &\equiv P \\ P \mid \mathsf{nil} &\equiv P \\ (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \\ P \mid Q &\equiv Q \mid P, \quad P : (m,0), Q : (0,n) \end{aligned}$$

$$\begin{aligned} (\nu x)(\nu y)P &\equiv (\nu y)(\nu x)P \\ (\nu x)(P \mid Q) &\equiv (\nu x)P \mid Q, \quad x \notin \mathsf{fn}(Q) \\ (\nu x)(P \mid Q) &\equiv P \mid (\nu x)Q, \quad x \notin \mathsf{fn}(P) \\ (\nu x)P &\equiv (\nu y)P\{y/x\}, \quad y \notin \mathsf{fn}(P) \\ (\nu x)\mathsf{nil} &\equiv \mathsf{nil} \end{aligned}$$

Abstraction axioms:

$$\begin{aligned} (x)P &\equiv (y)P\{y/x\}, \quad y \notin \mathsf{fn}(P) \\ (\nu x)(y)P &\equiv (y)(\nu x)P, \quad y \not\equiv x \\ (x)(P \mid Q) &\equiv (x)P \mid Q, \quad x \notin \mathsf{fn}(Q) \\ (x)(P \mid Q) &\equiv P \mid (x)Q, \quad x \notin \mathsf{fn}(P), P : (0,m) \end{aligned}$$

Application axioms:

$$\begin{aligned} (\nu x)(P \cdot Q) &\equiv (\nu x)P \cdot Q, \quad x \notin \mathsf{fn}(Q) \\ (\nu x)(P \cdot Q) &\equiv P \cdot (\nu x)Q, \quad x \notin \mathsf{fn}(P) \\ (x)P \cdot Q &\equiv (x)(P \cdot Q), \quad x \notin \mathsf{fn}(Q) \\ (\langle y \rangle \mid Q) \cdot (x)P &\equiv Q \cdot P\{y/x\} \\ Q \cdot P &\equiv Q \mid P, \quad Q : (m,0), P : (0,n) \end{aligned}$$

Fig. 1. The structural congruence between π-processes of the same arity, written $\equiv$, is the smallest equivalence relation satisfying these axioms and closed with respect to contexts.

where the $\vec{x}$s and $\vec{z}$s are distinct, the $\vec{z}$s are contained in the $\vec{y}$s, and $P : (0,0)$ does not contain any applications. Standard forms are essentially unique in the sense that, given two congruent standard forms

$$(\vec{x}_1)(\nu\vec{z}_1)(\langle\vec{y}_1\rangle \mid P_1) \equiv (\vec{x}_2)(\nu\vec{z}_2)(\langle\vec{y}_2\rangle \mid P_2),$$

then $|\vec{x}_1| = |\vec{x}_2|$ and $|\vec{z}_1| = |\vec{z}_2|$, the $\vec{y}_1$ and $\vec{y}_2$ are the same up to α-conversion of the $\vec{x}$s and $\vec{z}$s, and P_1 and P_2 are structurally congruent again up to α-conversion. Using the standard forms, application can be derived from the other operators.

The *reaction relation* between π-processes of the same arity, written $\searrow$, is the smallest relation generated by

$$\overline{x}.P \mid x.Q \searrow P \cdot Q,$$

where P and Q have arities $(0, m)$ and $(m, 0)$ respectively, and the relation is closed with respect to $\_ \mid \_$, $\_ \cdot \_$, $(x)\_$, $(\nu x)\_$ and $\_ \equiv \_$.

In [Mil99], Milner defines a labelled transition system and the corresponding strong bisimulation for the π-calculus, using standard transitions with labels x, $\overline{x}$ and τ. It is straightforward to adapt his definitions to our presentation.

2.2 Action Calculi

An action calculus is generated from the basic process operators of the π-calculus, except restriction, plus control operators specific to the particular action calculus under consideration. However, the definition of the structural congruence cannot be simply lifted from the corresponding definition for the π-calculus. In the π-calculus, application is only used to apply datums to abstractions. In the framework, application is also used to apply control processes to other processes. This additional use of application requires a more general description of its properties. For example, the associativity of application can be derived in the π-calculus, but must be declared explicitly in the framework.

The simplest way to define the structural congruence for the framework is to adapt the basic constructs of the π-calculus to include the *identity* process id_m and the *permutation* process $\mathsf{p}_{m,n}$, where m and n are arities. These constructs can be derived from the other operators: for example, we have $\mathsf{id}_1 \equiv (x)\langle x \rangle$ and $\mathsf{p}_{1,1} \equiv (x, y)\langle y, x \rangle$ where $y \not\equiv x$. We choose to regard them as primitive, since they play a fundamental role in the congruence definition.

An action calculus is specified by a set $\mathcal{K}$ of controls, plus a reaction relation which describes the interaction between control processes. Each control K in $\mathcal{K}$ has an associated arity $((m_1, n_1), \ldots, (m_r, n_r)) \to (k, l)$, which informs us that a control process $K(P_1, \ldots, P_r)$ has arity (k, l) such that P_i has arity (m_i, n_i).

Definition 4
The set $\mathcal{P}'_{\mathsf{AC}}(\mathcal{K})$ of *pre-processes* of an action calculus specified by control set $\mathcal{K}$ is defined by the grammar

P	::=	id_m	Identity
		$\mathsf{p}_{m,n}$	Permutation
		$P \mid P$	Parallel composition
		$P \cdot P$	Application
		$\langle x \rangle$	Datum
		$(x)P$	Abstraction
		$K(P_1, \ldots, P_r)$	Control process

The set $\mathcal{P}_{\mathsf{AC}}(\mathcal{K})$ of *action processes* of arity (m, n), specified by control set $\mathcal{K}$, is defined by the identity and permutation axioms

$$\mathsf{id}_m : (m, m) \qquad \mathsf{p}_{m,n} : (m + n, n + m),$$

the appropriate rules in definition 2, and the control rule

$$\frac{P_i : (m_i, n_i) \quad i \in \{1, \ldots, r\}}{K(P_1, \ldots, P_r) : (k, l)}$$

where control K has arity $((m_1, n_1), \ldots, (m_r, n_r)) \to (k, l)$.

The axioms generating the structural congruence for the framework are a modification of the axioms for the π-calculus given in figure 1 of section 2.1, minus those referring to restriction. The best way to describe the axioms is to focus on the categorical structure of processes. An action process $P : (m, n)$ can be viewed as a morphism in a category with domain m and codomain n. The id_m operator corresponds to the identity of the category, application to sequential composition, parallel to tensor, and $\mathsf{p}_{m,n}$ to permutation. The structural congruence on action processes is defined in figure 2. It is generated by the axioms of a strict symmetric monoidal category and two additional naming axioms.

The axioms of a strict symmetric monoidal category just state that the identity, parallel composition, application and permutation operators behave as we would expect. A possibility helpful intuition is that the processes id_0 and $\mathsf{p}_{0,0}$ correspond to the nil process and

$$\begin{aligned} \mathsf{id}_m &\equiv (\vec{x})\langle\vec{x}\rangle, \quad \vec{x} \text{ distinct }, |\vec{x}| = m > 0 \\ \mathsf{p}_{m,n} &\equiv (\vec{x}, \vec{y})\langle\vec{y}, \vec{x}\rangle, \quad \vec{x}, \vec{y} \text{ distinct }, |\vec{x}| = m, |\vec{y}| = n, \quad m + n > 0. \end{aligned} \tag{1}$$

For example, the last axiom generalises the commutativity of parallel composition for the π-calculus. The first naming axiom is just a special case of the application axiom for the π-calculus given in figure 1, which describes the application of datums to abstractions. The second naming axiom is a generalisation of the structural congruence given in (1) for the identity process. The appropriate π-axioms in figure 1 of section 2.1 are derivable from the axioms given here. With the above interpretation of the identity and permutation processes, the axioms of the framework are also derivable in the π-calculus.

The *reaction relation* $\searrow$ is a binary relation between action processes of the same arity, which is closed with respect to $\_|\,\_$, $\_ \cdot \_$, $(x)\_$ and $\_ \equiv \_$. We specify whether reaction may occur inside controls. Both examples studied in this section prevent it.

We have a very different type of standard form result for action calculi, since application plays such a prominent role. The standard forms are not special processes, but rather have a completely different notation to processes. We therefore call them *molecular forms*, to emphasise this difference and conform with the literature [Mil96]. The molecular forms are generated by the grammars

$$\begin{aligned} \text{Molecular forms} \quad P &::= (\vec{x})[M, \ldots, M]\langle\vec{y}\rangle, \qquad \vec{x} \text{ distinct} \\ \text{Molecules} \quad M &::= \langle\vec{u}\rangle K(P_1, \ldots, P_r)(\vec{v}), \quad \vec{v} \text{ distinct} \end{aligned}$$

with r, $|\vec{u}|$ and $|\vec{v}|$ determined by the arity of control K. The $\vec{x}$ and $\vec{y}$ correspond to the abstractions and datums in the standard forms for the π-calculus given

Axioms of a strict symmetric monoidal category:

$$\begin{array}{rcl} P \cdot \mathsf{id}_n & \equiv & P \equiv \mathsf{id}_m \cdot P \\ P \cdot (Q \cdot R) & \equiv & (P \cdot Q) \cdot R \\ (P_1 \cdot P_2) \,|\, (Q_1 \cdot Q_2) & \equiv & (P_1 \,|\, Q_1) \cdot (P_2 \mid Q_2) \end{array} \qquad \begin{array}{rcl} P \mid \mathsf{id}_0 & \equiv & P \equiv \mathsf{id}_0 \mid P \\ (P \,|\, Q) \,|\, R & \equiv & P \,|\, (Q \,|\, R) \\ \mathsf{id}_m \mid \mathsf{id}_n & \equiv & \mathsf{id}_{m+n} \end{array}$$

$$\begin{array}{rcl} \mathsf{p}_{m,n} \cdot \mathsf{p}_{n,m} & \equiv & \mathsf{id}_{m+n} \\ \mathsf{p}_{m+n,k} & \equiv & (\mathsf{id}_m \mid \mathsf{p}_{n,k}) \cdot (\mathsf{p}_{m,k} \mid \mathsf{id}_n) \\ \mathsf{p}_{m,n} \cdot (P \,|\, Q) & \equiv & (Q \,|\, P) \cdot \mathsf{p}_{k,l} \,, \quad P : (m,l), Q : (n,k) \end{array}$$

Naming axioms:

$$\begin{array}{rcl} (\langle y \rangle \,|\, \mathsf{id}_m) \cdot (x)P & \equiv & P\{y/x\} \\ (x)((\langle x \rangle \,|\, \mathsf{id}_m) \cdot P) & \equiv & P, \quad x \notin \mathsf{fn}(P) \end{array}$$

Fig. 2. The structural congruence between action processes of the same arity, written $\equiv$, is the smallest equivalence relation satisfying these axioms and closed with respect to contexts.

in section 2.1, where $(|\vec{x}|, |\vec{y}|)$ is the arity of the molecular form. The $\vec{u}$ are free, and the $\vec{v}$ are distinct and bind to the right. These $\vec{v}$ provide the mechanism for recording the sequential dependency of control processes. The molecular forms have a simple structural congruence, given by α-conversion of bound names and the partial reordering of the molecules when there is no name clash.

The π Action Calculus The π action calculus is specified by the controls

$$\mathsf{out} : ((0,m)) \to (1,0) \qquad \mathsf{in} : ((m,0)) \to (1,0) \qquad \mathsf{new} : (\,) \to (0,1)$$

The input and output processes correspond to the action processes of the form $\langle x \rangle \cdot \mathsf{in}(P)$ and $\langle x \rangle \cdot \mathsf{out}(P)$ respectively. Restriction is interpreted by action processes of the form $\mathsf{new} \cdot (x)P$, where the new control blocks the access to the abstraction. The reaction relation is generated by the rule

$$\langle x \rangle \cdot \mathsf{out}(P) \,|\, \langle x \rangle \cdot \mathsf{in}(Q) \quad \searrow \quad P \cdot Q$$

where P and Q must have arities $(m,0)$ and $(0,n)$ respectively and we specify that reaction does not occur inside the controls.

The correspondence is not absolutely exact, since the π-axiom

$$(\nu x)\mathsf{nil} \equiv \mathsf{nil}$$

is not preserved by the translation. However, the translated processes are bisimilar. Apart from this point, it is not difficult to show that the structural congruence and the reaction relation are strongly preserved by the embedding [Mil96]. These results rely on the molecular forms described earlier.

The λ_v Action Calculus The λ_v action calculus interprets λ-terms as processes of arity $(0,1)$. It is specified by the controls

$$\mathsf{lam} : ((1,1)) \to (0,1) \qquad \mathsf{ap} : (\,) \to (2,1).$$

For example, the λ-term $\lambda x.x$ corresponds to the action process $\mathsf{lam}((x)\langle x\rangle)$, which illustrates the use of abstracted datums. Again we do not permit reaction inside a lam-control. The reaction relation is generated by the two rules

$$\begin{array}{rcl} (\mathsf{lam}(P) \,|\, \mathsf{id}_1) \cdot \mathsf{ap} & \searrow & P \\ (\mathsf{lam}(P) \,|\, \mathsf{id}_m).(x)Q & \searrow & Q\{\mathsf{lam}(P)/\langle x\rangle\}. \end{array}$$

The first axiom describes β-reduction, and the second describes the explicit substitution of a lam-process for abstracted datums. In fact, there is a higher-order extension of action calculi where the lam- and ap-controls are regarded as primitive constructs. The details can be found in [Mil94a,HG97].

There is another extension of the action calculus framework [Mil94b], which includes a reflexion (or feedback) operator

$$\frac{P : (m+1, n+1)}{\uparrow P : (m,n)}$$

The reflexion operator does not have a direct analogy in the π-calculus. It does have a direct analogy in the λ-calculus, in that the λ_v action calculus plus reflexion corresponds to the cyclic λ-calculus of Ariola and Klop [Has97]. The reflexion axioms correspond to adding a trace operator [JSV96] to the symmetric monoidal category. We do not concentrate on the reflexive extension here, since our aim is to illustrate the close connection between the π-calculus and the action calculus framework. It is however of fundamental importance. In particular, the recent bisimulation results of Leifer and Milner rely on this extension.

Leifer and Milner are currently exploring general bisimulation results for (simple) reactive systems [LM00], and aim to extend their results to a class of reflexive action calculi. Such results have also been studied by Sewell [Sew00]. The idea is to automatically generate a labelled transition system from an underlying reaction relation, and yield a corresponding bisimulation congruence. The labelled transitions have the form $P \xrightarrow{C} Q$, with the intuition that C is a small context necessary to complete a redex in P. Leifer and Milner have characterised what it means for these small contexts to exist. Their challenge is to show that they do indeed exist for specific reactive systems such as action calculi.

3 The π_F-Calculus

In this section, we describe the π_F-calculus and define a natural bisimulation congruence. The π_F-calculus is similar in many respects to the fusion calculus of Parrow and Victor [PV98], and the χ-calculus of Fu [Fu97], although we did not invent the π_F-calculus with these connections in mind. There is no

abstraction. Instead, the π_F-calculus has symmetric input and output processes, with a reaction relation which fuses the names together using *explicit fusions*. An explicit fusion $\langle z = y \rangle$ is a process which declares that two names can be used interchangeably.

The key difference between the π_F-calculus and the fusion calculus is how the name-fusions have effect. In the π_F-calculus, fusions are explicitly recorded and have effect though the structural congruence. For example, a typical π_F-reaction is

$$x.\langle z \rangle P \mid \overline{x}.\langle y \rangle Q \mid R \searrow \langle z = y \rangle \mid P \mid Q \mid R.$$

The explicit fusion $\langle z = y \rangle$ declares that z and y can be used interchangeably throughout the process. The reaction on the channel x is a local one between the input and output processes. However, its effect is global in that the scope of $\langle z = y \rangle$ affects process R. The scope of an explicit fusion is limited by the restriction operator. In the fusion calculus on the other hand, fusions occur implicitly within the reaction relation. For example, the fusion reaction

$$(\nu y)(x.\langle z \rangle P \mid \overline{x}.\langle y \rangle Q \mid R) \searrow P\{z/y\} \mid Q\{z/y\} \mid R\{z/y\}$$

corresponds to the restriction (with respect to y) of the π_F-reaction given above. This fusion reaction is not a simple local reaction between input and output processes. Instead, the redex is parametrized by R and requires y (or z) to be restricted. The full definition, using many $\vec{y}$s and $\vec{z}$s, is in fact quite complicated.

Definition 5
The set $\mathcal{P}'_{\pi_F}$ of pre-processes of the π_F-calculus is defined by the grammar

$P ::=$	nil	Nil process
	$P \mid P$	Parallel composition
	$P@P$	Connection
	$\langle x \rangle$	Datum
	$\langle x = y \rangle$	Fusion
	$(\nu x)P$	Restriction
	$x.P$	Input Process
	$\overline{x}.P$	Output process

The definitions of *free* and *bound* names are standard. The *restriction* operator $(\nu x)P$ binds x; otherwise x is free. The connection operator is used to connect the contents of input and output processes during reaction:

$$x.P \mid \overline{x}.Q \searrow P@Q.$$

We regard the connection operator as primitive. However, we shall see that it can be derived from the other axioms. It can also be derived in the fusion system framework, so our choice to include it is really not essential.

To define reaction properly, we require the correct number of datums in P and Q to connect together. This information is given by the *arity* of a pre-process. We write $P : m$ to declare that a pre-process has arity m, where m is a natural number used to count datums.

Definition 6
The set $\mathcal{P}_{\pi_F}$ of *processes* of the π_F-calculus with arity m is defined inductively by the rules

$$\mathsf{nil} : 0 \qquad \langle x = y \rangle : 0 \qquad \langle x \rangle : 1$$

$$\frac{P : m \qquad Q : n}{P \mid Q : m + n} \qquad \frac{P : m \qquad Q : m}{P @ Q : 0} \qquad \frac{P : m}{(\nu x)P : m}$$

$$\frac{P : m}{x.P : 0} \qquad \frac{P : m}{\overline{x}.P : 0}$$

Definition 7
The *structural congruence* between processes, written $\equiv$, is the smallest congruence satisfying the axioms given in figure 3, and which is closed with respect to $\_ \mid \_$, $\_ @ \_$, $(\nu x)\_$, $x.\_$ and $\overline{x}.\_$.

The side-condition on the commutativity of parallel composition allows for processes of arity 0 to be reordered, but not arbitrary processes. The connection axioms are simple.

The fusion axioms are similar in spirit to the name-equalities introduced by Honda in his work on a simple process framework [Hon00]. Our intuition is that $\langle x = y \rangle$ is a symmetric relation which declares that two names can be used interchangeably. The fusion $\langle x = x \rangle$ is congruent to the nil process. So too is $(\nu x)\langle x = y \rangle$, since the local name is unused. Notice that the standard axiom $(\nu x)\mathsf{nil} \equiv \mathsf{nil}$ is derivable from these two fusion axioms. The other six fusion axioms describe small-step substitution, allowing us to deduce $\langle x = y \rangle \mid P \equiv \langle x = y \rangle \mid P\{x/y\}$ as well as α-conversion. For example,

	$(\nu x)(\overline{x}.\mathsf{nil})$	
$\equiv$	$(\nu x)(\nu y)(\langle x = y \rangle \mid \overline{x}.\mathsf{nil})$	create fresh local name y as an alias for x
$\equiv$	$(\nu x)(\nu y)(\langle x = y \rangle \mid \overline{y}.\mathsf{nil})$	substitute y for x
$\equiv$	$(\nu y)(\overline{y}.\mathsf{nil})$	remove the now-unused local name x

Just as for the π-calculus, a property of the structural congruence is that every π_F-process is structurally congruent to a *standard form*. Standard forms are processes with the shape

$$(\nu \vec{x})(\langle \vec{y} \rangle \mid P) \mid \langle \vec{u} = \vec{v} \rangle,$$

where the $\vec{x}$s are distinct and contained in the $\vec{y}$s, and $P : 0$ does not contain connections or fusions. The standard form is essentially unique in the sense that, given two congruent standard forms

$$(\nu \vec{x}_1)(\langle \vec{y}_1 \rangle \mid P_1) \mid \langle \vec{u}_1 = \vec{v}_1 \rangle \equiv (\nu \vec{x}_2)(\langle \vec{y}_2 \rangle \mid P_2) \mid \langle \vec{u}_2 = \vec{v}_2 \rangle,$$

then $|\vec{x}_1| = |\vec{x}_2|$, the fusions $\langle \vec{u}_1 = \vec{v}_1 \rangle$ and $\langle \vec{u}_2 = \vec{v}_2 \rangle$ generate the same equivalence relation on names, the $\vec{y}_1$ and $\vec{y}_2$ are the same up to α-conversion of the $\vec{x}_1$s

Standard axioms for $\_|\_$, $(\nu x)\_$ and nil:

$$\begin{aligned} P \mid \mathsf{nil} &\equiv P \\ (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \\ P \mid Q &\equiv Q \mid P, \quad P : 0 \end{aligned}$$

$$\begin{aligned} (\nu x)(\nu y)P &\equiv (\nu y)(\nu x)P \\ (\nu x)(P \mid Q) &\equiv (\nu x)P \mid Q, \quad x \notin \mathsf{fn}(Q) \\ (\nu x)(P \mid Q) &\equiv P \mid (\nu x)Q, \quad x \notin \mathsf{fn}(P) \end{aligned}$$

Connection axioms:

$$\begin{aligned} (\nu x)(P@Q) &\equiv (\nu x)P@Q, \quad x \notin \mathsf{fn}(Q) \\ (\nu x)(P@Q) &\equiv P@(\nu x)Q, \quad x \notin \mathsf{fn}(P) \\ (\langle x \rangle \mid P)@(\langle y \rangle \mid Q) &\equiv \langle x = y \rangle \mid (P@Q) \\ P@Q &\equiv P \mid Q, \quad P, Q : 0 \end{aligned}$$

Fusion axioms:

$$\begin{aligned} \langle x = x \rangle &\equiv \mathsf{nil} & \langle x = y \rangle \mid y.P &\equiv \langle x = y \rangle \mid x.P \\ \langle x = y \rangle &\equiv \langle y = x \rangle & \langle x = y \rangle \mid \overline{y}.P &\equiv \langle x = y \rangle \mid \overline{x}.P \\ (\nu x)\langle x = y \rangle &\equiv \mathsf{nil} & \langle x = y \rangle \mid \langle y = z \rangle &\equiv \langle x = y \rangle \mid \langle x = z \rangle \\ & & \langle x = y \rangle \mid \langle y \rangle &\equiv \langle x = y \rangle \mid \langle x \rangle \\ & & \langle x = y \rangle \mid z.P &\equiv \langle x = y \rangle \mid z.(\langle x = y \rangle \mid P) \\ & & \langle x = y \rangle \mid \overline{z}.P &\equiv \langle x = y \rangle \mid \overline{z}.(\langle x = y \rangle \mid P) \end{aligned}$$

Fig. 3. The structural congruence between π_F-processes of the same arity, written $\equiv$, is the smallest equivalence relation satisfying these axioms and closed with respect to contexts.

and $\vec{x}_2$s, and up to the name-equivalence generated by fusions, and P_1 and P_2 are structurally congruent up to α-conversion and the name-equivalence. We write $E(P)$ for the smallest equivalence relation on names generated by P. This equivalence relation can be inductively defined on the structure of processes; a simple characterisation is given by $(x, y) \in E(P)$ if and only if $P \equiv P \mid \langle x = y \rangle$.

Using the standard forms, it is possible to express the connection operator as a derived operator.

Definition 8
The *reaction relation* between π_F-processes of the same arity , written $\searrow$, is the smallest relation generated by

$$x.P \mid \overline{x}.Q \searrow P@Q,$$

where P and Q have arity m and the reaction is closed with respect to $\_|\_$, $\_@\_$, $(\nu x)\_$ and $\_ \equiv \_$.

3.1 Bisimulation Congruence

A natural choice of labelled transition system (LTS) for the π_F-calculus consists of the usual CCS-style transitions using labels $\overline{x}$, x and τ, accompanied by a definition of bisimulation which incorporates fusions: $PSQ : 0$ implies

$$\text{for all } x \text{ and } y, \text{ if } P \mid \langle x = y \rangle \xrightarrow{\alpha} P_1 \text{ then } Q \mid \langle x = y \rangle \xrightarrow{\alpha} Q_1 \text{ and } P_1 S Q_1.$$

We call this bisimulation the *open bisimulation*, since it is similar to the open bisimulation of the π-calculus.

In the definition of open bisimulation, labelled transitions are analysed with respect to the fusion contexts $\_ \mid \langle x = y \rangle$. In fact, we do not need to consider all such contexts. Instead, we introduce the *fusion transitions*, generated by the axiom

$$x.P \mid \overline{y}.Q \xrightarrow{?x=y} P@Q.$$

A fusion transition with label $?x = y$ declares the presence of input and output processes with channel names x and y, although we do not know which name goes with which process. The resulting bisimulation definition is simpler, in that it is enough to analyse the fusion transitions rather than consider all fusion contexts. However, these fusion transitions do seem to provide additional information about the structure of processes. In order to define a bisimulation relation which equals the open bisimulation, we remove this information in the analysis of the labelled transitions: $PSQ : 0$ implies

$$\text{if } P \xrightarrow{?x=y} P_1 \text{ then either } Q \xrightarrow{?x=y} Q_1 \text{ or } Q \xrightarrow{\tau} Q_1, \text{ and } P_1 \mid \langle x = y \rangle \, S \, Q_1 \mid \langle x = y \rangle.$$

A consequence of adding fusion transitions is that we can use standard techniques to show that the resulting bisimulation relation is a congruence, and does indeed equal the open bisimulation.

In [GW00], we also study the more standard definition of bisimulation, which requires that fusion transitions match exactly. We know it yields a congruence which is contained in the open bisimulation. At the moment, we do not know whether the containment is strict. This question relates to an open problem for the π-calculus without replication and summation [1], of whether strong bisimulation is closed with respect to substitution.

The fusion LTS is given in figure 4. The labelled transitions follow the style of transition given in [Mil99], although our transitions are defined for arbitrary processes instead of processes of arity 0. This choice is not essential, since the bisimulation definition only refers to labelled transitions for processes of arity 0. The only additional complexity is that we have two rules for parallel composition, since this operator is only commutative for processes of arity 0. Notice that the structural congruence rule allows fusions to affect the labels: for example, the process $\langle x = y \rangle \mid \overline{x}.P$ can undergo the transition $\xrightarrow{\overline{y}}$ as well as $\xrightarrow{\overline{x}}$, because it is structurally congruent to $\langle x = y \rangle \mid \overline{y}.P$. Also notice that we do not have an explicit structural rule for the connection operator. Indeed, it is not possible to write such a rule, since the arity information would cause problems.

[1] Personal communication with Davide Sangiorgi.

$$x.P \xrightarrow{x} P \qquad \overline{x}.P \xrightarrow{\overline{x}} P$$

$$x.P \,|\, \overline{y}.Q \xrightarrow{?x=y} P@Q \qquad x.P \,|\, \overline{x}.Q \xrightarrow{\tau} P@Q$$

$$\frac{P \xrightarrow{\alpha} P_1}{P \,|\, Q \xrightarrow{\alpha} P_1 \,|\, Q} \qquad \frac{Q \xrightarrow{\alpha} Q_1}{P \,|\, Q \xrightarrow{\alpha} P \,|\, Q_1}$$

$$\frac{P \xrightarrow{\alpha} Q, \quad x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)Q} \qquad \frac{P \equiv P_1 \xrightarrow{\alpha} Q_1 \equiv Q}{P \xrightarrow{\alpha} Q}$$

Fig. 4. The labelled transition system for arbitrary π_F-processes. We do not distinguish between the labels $?x = y$ and $?y = x$.

Proposition 9
$P \searrow Q$ if and only if $P \xrightarrow{\tau} Q$.

The basic intuition regarding our definition of bisimulation is that two processes are bisimilar if and only if their standard forms have the same outer structure and, in all contexts of the form $\_@\langle\vec{y}\rangle$, if one process can do a labelled transition then so must the other to yield bisimilar processes. In fact, we do not need to consider all such contexts. Instead, it is enough to remove the top-level datums from the standard forms, and analyse the labelled transitions for the resulting processes.

Definition 10 (Fusion Bisimulation)
A symmetric relation S is a *fusion bisimulation* if and only if PSQ implies

1. P and Q have standard forms $(\nu\vec{x})(\langle\vec{y}\rangle \,|\, P_1) \,|\, \langle\vec{u} = \vec{v}\rangle$ and $(\nu\vec{x})(\langle\vec{y}\rangle \,|\, Q_1) \,|\, \langle\vec{u} = \vec{v}\rangle$ respectively;

2a. if $P_1 \,|\, \langle\vec{u} = \vec{v}\rangle \xrightarrow{\alpha} P_1'$ where α is $x, \overline{x}$ or τ then $Q_1 \,|\, \langle\vec{u} = \vec{v}\rangle \xrightarrow{\alpha} Q_1'$ and $P_1' S Q_1'$;

2b. if $P_1 \,|\, \langle\vec{u} = \vec{v}\rangle \xrightarrow{?x=y} P_1'$ then either $Q_1 \,|\, \langle\vec{u} = \vec{v}\rangle \xrightarrow{?x=y} Q_1'$ or $Q_1 \,|\, \langle\vec{u} = \vec{v}\rangle \xrightarrow{\tau} Q_1'$ and $P_1' \,|\, \langle x = y\rangle \, S \, Q_1' \,|\, \langle x = y\rangle$;

3. similarly for Q.

Two processes P and Q are *fusion bisimilar*, written $P \sim_f Q$, if and only if there exists a fusion bisimulation S between them. The relation $\sim_f$ is the largest fusion bisimulation. We call it *the* fusion bisimulation, when the meaning is apparent.

This definition of fusion bisimulation is related to Sangiorgi's efficient characterisation of open bisimulation for the π-calculus with matching [San93].

The fusion transitions enable us to use standard techniques for proving that the fusion bisimulation is a congruence. We remove the structural congruence rule, and define an alternative LTS based on the structure of processes. This alternative LTS is given in figure 5. The non-standard rules are the fusion rules. The first two play a similar role to the τ-rules for the π-calculus. The other two

$$x.P \xrightarrow{x} P \qquad \overline{x}.P \xrightarrow{\overline{x}} P$$

$$\frac{P \xrightarrow{x} P_1 \quad Q \xrightarrow{\overline{y}} Q_1}{P \,|\, Q \xrightarrow{?x=y} P_1 @ Q_1} \qquad \frac{P \xrightarrow{\overline{x}} P_1 \quad Q \xrightarrow{y} Q_1}{P \,|\, Q \xrightarrow{?x=y} P_1 @ Q_1}$$

$$\frac{P \xrightarrow{\alpha} Q \quad (x,y) \in E(P)}{P \xrightarrow{\alpha[y/x]} Q} \qquad \frac{P \xrightarrow{?x=x} Q}{P \xrightarrow{\tau} Q}$$

$$\frac{P \xrightarrow{\alpha} P_1}{P \,|\, Q \xrightarrow{\alpha} P_1 \,|\, Q} \qquad \frac{Q \xrightarrow{\alpha} Q_1}{P \,|\, Q \xrightarrow{\alpha} P \,|\, Q_1}$$

$$\frac{P \xrightarrow{\alpha} Q, \quad x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)Q}$$

Fig. 5. The alternative LTS for arbitrary processes. The judgement $\alpha[y/x]$ denotes the simple substitution of one y for one x, generated by: $x[y/x] = y$; $\overline{x}[y/x] = \overline{y}$; $(?x = z)[y/x] = ?y = z$, and $\alpha[y/x] = \alpha$ when $x \notin \alpha$. Recall that $E(P)$ is the smallest equivalence relation on names generated by P.

express the effect of explicit fusions on labels and the generation of τ-transitions from simple fusion transitions. Notice that we have no rules for the connection operator. The connection between the fusion LTS and the alternative LTS is therefore only valid up to structural congruence.

Using the alternative LTS, we are able to define a bisimulation relation, prove that it is a congruence and show that it equals the fusion bisimulation. The details are given in [GW00]. With this congruence result, it is not difficult to show that fusion bisimulation equals open bisimulation.

4 Fusion Systems

The fusion system framework consists of the same basic process constructs as the π_F-calculus. It generalises the input and output processes to more general *control processes*. For action calculi, control processes have the form $K(P_1, \ldots, P_r)$. It is possible to use the same approach for fusion systems. Instead, we choose to focus on control processes of the form $\vec{x}.K(P_1, \ldots, P_r)$, where the control K is a boundary containing the processes P_i and the names $\vec{x}$ provide the interface to the control. With this choice of control process, the connection operator can be derived from the other operators.

A fusion calculus is specified by a set $\mathcal{K}$ of controls, and a reaction relation describing the interaction between controls. Each control K in $\mathcal{K}$ has an asso-

ciated arity $(m_1, \ldots, m_r) \to k$. The arity tells us how to construct a control process $\vec{x}.K(P_1, \ldots, P_r)$, where the m_i specify the arities of the P_i and $|\vec{x}| = k$.

Definition 11
The set $\mathcal{P}'_{\mathsf{F}}(\mathcal{K})$ of *pre-processes* of a fusion system specified by control set $\mathcal{K}$ is defined by the grammar

P	::=	nil	Nil process
		$P \mid P$	Parallel composition
		$P@P$	Connection
		$\langle x \rangle$	Datum
		$\langle x = y \rangle$	Fusion
		$(\nu x)P$	Restriction
		$\vec{x}.K(P_1, \ldots, P_r)$	Control process

The set $\mathcal{P}_{\mathsf{F}}(\mathcal{K})$ of *fusion processes* with arity m is defined by the rules in definition 6, with the input and output rules generalised to the control rule

$$\frac{P_i : m_i \quad i \in \{1, \ldots, r\}}{\vec{x}.K(P_1, \ldots, P_r) : 0}$$

where control K has arity $(m_1, \ldots, m_r) \to k$ and $|\vec{x}| = k$.

The definitions of *free* and *bound* names are standard. The definition of the *structural congruence* between fusion processes is the same as that given for the π_F-calculus in figure 3, except that it is closed with respect to arbitrary control processes. The *standard forms* are defined in the same way as those for the π_F-calculus given in section 3. We can therefore derive the connection operator from the other operators. The *reaction relation* $\searrow$ is a binary relation between fusion processes of the same arity, which is closed with respect to $\_\,|\,$, $\_@\_$, $(\nu x)\_$ and $\_ \equiv \_$. We are able to specify whether the reaction relation is closed with respect to the controls.

The π Fusion System We define the π fusion system, which is just a reformulation of the π_F-calculus. It is specified by the controls

$$\mathsf{in} : (m) \to 1 \qquad \mathsf{out} : (m) \to 1$$

where the π_F-processes $x.P$ and $\overline{x}.P$ correspond to the control processes of the form $x.\mathsf{in}(P)$ and $x.\mathsf{out}(P)$ respectively. The reaction relation is generated by the rule

$$x.\mathsf{in}(P) \mid x.\mathsf{out}(Q) \searrow P@Q,$$

where P and Q have the same arity and reaction does not occur inside the controls. Since the correspondence with the π_F-calculus is exact, the bisimulation results described in section 3.1 easily transfer to the π_F fusion system.

We summarise the embedding results of the π-calculus and the fusion calculus in the π_F fusion system. These results follow immediately from the analogous embeddings in the π_F-calculus [GW00]. The translations of the π-calculus and the fusion calculus into π_F fusion system are characterised by the input and output cases described above, plus the abstraction and concretion cases:

$$\begin{array}{rcll} [\![(\vec{x})P]\!] & = & (\nu\vec{x})(\langle\vec{x}\rangle \mid [\![P]\!]) & \text{Abstraction} \\ [\![(\nu\vec{x})\langle\vec{z}\rangle P]\!] & = & (\nu\vec{x})(\langle\vec{z}\rangle \mid [\![P]\!]) & \text{Concretion} \end{array}$$

There is a key difference between the embedding of the π-calculus and the embedding of the fusion calculus. For the π-embedding, reaction of a process in the image of $[\![\_]\!]$ necessary results in a process congruent to one in the image. The same is not true with the embedding of the fusion calculus, since for example

$$x.\mathsf{in}(\langle z\rangle \mid [\![P]\!]) \mid x.\mathsf{out}(\langle y\rangle \mid [\![Q]\!]) \quad \searrow \quad \langle z = y\rangle \mid [\![P]\!] \mid [\![Q]\!],$$

which does not have a corresponding reaction in the fusion calculus. The process on the left is in the image of the fusion calculus under $[\![\_]\!]$, but the one on the right has an unbound explicit fusion and so is not congruent to anything in the image. We do obtain an embedding result, in that by restricting y (or z) we obtain the π_F fusion reaction

$$\begin{array}{rcl} (\nu y)(x.\mathsf{in}(\langle z\rangle \mid [\![P]\!]) \mid x.\mathsf{out}(\langle y\rangle \mid [\![Q]\!])) & \searrow & (\nu y)(\langle z = y\rangle \mid [\![P]\!] \mid [\![Q]\!]) \\ & \equiv & [\![P]\!]\{z/y\} \mid [\![Q]\!]\{z/y\}, \end{array}$$

which does indeed correspond to a fusion reaction. The full translations and embedding results are given for the π_F-calculus in [GW00].

The λ_v Fusion System We define the λ_v fusion system, which corresponds to a call-by-value λ-calculus. It is also possible to define fusion systems for the usual untyped λ-calculus, and the simply-typed versions by adding more structure to the arities. These fusion systems require the ability to replicate processes, which we have not used before. We are currently checking that our bisimulation results in section 3.1 hold in the presence of replication. We do not envisage difficulties.

The λ_v fusion system is specified by the controls

$$\mathsf{lam} : (2) \to 1 \qquad \mathsf{ap} : (\,) \to 3.$$

The idea is that the untyped λ-terms correspond to processes of arity 1. A lam-process of the form $u.\mathsf{lam}(P)$ 'locates' the function at u, with the extra arity for the process P corresponding to the abstraction of the λ-term. An ap-control $uvw.\mathsf{ap}$ 'locates' its function at u, its argument at v and its result at w. The reaction relation is generated by

$$u.\mathsf{lam}(P) \mid uvw.\mathsf{ap} \searrow P@\langle vw\rangle.$$

and reaction does not occur inside the lam-control. The translation from lambda terms to fusion processes is defined inductively by

$$\begin{array}{rcll} [\![x]\!] & \longmapsto & \langle x\rangle & \\ [\![\lambda x.t]\!] & \longmapsto & (\nu u)(\langle u\rangle \mid !u.\mathsf{lam}((\nu x)(\langle x\rangle \mid [\![t]\!]))) & u \notin \mathsf{fn}([\![\lambda x.t]\!]) \\ [\![st]\!] & \longmapsto & (\nu uvw)(\langle w\rangle \mid [\![s]\!]@\langle u\rangle \mid [\![t]\!]@\langle v\rangle \mid uvw.\mathsf{ap}) & u,v,w \notin \mathsf{fn}([\![st]\!]) \end{array}$$

The replication in the translation of the lambda abstraction is used to generate copies of lambda processes via the structural congruence.

The results for the call-by-value λ-calculus are not as straightforward as the results for the π_F-example. To illustrate this, consider the lambda reaction $(\lambda x.x)y \searrow y$ which corresponds to the fusion system reaction

$$(\nu uw)(\langle w\rangle \mid !u.((\nu x)\langle xx\rangle) \mid \overline{u}.(\langle yw\rangle)) \searrow \langle y\rangle \mid (\nu u)(!u.((\nu x)\langle xx\rangle)).$$

After reaction, the process $(\nu u)(!u.((\nu x)\langle xx\rangle))$ is redundant. This redundancy is expressed by a bisimulation congruence, with the process $(\nu u)(!u.((\nu x)\langle xx\rangle))$ bisimilar to nil. Bisimulation is thus required to relate the reaction relation of the λ-calculus with reaction in the λ_v fusion system.

Bisimulation for the λ_v fusion system is easy. It follows directly from bisimulation for the π_F fusion system. This is because the λ_v fusion system can be regarded as a simple subsystem of the π_F fusion system, using the translation

$$\begin{array}{rcl} [\![u.\mathsf{lam}(P)]\!] & \mapsto & u.\mathsf{in}([\![P]\!]) \\ [\![uvw.\mathsf{ap}]\!] & \mapsto & \overline{u}.\mathsf{out}(\langle vw\rangle) \end{array}$$

which trivially preserves the reaction relation. It is future work to show whether the resulting congruence corresponds to a known behavioural congruence for the call-by-value λ-calculus.

It also remains future work to study general bisimulation congruences for fusion systems. One option is to try to generalise the bisimulation results for the π_F and λ_v fusion systems. Another option is to adapt the techniques of Leifer and Milner, in their work on general bisimulation congruences for reactive systems [LM00]. However, this is not our primary concern. We would first like to explore specific examples of fusion systems to illustrate the expressive power of explicit fusions.

References

Fu97. Y. Fu. A proof-theoretical approach to communication. *ICALP*, LNCS 1256, Springer-Verlag, 1997. 70, 78

GW99. P. A. Gardner and L. Wischik. A process framework based on the π_F-calculus. *EXPRESS*, Elsevier Science Publishers, 1999. 69

GW00. P. A. Gardner and L. Wischik. Explicit fusions. *MFCS*, 2000. To appear. 69, 71, 82, 84, 86

Has97. M. Hasegawa. *Models of Sharing Graphs (A Categorical Semantics of Let and Letrec).* PhD thesis, ECS-LFCS-97-360, University of Edinburgh, 1997. 78

HG97. M. Hasegawa and P. Gardner. Higher order action calculi and the computational λ-calculus. *Theoretical Aspects of Computer Software*, Sendai, 1997. 78

Hon00. K. Honda. Elementary structures in process theory (1): sets with renaming. *Mathematical Structures in Computer Science*, 2000. To appear. 80

JSV96. A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3), 1996. 78

LM00. Jamey Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. *CONCUR*, 2000. To appear. 70, 78, 87

Mil94a. R. Milner. Higher order action calculi. *Computer Science Logic*, LNCS 832, Springer-Verlag, 1994. 78

Mil94b. R. Milner. Reflexive action calculi. Unpublished manuscript, 1994. 78

Mil96. R. Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996. 69, 76, 77

Mil99. R. Milner. *Communicating and mobile systems: the pi calculus*. CUP, 1999. 70, 71, 72, 75, 82

PV98. J. Parrow and B. Victor. The fusion calculus: expressiveness and symmetry in mobile processes. *LICS*, Computer Society Press, 1998. 70, 78

San93. D. Sangiorgi. A theory of bisimulation for the pi-calculus. *CONCUR*, Springer-Verlag, 1993. 83

Sew00. Peter Sewell. From rewrite rules to bisimulation congruences. *Theoretical Computer Science*, 2000. To appear. 78

Verification Using Tabled Logic Programming*

C. R. Ramakrishnan

Department of Computer Science, SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA
cram@cs.sunysb.edu

1 Abstract

The LMC project aims to advance the state of the art of system specification and verification using the latest developments in logic programming technology [CDD+98]. Initially, the project was focussed on developing an efficient model checker, called XMC [RRR+97], for value-passing CCS [Mil89] and the modal mu-calculus [Koz83] based on the XSB logic programming system [XSB00]. We developed an optimizing compiler to translate specifications in a dialect of value-passing CCS to compact labeled transition systems [DR99], improving verification performance several fold. The core principles of this translation have been recently incorporated in SPIN [Hol97] showing similar gains in performance [Hol99]. The XMC system can be downloaded from `http://www.cs.sunysb.edu/~lmc`.

More recently, we have developed

- techniques using logic-program transformations [TS84,PP99,RKRR99] for verifying parameterized systems, i.e., infinite families of finite-state systems [RKR+00];
- a proof-tree viewer for justifying successful or failed verification runs for branching-time properties [RRR00];
- a symbolic bisimulation checker (based on the work of [HL95]) for value-passing systems [MRRV00];
- model checkers for
  - real-time systems [DRS00] based loosely on the local model checking algorithm of [SS95];
  - LTL with actions [PR00] based on GCTL* of [BCG00] and the on-the-fly model checking algorithm in [BCG95];

In this tutorial, we describe the XMC system as well as the above developments. In addition, we outline the research efforts of the verification and the logic programming community that have been instrumental in these developments.

* Research supported in part by NSF grants EIA-9705998, CCR-9711386, CCR-9805735, and CCR-9876242.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 89–91, 2000.

References

BCG95. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *IEEE Symposium on Logic in Computer Science*. IEEE Press, 1995. 89

BCG00. G. Bhat, R. Cleaveland, and A. Groce. Efficient model checking via Büchi tableau automata. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000. 89

CDD+98. B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Static Analysis Symposium*. Springer Verlag, 1998. 89

DR99. Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE/PSTV '99*, 1999. 89

DRS00. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000. URL: `http://www.cs.sunysb.edu/~cram/papers`. 89

HL95. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995. 89

Hol97. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 89

Hol99. G. Holzmann. The engineering of a model checker: Gnu i-protocol case study revisited. In *6th SPIN Workshop on Practical Aspects of Model Checking*, 1999. 89

Koz83. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983. 89

Mil89. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 89

MRRV00. M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. Verma. Symbolic bisimulation using tabled constraint logic programming. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000. URL: `http://www.cs.sunysb.edu/~cram/papers`. 89

PP99. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 1999. 89

PR00. L. R. Pokorny and C. R. Ramakrishnan. Model checking linear temporal logic using tabled logic programming. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000. URL: `http://www.cs.sunysb.edu/~cram/papers`. 89

RKR+00. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic-program transformations. In *Proceedings of TACAS 2000*. Springer-Verlag, 2000. 89

RKRR99. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *Principles and Practice of Declarative Programming (PPDP)*, volume 1702 of *Lecture Notes in Computer Science*, pages 396–413, 1999. 89

RRR+97. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag. 89

RRR00. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, 2000. 89

SS95. O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification*. American Mathematical Society, 1995. 89

TS84. H. Tamaki and T. Sato. Fold/unfold transformation of logic programs. In *International Conference on Logic Programming*, pages 127–138, 1984. 89

XSB00. The XSB Group. The XSB logic programming system v2.1, 2000. Available from `http://www.cs.sunysb.edu/~sbprolog`. 89

Open Systems in Reactive Environments: Control and Synthesis*

Orna Kupferman[1], P. Madhusudan[2**], P. S. Thiagarajan[3], and Moshe Y. Vardi[4***]

[1] Hebrew University
orna@cs.huji.ac.il
[2] The Institute of Mathematical Sciences
madhu@imsc.ernet.in
[3] Chennai Mathematical Institute
pst@smi.ernet.in
[4] Rice University
vardi@cs.rice.edu

Abstract. We study the problems of synthesizing open systems as well as controllers for them. The key aspect of our model is that it caters to *reactive environments*, which can disable different sets of responses when reacting with the system. We deal with specifications given as formulas in CTL* and its sub-logic CTL. We show that both these problems, with specifications in CTL (CTL*), are 2EXPTIME-complete (resp. 3EXPTIME-complete). Thus, in a sense, reactive environments constitute a provably harder setting for the synthesis of open systems and controllers for them.

1 Introduction

A *closed* system is a system whose behavior is completely determined by its state. An *open* system is one that interacts with its environment and whose behavior crucially depends on this interaction [HP85]. In an open system, we are required to distinguish between output signals (generated by the system) over which we have control and input signals (generated by the environment) over which we have no control. Given a specification of an open system, say as a temporal logic formula, satisfiability of the formula only guarantees that we can synthesize a system for *some* environment, whereas we need to synthesize a system that meets the specification for *all* environments.

To make this intuition more precise, suppose we are given finite sets I and O of input and output signals. A *program* can be viewed as a *strategy*

* For a full version of this extended abstract, see Technical Report TCS-2000-03, Chennai Mathematical Institute, available at www.smi.ernet.in

** On leave visiting Informatik VII, RWTH-Aachen, Germany

*** Supported in part by NSF grant CCR-9700061, and by a grant from the Intel Corporation.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 92–107, 2000.

$f : (2^I)^* \rightarrow 2^O$ that maps finite sequences of input signal sets into an output signal set. When f interacts with an environment that generates infinite input sequences what results is an infinite computation over $2^{I \cup O}$. Though f is deterministic, it produces a computation tree. The branches of the tree correspond to external non-determinism caused by the different possible inputs. One can now specify a property of an open system by a linear or branching temporal logic formula (over $I \cup O$). Unlike linear temporal logics, in branching temporal logics one can specify possibility requirements such as "every input sequence can be extended so that the output signal v eventually becomes true" (cf. [DTV99]). This is achieved via existential and universal quantification provided in branching temporal logics [Lam80,Eme90]. In this paper, we concentrate on branching temporal logics.

The *realizability problem* for a branching temporal logic is to determine, given a branching-time specification φ, whether there exists a program $f : (2^I)^* \rightarrow 2^O$ whose computation tree satisfies φ [ALW89,PR89]. Realizing φ boils down to synthesizing such a function f. An important aspect of the computation tree associated with f is that it has a fixed branching degree $|2^I|$. It reflects the assumption that at each stage, all possible subsets of I are provided by the environment. Such environments are referred to as *maximal environments*. Intuitively, these are static environments in terms of the branching possibilities they contribute to the associated computation trees. Equivalently, as we have noted already, each program has just one computation tree capturing its behavior in the presence of a maximal environment. In a more general setting, however, we have to consider environments that are, in turn, open systems. We term such environments *reactive*. They might offer different subsets of 2^I as input possibilities at different stages in the computation.

As an illustration, consider $I = \{r_1, \ldots, r_n\}$ and $O = \{t_1, \ldots, t_n\}$ where I represents n different types of resources and O represents n different types of tasks with the understanding that, at each stage, the system needs to receive r_i from the environment in order to execute t_i. In the case of the maximal environment, the specification "it is always possible to reach a stage where t_i is executed" ($AGEF(t_i)$ in CTL parlance) is realizable. This is so because at each stage in the computation, the maximal environment presents all possible combinations of the resources. In the case of the reactive environment, the above specification is *not* realizable; there could be an environment driven by an open system that can produce only a a finite number of the resource r_i along any computation. In the resulting computation tree, each path eventually reaches a node in which the environment stop offering r_i. From then on, t_i cannot be executed.

So, a reactive environment associates a *set* of computation trees with a program. Consequently, in the presence of reactive environments, the realizability problem must seek a program *all* of whose computation trees satisfy the specification.

A closely related problem concerns the controllability of open systems. Here we are given an open system, often called a *plant* in this context, typically

consisting of system states and environment states. We can control the moves made from the system states (i.e. disable some of the possible moves), but not the moves made from the environment states. Given a branching-time specification φ, the *control problem* is to come up with a strategy for controlling the moves made from the system states so that the resulting computation tree satisfies φ. Here again, assuming a reactive environment requires the controller to function correctly no matter how the environment disables some of its moves; thus correctness should be checked with respect to a whole set of computation trees.

We study control problems for both CTL* and CTL specifications. We show that the realizability problem can be reduced to the control problem. We prove that both these problems are 3EXPTIME-complete and 2EXPTIME-complete for CTL* and CTL, respectively. As established in [KV99a,KV99b], these problems are 2EXPTIME-complete and EXPTIME-complete for maximal environments, respectively. In this sense, reactive environments make it more difficult to realize open systems and synthesize controllers for them.

In the literature, a variety of realizability and control problems have been studied for open systems. These studies have been mainly in linear-time settings where the nature of the environment (maximal or reactive) does not play a role (this is since a maximal environment is the most challenging one for the system). In branching-time settings, the emphasis has been mainly on the realizability problem (often referred to as the *synthesis problem*) in the presence of maximal environments. For the linear time case, the literature goes back to a closely related realizability problem due to Church [Chu63], which was solved in [BL69] but more elegantly dealt with later using tree automata [Rab72,PR89]. In branching-time settings, the realizability problem for CTL* and CTL has been solved for maximal environments as cited above and more recently also for the *μ-calculus* [KV00].

As for controller synthesis, as mentioned above, most of the settings considered in the literature are linear time ones and often involve dealing with incomplete information [KG95,KS95,PR89,Var95]. As for branching-time settings, synthesis of memoryless controllers for settings with maximal environments is studied in [Ant95]. Also, for maximal environments, the control problem can be transformed (by flipping the role of the system and the environment) into *module-checking* problems solved in [KV96,KV97]. Yet another work, but in a different framework is [MT98] where the specification is also modeled by a transition system and the correctness criterion is that there should be a behavior preserving simulation from the plant to the specification. This has been later extended to the case of bisimulation in [MT00].

2 The Problem Setting

We assume familiarity with the branching temporal logic CTL* and its sublogic CTL (see [Eme90] for details). Here we just fix the notation for (Kripke) structures, which provide the semantics. A *(Kripke) structure* is a tuple $S =$

$\langle AP, W, R, w_0, L\rangle$, where AP is the set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $R(w, w')$), w_0 is an initial state, and $L : W \to 2^{AP}$ maps each state to a set of atomic propositions true in this state. For w and w' with $R(w, w')$, we say that w' is a successor of w. A *path* of S is an infinite sequence $\pi = w^0, w^1, \ldots$ of states such that for every $i \geq 0$, we have $R(w^i, w^{i+1})$. The suffix $w^i, w^{i+1}, \ldots$ of π is denoted by π^i. We use $w \models \varphi$ to indicate that a state formula φ holds at state w, and we use $\pi \models \varphi$ to indicate that a path formula φ holds at path π (assuming a structure S). We say that S is a model of φ to mean that $w_0 \models \varphi$.

We model a *plant* as $P = \langle AP, W_s, W_e, R, w_0, L\rangle$, where AP, R, w_0, and L are as in a structure with $W = W_s \cup W_e$. Here set of *environment states*. Throughout what follows we are concerned only with *finite* plants; AP and W are both finite sets. We also assume that $W_s \cap W_e = \emptyset$. The *size* of the plant is $|P| = |W| + |R|$.

Given a finite set Υ, an Υ-*tree* is a nonempty set $T \subseteq \Upsilon^*$ such that, if $v.c \in T$ with $v \in \Upsilon^*$ and $c \in \Upsilon$, then $v \in T$. When Υ is clear from the context we call T a tree. The elements of T are called *nodes* and the empty word ε is the *root* of T. For every $v \in T$, the set $succ_T(v) = \{v.c \mid c \in \Upsilon \text{ and } v.c \in T\}$ is the set of children (successors) of v. Where T is clear from the context, we drop the subscript T and write $succ(v)$. In what follows, every tree T we consider is assumed to satisfy $succ(v) \neq \emptyset$ for every v in T. We associate a direction $dir(v) \in \Upsilon$ with each node $v \in T$. A designated element $c_0 \in \Upsilon$ is the direction of ε. For each non-root node $v.c$ with $c \in \Upsilon$, we set $dir(v.c) = c$. A *path* π of a tree is a minimal set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for each $v \in \pi$, we have $|\pi \cap succ(v)| = 1$. Finally, given a set Σ, a Σ-labeled Υ-tree is a pair (T, V) where T is a Υ-tree and $V : T \to \Sigma$ is a labeling function. Of special interest to us are 2^{AP}-labeled trees. We call such trees *computation trees* and we sometimes interpret CTL* formulas with respect to them. Formally, a computation tree (T, V) satisfies a CTL* formula φ if φ is satisfied in the infinite-state structure $\langle AP, T, R_T, \varepsilon, V\rangle$, where $R_T(v, v.c)$ iff $v.c \in succ_T(v)$.

Let $P = \langle AP, W_s, W_e, R, w_0, L\rangle$ be a plant. Then P can be unwound into a W-tree T_P in the obvious manner; thus $\varepsilon \in T_P$, and we set $dir(\varepsilon) = w_0$, and for all $v \in T_P$, we have that $v.c \in T_P$ iff $R(dir(v), c)$. The tree T_P induces the 2^{AP}-labelled tree (T_P, V_P) where for each $v \in T_P$, we have $V_P(v) = L(dir(v))$. A *restriction* (T, V) of (T_P, V_P) is a tree $T \subseteq T_P$ such that V is V_P restricted to T. We denote by $T_P^s = \{v \mid v \in T_P \text{ and } dir(v) \in W_s\}$ the set of nodes of T_P that correspond to system states, and by $T_P^e = T_P \setminus T_P^s$ the set of states that correspond to environment states.

Consider a plant P and a restriction (T, V) of (T_P, V_P). We can think of (T, V) as a computation tree generated by an interaction of the system with its environment: each position in this interaction corresponds to a node $v \in T$. When $v \in T_P^s$, the system chooses a nonempty subset of $succ_{T_P}(v)$. This subset corresponds to successors of $dir(v)$ to which the system enables the transition from $dir(v)$. Similarly, when $v \in T_P^e$, the environment chooses a nonempty subset of $succ_{T_P}(v)$ corresponding to the successors of $dir(v)$ to which transitions are

enabled. Note we can have v and v' with $dir(v) = dir(v')$ and still $succ_T(v) \neq succ_T(v')$. Indeed, the decisions made by the system and the environment depend not only on the current state of the interaction (that is, $dir(v)$), but also in the entire interaction between the system and the environment so far (that is, v). Intuitively, given P and a CTL* formula φ, the control problem is to come up with a strategy for the system so that no matter how the environment responds, the resulting restriction of (T_P, V_P) satisfies φ.

We now make this intuition formal. A *strategy* for the system is a function g that assigns to each $v \in T_P^s$, a non-empty subset of $succ_{T_P}(v)$. A *g-respecting execution* of P is a restriction (T, V) of (T_P, V_P) such that for every $v \in T \cap T_P^s$, we have $succ_T(v) = g(v)$. Note that a strategy g determines only the successors of nodes in T_P that correspond to system states. Thus, P may have many g-respecting executions, each corresponding to a different strategy of the environment. Given a CTL* formula φ, the strategy g is *winning* for φ if all the g-respecting executions satisfy φ. The *control problem* is then to decide, given a plant P and a CTL* specification φ, whether the system has a strategy winning for φ, denoted $controllable(P, \varphi)$.

The realizability problem for programs with reactive environments can be tackled along very similar lines: we consider a program f interacting with its environment via two finite sets I and O of input and output signals respectively. We can view f as a strategy $f : (2^I)^* \to 2^O$ that maps finite sequences of input signal sets into an output signal set. The interaction starts by the program generating the output $f(\varepsilon)$. The environment replies with some response of f for the input sequence (infinite) interaction can be represented by a computation tree. The branches of the tree correspond to external non-determinism caused by different input signal sets chosen by the environment. Thus f can be viewed as the full $2^{I \cup O}$- labeled 2^I-tree (T_f, V_f) with $T_f = (2^I)^*$ and $V_f(v) = dir(v) \cup f(v)$ for each $v \in T_f$. Given a CTL* formula φ, the *realizability problem* is to find a strategy f so that φ is satisfied in f no matter how the environment disables (in a non-blocking manner) its possible responses at different stages. Formally, let $f : (2^I)^* \to 2^O$ be a strategy and let (T, V) be a $2^{I \cup O}$-labeled 2^I-tree. We say that (T, V) is an *f-respecting execution* iff $V(v) = f(v) \cup dir(v)$ for each $v \in T$. (In other words, (T, V) is a restriction of the computation tree (T_f, V_f)). We say that f *realizes* φ if every f-respecting execution satisfies φ. Also, φ is *realizable* iff there is a program f that realizes φ.

Using a universal plant, which embodies all the possible assignments to I and O, the program-synthesis problem can be reduced to the control problem.

Lemma 1. *Let φ be a CTL* (CTL) formula over $AP = I \cup O$. We can effectively construct a finite plant P and a CTL* (resp. CTL) formula φ' such that $|P| = O((2^{AP})^2)$, $|\varphi'| = O(|\varphi| + 2^{AP})$, and φ is realizable iff $controllable(P, \varphi')$.*

Proof. We sketch the reduction for the case of CTL. A similar procedure can be developed for CTL*. We define $P = (AP', W_s, W_e, R, w_0, L)$ as follows:

- $AP' = AP \cup \{p_e\}$ with $p_e \notin AP$. (The role of p_e will become clear soon).
- $W_e = 2^I \times 2^O$; $W_s = \{w_0\} \cup 2^I$ with $w_0 \notin 2^I \cup W_e$

- $R = R_0 \cup R_1 \cup R_2$ where $R_0 = \{w_0\} \times (\{\emptyset\} \times 2^O)$, $R_1 = W_e \times 2^I$, and $R_2 = \{(X,(X,Y))|X \subseteq I \text{ and } Y \subseteq O\}$.
- $L((X,Y)) = X \cup Y \cup \{p_e\}$ for each $(X,Y) \in W_e$ and $L(w) = \emptyset$ for each $w \in W_s$.

Next we construct the CTL formula φ' over AP' by setting $\varphi' = \varphi'_1 \wedge \varphi'_2$. The role of φ'_1 is to ensure that the truthhood of φ matters only at the states in W_e. The conjunct φ'_1 is the formula $EX(\|\varphi\|)$ where $\|\varphi\|$ is given inductively via: $\|p\| = p$ for $p \in AP$, $\|\neg\varphi\| = \neg\|\varphi\|$ and $\|\varphi_1 \vee \varphi_2\| = \|\varphi_1\| \vee \|\varphi_2\|$. Further, $\|EX\varphi\| = EXEX(\|\varphi\|)$ and $\|E(\varphi_1 U \varphi_2)\| = E((p_e \rightarrow \|\varphi_1\|)U(p_e \wedge \|\varphi_2\|))$. Finally, $\|A(\varphi_1 U \varphi_2)\|$ is defined by replacing E by A in the clause for $\|E(\varphi_1 U \varphi_2)\|$.

The conjunct φ'_2 ensures that the system chooses only one move at states in W_s (since the 2^O labelling required must be unique). It is given by $\varphi'_2 = AG(\neg p_e \rightarrow (\bigwedge_{z \in O}(EXz \rightarrow AXz))$ It is easy to check that P and φ' have the required properties. □

3 Upper Bounds

Since our decision procedure uses tree automata, we first introduce Büchi and parity tree automata (see [Tho97] for more details). Tree automata run on Σ-labeled Υ-trees. We assume that the set Υ of directions is ordered. For an Υ-tree T and a node $v \in T$, we use $o\_succ(v)$ to denote an ordered list of $succ(v)$. Let $|\Upsilon| = k$, and let $[k] = \{1,\ldots,k\}$. A *nondeterministic tree automaton* over Σ-labeled Υ-trees is $A = \langle \Sigma, Q, \delta, Q_0, \mathcal{F}\rangle$ where Σ is a finite alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \times [k] \rightarrow 2^{Q^*}$ is a transition function that maps a state, a letter, and a branching degree $i \in [k]$ to a set of i-tuples of states, $Q_0 \subseteq Q$ is the set of initial states, and $\mathcal{F}$ is an acceptance condition that depends on the kind of automata we consider. For *Büchi automata*, $\mathcal{F}$ is a subset of Q and for *parity automata,* $\mathcal{F}$ is a function $\mathcal{F} : Q \rightarrow \{0,\ldots,h\}$ for some $h \in \mathbb{N}$ (the set $\{0,\ldots,h\}$ is called the set of *colors*).

Let (T,V) be a Σ-labeled tree. A *run* of A over (T,V) is a Q-labeled tree (T,ρ) such that $\rho(\varepsilon) \in Q_0$, and for all $v \in T$ with $o\_succ(v) = \langle x_1,\ldots,x_l\rangle$, we have $\langle \rho(x_1),\ldots,\rho(x_l)\rangle \in \delta(q, V(x), l)$. For a path π of T, let $Inf_\rho(\pi)$ be the set of states that appear as the labels of infinitely many nodes on π. If A is a Büchi automaton, then ρ is *accepting* if for every path π of T, $Inf_\rho(\pi) \cap \mathcal{F} \neq \emptyset$. For parity automata, ρ is *accepting* if for every path π of T, $min(\mathcal{F}(Inf_\rho(\pi)))$ is an even number. A Σ-labeled tree (T,V) is accepted by A iff there is an accepting run of A over (T,V). The language accepted by A is the set of all Σ-labeled trees accepted by A. We say that A is *universal* iff it accepts all Σ-labeled Υ-trees.

There are well-known connections relating branching temporal logics and tree automata. For our purposes, it suffices to know the following.

Theorem 1.

(1) [VW86] *Given a CTL formula φ over AP and a finite set Υ, we can construct a nondeterministic Büchi tree automaton $A_{\Upsilon,\varphi}$ with $2^{O(|\varphi|)}$ states that accepts exactly the set of 2^{AP}-labeled Υ-trees that satisfy φ.*

(2) [EJ88,Saf88,Tho97] *Given a CTL* formula φ over AP and a set Υ, one can construct a nondeterministic parity tree automaton $A_{\Upsilon,\varphi}$ with $2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ colors that accepts exactly the set of 2^{AP}-labeled Υ-trees that satisfy φ.* □

In the control problem, we are given a plant $P = \langle AP, W_s, W_e, R, w_0, L\rangle$ and a CTL (or CTL*) formula φ over AP, and we have to decide whether there is a strategy g for the system so that all the g-respecting executions of P satisfy φ.

Recall that a strategy g for the system assigns to each $v \in T_P^s$ a nonempty subset of $succ(v)$. We can associate with g a $\{\bot, \top, d\}$-labeled W-tree (T_P, V_g), where for every $v \in T_P$, the following hold:

- If $v \in T_P^s$, then the children of v that are members of $g(v)$ are labeled by $\top$, and the children of v that are not members of $g(v)$ are labeled by $\bot$.
- If $v \in T_P^e$, then all the children of v are labeled by d.

Intuitively, nodes $v.c$ are labeled by $\top$ if g enables the transitions from $dir(v)$ to c (given that the execution so far has traversed v), they are labeled by $\bot$ if g disables the transition from $dir(v)$ to c, and they are labeled by d if $dir(v)$ is an environment state, where the system has no control about the transition from $dir(v)$ to c being enabled. We call the tree (T_P, V_g) the *strategy tree* of g. Note that not every $\{\bot, \top, d\}$-labeled W-tree (T_P, V) is a strategy tree. Indeed, in order to be a strategy tree, V should label all the successors of nodes corresponding to environment states by d, and it should label all the successors of nodes corresponding to system states by either $\top$ or $\bot$, with at least one successor being labeled by $\top$. In fact, given a plant P with state space W, there is a nondeterministic Büchi automaton A_{stra} over $\{\bot, \top, d\}$-labeled exactly all the strategy trees of P.

Next, given a formula φ we first construct a tree automaton A such that A accepts a strategy tree (T_P, V_g) iff there is a g-respecting execution of P that does not satisfy φ. More precisely, we have:

Theorem 2. *Given a plant P with state space W and a branching-time formula φ, we can construct a nondeterministic tree automaton A over $\{\bot, \top, d\}$-labeled W-trees such that the following hold:*

1. *A accepts a strategy tree (T_P, V_g) iff there is a g-respecting execution of P that does not satisfy φ.*
2. *If φ is a CTL formula, then A is a Büchi automaton with $|W| \cdot 2^{O(\varphi)}$ states.*
3. *If φ is a CTL* formula, then A is a parity automaton with $|W| \cdot 2^{2^{O(|\varphi|)}}$ states and $2^{O(|\varphi|)}$ colors*

Proof. Let $P = \langle AP, W_s, W_e, R, w_0, L\rangle$, and let $A_{W,\neg\varphi} = \langle 2^{AP}, Q, \delta, Q_0, \mathcal{F}\rangle$ be the automaton that accepts exactly all 2^{AP}-labeled W-trees that satisfy $\neg\varphi$, as described in Theorem 1. We define $A = \langle \{\bot, \top, d\}, Q', \delta, Q'_0, \mathcal{F}'\rangle$ as follows.

- $Q' = (W \times Q \times \{\top, \bot\}) \cup \{q_{acc}\}$. The state q_{acc} is an accepting sink. Consider a state $(w, q, m) \in W \times Q \times \{\top, \bot\}$. The last component m is the *mode* of

the state. When $m = \top$, it means that the transition to the current node is enabled (by either the system or the environment). When $m = \bot$, it means that the transition to the current node is disabled.
When A is at a state $(w, q, \top)$ as it reads a node v, it means that $dir(v) = w$, and that v has to participate in the g-respecting execution. Hence, A can read $\top$ or d, but not $\bot$. If v is indeed labeled by $\top$ or d, the automaton A guesses a subset of successors of w of some size $l \geq 1$. It then moves to states corresponding to the successors of w and q, with an appropriate update of the mode ($\top$ for the successors in the guessed subset, $\bot$ for the rest).
When A is in a state $(w, q, \bot)$ and it reads a node v, it means that $dir(v) = w$ and that v does not take part in a g-respecting execution. Then, A expects to read $\bot$ or d, in which case it goes to the accepting sink.

- $Q'_0 = \{w_0\} \times Q_0 \times \{\top\}$.
- The transition function δ' is defined, for all $w \in W$, $q \in Q$, and $l = |succ_{T_P}(w)|$ as follows.
  - $\delta((w, q, \top), \bot, l) = \delta((w, q, \bot), \top, l) = \emptyset$
  - If $x \in \{\bot, d\}$, then $\delta'((w, q, \bot), x, l) = \{\langle q_{acc}, \ldots, q_{acc}\rangle\}$,
  - If $x \in \{\top, d\}$, then $\delta((w, q, \top), x, l)$ is defined as follows. Let $o\_succ(w) = \langle w_1, \ldots, w_l\rangle$ and let $Y = \{w_{y_1}, \ldots, w_{y_n}\}$ be a nonempty subset (of size n) of $succ(w)$. Then, $\delta'((w, q, \top), x, l)$ contains all tuples $\langle(w_1, s_1, m_1), \ldots, (w_l, s_l, m_l)\rangle$ such that there is $\langle q_1, \ldots, q_n\rangle \in \delta(q, L(w), n)$ and for all $1 \leq i \leq l$, the following hold:
    - If $w_i \in Y$, namely $w_i = w_{y_j}$ for some $1 \leq j \leq n$, then $s_i = q_j$ and $m_i = \top$.
    - If $w_i \notin Y$, then $w_i = w$ (in fact, w_i can be an arbitrary state) and $m_i = \bot$.

    Intuitively, δ' propagates the requirements imposed by $\delta(q, L(w), n)$ among the successors of w to which the transition from w is enabled.
    Note that δ' is independent of w being a system or an environment state. The type of w is taken into consideration only in the definition of A_{stra}.
- The final states are inherited from the formula automaton. Thus, if φ is in CTL, then $\mathcal{F}' = (W \times \mathcal{F} \times \{\top, \bot\}) \cup \{q_{acc}\}$. If φ is in CTL*, let $\mathcal{F} : Q \to \{0, \ldots, h\}$. Then, $\mathcal{F}' : Q' \to \{0, \ldots, h\}$ is such that $\mathcal{F}(q_{acc}) = 0$ and for all $w \in W$, $q \in Q$, and $m \in \{\bot, \top\}$, we have $\mathcal{F}'((w, q, m)) = \mathcal{F}(q)$. □

It is left to check whether the language of A_{stra} is contained in the language of A. Since tree automata are closed under complementation [Rab69,Tho97], we can complement A, get an automaton $\tilde{A}$, and then check the nonemptiness of the intersection of A_{stra} with $\tilde{A}$. Hence the following theorem.

Theorem 3. *Given a plant P and a formula φ in CTL, the control problem for φ is in 2EXPTIME. More precisely, it can be solved in time $O(\exp(|P|^2 \cdot 2^{O(|\varphi|)}))$.*[1] *For φ in CTL*, the problem is in 3EXPTIME. More precisely, it can be solved in time $O(\exp(|P|^2 \cdot 2^{2^{O(|\varphi|)}}))$.*

[1] exp(x) stands for $2^{O(x)}$

Proof. For the complexity of this procedure, it is easy to see that if φ is in CTL, the automaton A has a state-space size of $O(|P| \cdot 2^{O(|\varphi|)})$. Though A runs on k-ary trees (where k depends on P), it can be complemented as easily as automata on binary trees — the complemented automaton $\tilde{A}$ (as well as its intersection with A_{stra}) is a parity automaton with $O(\exp(|P| \cdot 2^{O(|\varphi|)}))$ states and $O(|P| \cdot 2^{O(|\varphi|)})$ colors ([Tho97]). Since emptiness of parity tree automata can be done in time polynomial in the state-space and exponential in the number of colors [EJ88,PR89], we can check emptiness of this intersection in time $O(\exp(|P|^2 \cdot 2^{O(|\varphi|)}))$. For CTL* specifications, the analysis is similar except that the complexity contributed by the formula increases by one exponential. □

By [Rab69], if there is indeed a strategy that is winning for the system, then the automaton that is the product of A_{stra} and the complement of the automaton constructed on in Theorem 2 accepts it and when we test it for emptiness, we can get a *regular* tree accepted by the automaton. This then provides a finite-memory winning strategy that can be realized as a finite-state controller for the system. We note that our upper bounds apply also to the realizability problem.

4 Lower Bounds

For two 2^{AP}-labeled trees (T, V) and (T', V'), and a set $Q = \{q_1, \ldots, q_k\} \subseteq AP$, we say that (T, V) and (T', V') are *Q-different* if they agree on everything except possibly the labels of the propositions in Q. Formally, $T = T'$ and for all $x \in T$, we have $V(x) \setminus Q = V'(x) \setminus Q$. The logic AQCTL* extends CTL* by universal quantification on atomic propositions: if ψ is a CTL* formula and $q_1, \ldots, q_k$ are atomic propositions, then $\forall q_1, \ldots, q_k \psi$ is an AQCTL* formula. The semantics of $\forall q_1, \ldots, q_k \psi$ is given by $S \models \forall q_1, \ldots, q_k \psi$ iff for all trees (T, V) such that (T, V) and the unwinding (T_S, V_S) of S are $\{q_1, \ldots, q_k\}$-different, $(T, V) \models \psi$. The logics AQLTL and AQCTL are defined similarly as the extensions of LTL and CTL with universal quantification on atomic propositions.

The following theorem is taken from [SVW87]. We describe here the full proof, as our lower-bound proofs are based on it. We assume familiarity with LTL [Eme90].

Theorem 4. [SVW87] *The satisfiability problem for AQLTL is EXPSPACE-hard.*

Proof. We reduce the problem of checking whether an exponential-space deterministic Turing machine T accepts an input word x. That is, given T and x, we construct an AQLTL formula formula $\forall q \varphi$ such that T accepts x iff $\forall q \varphi$ is satisfiable. Below we describe the formula φ informally. The formal description of φ and of the function *next* we use below are given in the full version.

Let $T = \langle \Gamma, Q, \rightarrow, q_0, F \rangle$, where Γ is the alphabet, Q is the set of states, $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the transition relation (we use $(q, a) \rightarrow (q', b, \Delta)$ to indicate that when T is in state q and it reads the input a in the current tape cell, it moves to state q', writes b in the current tape cell, and its reading head

moves one cell to the left or to the right, according to Δ), q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. Let $n = a \cdot |x|$, for some constant a, be such that the working tape of T has 2^n cells. We encode a configuration of T by a word $\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{2^n}$. The meaning of such a configuration is that the j^{th} cell of T, for $1 \leq j \leq 2^n$, is labeled γ_j, the reading head points on cell i, and T is in state q. We now encode a computation of T by a sequence of configurations.

Let $\Sigma = \Gamma \cup (Q \times \Gamma)$. We can encode letters in Σ by a set $AP(T) = \{p_1, \ldots, p_m\}$ (with $m = \lceil log|\Sigma| \rceil$) of atomic propositions). We define our formulas over the set $AP = AP(T) \cup \{b, c, d, e, q\}$ of atomic propositions. The task of the last five atoms will be explained shortly. Since T is fixed, so is Σ, and hence so is the size of AP.

Consider an infinite sequence π over 2^{AP}. For an atomic proposition $p \in AP$ and a node u in π, we use $p(u)$ to denote the truth value of p at u. That is, $p(u)$ is 1 if p holds at u and is 0 if p does not hold at u. We divide the sequence π to blocks of length n. Every such block corresponds to a single tape cell of the machine T. Consider a block $u_1, \ldots, u_n$ that corresponds to a cell ρ. We use the node u_1 to encode the content of cell ρ. Thus, the bit vector $p_1(u_1), \ldots, p_m(u_1)$ encodes the letter (in $\Gamma \cup (Q \times \Gamma)$) that corresponds to cell ρ. We use the atomic proposition b to mark the beginning of the block; that is, b should hold on u_1 and fail on $u_2, \ldots, u_n$ Recall that the letter with which cell ρ is labeled is encoded at the node u_1 of the block $u_1, \ldots, u_n$ that corresponds to ρ. Why then do we need a block of length n to encode a single letter? The block also encodes the location of the cell ρ on the tape. Since T is an exponential-space Turing machine, this location is a number between 0 and $2^n - 1$. Encoding the location eliminates the need for exponentially many X operators when we attempt to relate two successive configurations. Encoding is done by the atomic proposition c, called *counter*, Let $c(u_n), \ldots, c(u_1)$ encode the location of ρ. A sequence of 2^n blocks corresponds to 2^n cells and encodes a configuration of T. The value of the counters along this sequence goes from 0 to $2^n - 1$, and then start again from 0. This can be enforced by a conjunct in φ which is only $O(n)$-long by using the proposition d as a carry-bit. The atomic proposition e marks the last block of a configuration, that is, e holds in a node u_1 of a block $u_1, \ldots, u_n$ iff c holds on all nodes in the block

Let $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$ be two successive configurations of T. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 \leq i \leq 2^n$ (taking σ_{2^n+1} to be σ'_1 and σ_0 to be the label of the last letter in the configuration before $\sigma_1 \ldots \sigma_{2^n}$, or some special label when $\sigma_1 \ldots \sigma_{2^n}$ is the initial configuration), we know, by the deterministic transition relation of T, what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i.

Consistency with *next* gives us a necessary condition for a word to encode a legal computation. In addition, the computation should start with the initial configuration and end in a final configuration (with $q \in F$) — these properties can easily be enforced by φ.

The difficult part in the reduction is in guaranteeing that the sequence of configurations is indeed consistent with *next*. To enforce this, we have to re-

late σ_{i-1}, σ_i, and σ_{i+1} with σ'_i for any i in any two successive configurations $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$. One natural way to do so is by a conjunction of formulas like "whenever we meet a cell with counter $i-1$ and the labeling of the next three cells forms the triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$, then the next time we meet a cell with counter i, this cell is labeled $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$". The problem is that as i can take any value from 1 to 2^n, there are exponentially many such conjuncts. This is where the universal quantification of the AQLTL comes into the picture. It enables us to relate $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with σ'_i, for all i.

To understand how this is done, consider the atomic proposition q, and assume that the following hold. **(1)** q is true at precisely two points, both are points in which a block starts, **(2)** there is exactly one point between them (possibly in exactly one of them) in which e holds (thus, the two points are in successive configurations), and **(3)** the value of the counter at the blocks starting at the two points is the same. Then, it should be true that **(4)** if the labels of the three blocks starting one block before the first q are σ_{i-1}, σ_i, and σ_{i+1}, then the block starting at the second q is labeled by $next(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$.

The formula φ contains the conjunct $(((\mathbf{1}) \wedge (\mathbf{2}) \wedge (\mathbf{3})) \rightarrow (\mathbf{4}))$. and ensures that only computations consistent with *next* are satisfied by φ. Hence, $\forall q \varphi$ is satisfiable iff there is an accepting computation of T on x. □

We now show that AQCTL is also strong enough to describe an exponential-space Turing machine with a formula of polynomial length. Moreover, since CTL has both universal and existential path quantification, AQCTL can describe an alternating exponential-space Turing machine, implying a 2EXPTIME lower bound for its satisfiability problem [CKS81].

Theorem 5. *The satisfiability problem for AQCTL is 2EXPTIME-hard.*

Proof. We do a reduction from the problem whether an exponential-space alternating Turing machine T accepts an input word x. That is, given T and x, we construct an AQCTL formula $\forall q \psi$ such that T accepts x iff $\forall q \psi$ is satisfiable.

Let $T = \langle \Gamma, Q_u, Q_e, \mapsto, q_0, F \rangle$, where the sets Q_u and Q_e of states are disjoint, and contain the universal and the existential states, respectively. We denote their union (the set of all states) by Q. Our model of alternation prescribes that $\mapsto \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ has a binary branching degree. When a universal or an existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, a) \mapsto \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading input symbol a, it branches to the left with (q_l, b_l, Δ_l) and to the right with (q_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.)

For a configuration c of T, let $succ_l(c)$ and $succ_r(c)$ be the successors of c when applying to it the left and right choices in $\mapsto$, respectively. Given an input x, a computation tree of T on x is a tree in which each node corresponds to a configuration of T. The root of the tree corresponds to the initial configuration. A node that corresponds to a universal configuration c has two successors,

corresponding to $succ_l(c)$ and $succ_r(c)$. A node that corresponds to an existential configuration c has a single successor, corresponding to either $succ_l(c)$ or $succ_r(c)$. The tree is an accepting computation tree if all its branches reach an accepting configuration.

The formula ψ will describe accepting trees. As in the linear case, we encode a configuration of T by a sequence $\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{2^n}$, and we use a block of length n to describe each letter $\sigma_i \in \Gamma \cup (Q \times \Gamma)$ in the sequence. The construction of ψ is similar to the construction described for φ in the linear case. As in the linear case, the atomic propositions c and d are used to count, b is used to mark the beginning of blocks, and e is used to mark the last letter in a configuration. The formulas which ensure the correct behaviour of c, d, b and e can be suitably prefixed with the universal path quantifier to ensure they behave correctly along all paths of the tree.

The difficult part is to check that the $succ_l$ and $succ_r$ relations are maintained. For that, we add two atomic propositions, e_E and e_U, that refine the proposition e — exactly one of them hold whenever e holds and e_E (e_U) holds iff the configuration which has just ended is existential (universal).

In addition, we use an atomic proposition l to indicate whether the nodes belong to a left or a right successor. For clarity, we denote $\neg l$ by r. Formally, ψ contains the conjunct $AG(l \rightarrow (AlUe)) \wedge AG(r \rightarrow (ArUe))$. Since a universal configuration c has both $succ_l(c)$ and $succ_r(c)$ as successors, and an existential c has only one of them, ψ also contains the conjunct $AG(e_E \rightarrow (EXl \vee EXr)) \wedge AG(e_U \rightarrow (EXl \wedge EXr))$.

We can now use universal quantification over atomic propositions in order to check consistency with $succ_l$ and $succ_r$. Note that $succ_l(c)$ and $succ_r(c)$ are uniquely defined. Thus, we can define functions, $next_l$ and $next_r$, analogous to function $next$ of the linear case. Given a sequence $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ of letters in c, the function $next_l(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ returns the expectation for the i'th letter in $succ_l(c)$. We denote this letter by σ_i^l, and similarly for $next_r$ and σ_i^r.

In the linear case, we considered assignments to q in which q holds at exactly two points in the computation. Here, we look at assignments where q holds at exactly two points in each branch. The first point is a node where a block of σ_i starts, and the second point is a node where a block of σ_i^l or σ_i^r starts (note that each assignment to q may check consistency with $succ$ along different branches)[2]

The formula can also ensure that in every branch with two occurrences of q, there is exactly one node between them in which e holds (thus, the two nodes are in successive configurations) and that the value of the counter at the blocks starting at the two points is the same. If the q-labelling satisfies these properties, then φ will demand that if the three blocks starting before the first q along a path are σ_{i-1}, σ_i, σ_{i+1}, then the blocks starting at the second q must be labeled by $next_l(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ (if the second q belongs to the left branch) or $next_r(\sigma_{i-1}, \sigma_i, \sigma_{i+1})$ (if it belongs to the right branch).

[2] It is convenient to think of a satisfying tree for $\forall q\psi$ as a tree that has branching degree 1 everywhere except for nodes labeled by e_U, where the branching degree is 2. Our reduction, however, makes no assumption about such a structure.

One can then show that ψ is satisfied only in a computation tree consistent with $succ_l$ and $succ_r$. Hence, $\forall q\psi$ is satisfiable iff there is an accepting computation tree of T on x. □

The satisfiability problem for CTL* is exponentially harder than the one for CTL. We now show that this computational difference is preserved when we look at the extensions of these logics with universal quantification over atomic propositions.

Theorem 6. *The satisfiability problem for A*Q*CTL* is 3EXPTIME-hard.*

Proof. We do a reduction from the problem whether a doubly-exponential-space alternating Turing machine T accepts an input word x. That is, given T and x, we construct an AQCTL* formula $\forall q\psi$ such that T accepts x iff ψ is satisfiable.

In [VS85], the satisfiability problem of CTL* is proved to be 2EXPTIME-hard by a reduction from an exponential-space alternating Turing machine. Below we explain how universal quantification can be used to "stretch" the length of the tape that a polynomial CTL* formula can describe by another exponential. As in the proof of Theorem 4, the formula in [VS85] maintains an n-bit counter, and each cell of T's tape corresponds to a block of length n.

In order to point on the letters σ_i and σ'_i simultaneously (that is, the letters that the atomic proposition q point on in the proof of Theorem 4), [VS85] adds to each node of the tree a branch such that nodes that belong to the original tree are labeled by some atomic proposition p, and nodes that belong to the added branches are not labeled by p. Every path in the tree has a single location where the atom p stops being true. [VS85] uses this location in order to point on σ' and in order to compare the values of the n-bit counter in the current point (where σ is located) and in the point in the computation where p stops being true.

On top of the method in [VS85], we use the universal quantification in order to maintain a 2^n-bit counter and thus count to 2^{2^n}. Typically, each bit of our 2^n-bit counter is kept in a block of length n, which maintains the index of the bit (a number between 0 to $2^n - 1$). For example, when $n = 3$, the counter looks as follows.

000	001	010	011	100	101	110	111		000	001	010	011	100	101	110	111 ← n-bit counter
0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	1 ← 2^n-bit counter
000	001	010	011	100	101	110	111		000	001	010	011	100	101	110	111 ...
0	0	0	0	0	0	1	0		0	0	0	0	0	0	1	1 ...

To check that the 2^n-bit counter proceeds properly, we use a universally quantified proposition q and we check that if q holds at exactly two points (say, last points in a block of the n-bit counter), with the same value to the n-bit counter, and with only one block between them in which the n-bit counter has value 1^n, then the bit of the 2^n-bit counter that is maintained at the block of the second q is updated properly (we also need to relate and update carry bits, but the idea is the same). □

Note that the number of atomic propositions in ψ in the proofs of both Theorems 5 and 6 is fixed. Note also that if ψ is satisfiable, then it is also

satisfied in a tree of a fixed branching degree (a careful analysis can show that for CTL the sufficient branching degree is 2, and for CTL* it is 3).

The logic EAQCTL* extends AQCTL* by adding existential quantification on atomic propositions: if $\forall q_1, \ldots, q_k \psi$ is an AQCTL* formula and $p_1, \ldots, p_m$ are atomic propositions, then $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$ is an EAQCTL* formula. The semantics is given by $S \models \exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$ iff there is a tree (T, V) such that (T_S, V_S) and (T, V) are $\{p_1, \ldots, p_m\}$-different and $(T, V) \models \forall q_1, \ldots, q_k \psi$. The logic EAQCTL is the subset of EAQCTL* corresponding to CTL. It is not difficult now to show that the following Theorem follows from Theorems 5 and 6.

Theorem 7. *The model-checking problems for EAQCTL and EAQCTL* are 2EXPTIME-hard and 3EXPTIME-hard in the size of the specification, respectively.* □

Intuitively, the model-checking problem for EAQCTL* asks whether we can find an assignment to the propositions that are existentially quantified so that no matter how we assign values to the propositions that are universally quantified, the formula is satisfied. Recall that in the control problem we ask a similar question, namely whether there a strategy for the system so that no matter which strategy the environment uses, the formula is satisfied. In this spirit, it is not diffcult to make the relation between existential and universal quantification over atomic propositions and supervisory control formal. The technique is similar to the relation between existential quantification and the module-checking problem, as described in [KV96]. Consequently, we can show :

Theorem 8. *Given an EAQCTL* formula $\exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$, and a structure S, there is a plant P and a CTL* formula ψ' such that $|P| = O((1 + k+m) \cdot |S|)$, $|\psi'| = O(|S| + |\psi|)$, and $S \models \exists p_1, \ldots, p_m \forall q_1, \ldots, q_k \psi$ iff controllable (P, ψ').* □

Since the number of atomic propositions in the formulas used in the reductions in Theorems 5 and 6 is fixed, and since in the case P is fixed the size of ψ' in Theorem 8 is $O(|\psi|)$, we can conclude with the following.

Theorem 9. *The control problems for CTL and CTL* are 2EXPTIME-hard and 3EXPTIME-hard in the size of the specification, respectively.* □

We note that these lower bounds apply also to the realizability problem.

References

ALW89. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th ICALP*, LNCS 372, 1989. 93

Ant95. M. Antoniotti. *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system.* PhD thesis, New York University, New York, 1995. 94

BL69. J. R. Büchi and L.HG. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969. 94

Chu63. A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pages 23–35. institut Mittag-Leffler, 1963. 94

CKS81. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981. 102

DTV99. M. Daniele, P. Traverso, and M. Y. Vardi. Strong cyclic planning revisited. In *5th European Conference on Planning*, pages 34–46, 1999. 93

EJ88. E. A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th FOCS*, pages 328–337, White Plains, October 1988. 98, 100

Eme90. E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990. 93, 94, 100

HP85. D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985. 92

KG95. R. Kumar and V. K. Garg. *Modeling and control of logical discrete event systems*. Kluwer Academic Publishers, 1995. 94

KS95. R. Kumar and M. A. Shayman. Supervisory control of nondeterministic systems under partial observation and decentralization. *SIAM Journal of Control and Optimization*, 1995. 94

KV96. O. Kupferman and M. Y. Vardi. Module checking. In *Proc. 8th CAV*, LNCS 1102, pages 75–86, 1996. 94, 105

KV97. O. Kupferman and M. Y. Vardi. Module checking revisited. In *Proc. 9th CAV*, LNCS 1254, pages 36–47, 1997. 94

KV99a. O. Kupferman and M. Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245 – 263, June 1999. 94

KV99b. O. Kupferman and M. Y. Vardi. Robust satisfaction. In *Proc. 10th CONCUR*, LNCS 1664, pages 383–398, 1999. 94

KV00. O. Kupferman and M. Y. Vardi. μ-calculus synthesis. In *Proc. 25th MFCS*. 94

Lam80. L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th POPL* pages 174–185, January 1980. 93

MT98. P. Madhusudan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *Proc 9th CONCUR*, LNCS 146, pages 18–33, 1998. 94

MT00. P. Madhusudan and P. S. Thiagarajan. Branching time controllers for discrete event systems. *To appear in CONCUR'98 Special Issue, Theoretical Computer Science*, 2000. 94

PR89. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pages 179–190, 1989. 93, 94, 100

Rab69. M. O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969. 99, 100

Rab72. M. O. Rabin. Automata on infinite objects and Church's problem. *Amer. Mathematical Society*, 1972. 94

Saf88. S. Safra. On the complexity of ω-automata. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, October 1988. 98

SVW87. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987. 100

Tho97. W. Thomas. Languages, automata, and logic. *Handbook of Formal Language Theory*, III:389–455, 1997. 97, 98, 99, 100

Var95. M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. 7th CAV*, LNCS 939, pages 267–292, 1995. 94

VS85. M. Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th STOC*, pages 240–251, 1985. 104

VW86. M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, 1986. 97

Model Checking with Finite Complete Prefixes Is PSPACE-Complete*

Keijo Heljanko

Helsinki University of Technology, Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
Keijo.Heljanko@hut.fi

Abstract. Unfoldings are a technique for verification of concurrent and distributed systems introduced by McMillan. The method constructs a finite complete prefix, which can be seen as a symbolic representation of an interleaved reachability graph. We show that model checking a fixed size formula of several temporal logics, including LTL, CTL, and CTL*, is PSPACE-complete in the size of a finite complete prefix of a 1-safe Petri net. This proof employs a class of 1-safe Petri nets for which it is easy to generate a finite complete prefix in polynomial time.

1 Introduction

Unfoldings are a technique for verification of concurrent and distributed systems introduced by McMillan [19]. It can be applied to systems modeled by Petri nets, communicating automata, or process algebras [10,9,18]. The method is based on the notion of *unfolding*, which can be seen as the partial order version of an (infinite) computation tree [10,3].

The unfolding based verification methods construct a *finite complete prefix*, which is a finite initial part of the unfolding containing all the information about the interleaved reachability graph of the system in question. Thus a finite complete prefix can be seen as a symbolic representation of the reachability graph. The finite complete prefix is not, however, a canonical representation of the reachability graph in the same way as a ROBDD (reduced ordered binary decision diagram) is when the variable ordering is fixed [11].

McMillan showed that the deadlock checking problem is NP-complete in the size of a finite complete prefix of a 1-safe Petri net. A small variation of that proof can be used to show that reachability is also NP-complete, see e.g. [13]. Because reachability is PSPACE-complete in the size of a 1-safe Petri net, the prefix generation process mapped this PSPACE-complete problem into (a potentially exponentially larger) NP-complete problem.

* The financial support of Helsinki Graduate School in Computer Science and Engineering, Academy of Finland (Project 47754), Foundation for Financial Aid at Helsinki University of Technology, Emil Aaltonen Foundation, and Nokia Foundation are gratefully acknowledged.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 108–122, 2000.

We will show that model checking a fixed CTL formula containing nested temporal modalities is PSPACE-complete in the size of a finite complete prefix of a 1-safe Petri net. Because model checking a fixed CTL formula is also PSPACE-complete in the size of a 1-safe Petri net [6], using a prefix as input to a model checker does not change the complexity of CTL model checking. The fixed CTL formula we use can be expressed in most temporal logics interpreted over interleaved reachability graphs, and we obtain PSPACE-completeness results for several of them.

Our proof employs a class of 1-safe Petri nets for which it is easy to generate a finite complete prefix in deterministic polynomial time. We will show that the prefixes generated by currently employed prefix generation algorithms (see [10]) can sometimes be exponentially larger than what is allowed by the semantic prefix completeness criterion. We do not know whether these prefixes have some properties which would make model checking using them easier than with prefixes fulfilling only the semantic prefix completeness criterion.

The rest of the paper is structured as follows. First in Sect. 2 we define the Petri net notation used in this paper, followed by the definition of finite complete prefixes. We show in Sect. 3 that it is possible to sometimes create exponentially smaller prefixes than the algorithm of [10]. Next in Sect. 4 we present the main result of this work, a proof of model checking PSPACE-completeness for several logics in the size of a finite complete prefix. We give the conclusions in Sect. 5.

2 Petri Net Definitions

Our aim is to define *finite complete prefixes* as a symbolic representation of a reachability graph of a 1-safe Petri net system. Finite complete prefixes are *branching processes* fulfilling some additional constraints. To define branching processes we introduce *occurrence nets*, which are Petri nets of a restricted form. We do the definitions bottom-up, and begin with basic Petri net notation. We follow mainly the notation of [10].

2.1 Petri Nets

A triple $\langle S, T, F\rangle$ is a *net* if $S \cap T = \emptyset$ and $F \subseteq (S \times T) \cup (T \times S)$. The elements of S are called *places*, and the elements of T *transitions*. Places and transitions are also called *nodes*. We identify F with its characteristic function on the set $(S \times T) \cup (T \times S)$. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in S \cup T \,|\, F(y, x) = 1\}$. The *postset* of a node x, denoted by $x^\bullet$, is the set $\{y \in S \cup T \,|\, F(x, y) = 1\}$. Their generalizations on sets of nodes $X \subseteq S \cup T$ are defined as ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$, and $X^\bullet = \bigcup_{x \in X} x^\bullet$, respectively.

A *marking* of a net $\langle S, T, F\rangle$ is a mapping $S \mapsto \mathbb{N}$. A marking M is identified with the multi-set which contains $M(s)$ copies of s for every $s \in S$. A 4-tuple $\Sigma = \langle S, T, F, M_0\rangle$ is a *net system* if $\langle S, T, F\rangle$ is a net and M_0 is a marking of $\langle S, T, F\rangle$. A marking M enables a transition $t \in T$ if $\forall s \in S : F(s, t) \leq M(s)$. If t is enabled, it can *occur* leading to a new marking (denoted $M \xrightarrow{t} M'$), where M'

is defined by $\forall s \in S : M'(s) = M(s) - F(s,t) + F(t,s)$. A marking M_n is *reachable* in Σ if there is an *execution*, i.e. a sequence of transitions $t_1, t_2, \ldots, t_n$ and markings $M_1, M_2, \ldots, M_{n-1}$ such that: $M_0 \stackrel{t_1}{\rightarrow} M_1 \stackrel{t_2}{\rightarrow} \ldots M_{n-1} \stackrel{t_n}{\rightarrow} M_n$. A reachable marking is 1-safe if $\forall s \in S : M(s) \leq 1$. A net system Σ is 1-safe if all its reachable markings are 1-safe. In this work we will restrict ourselves to net systems which are 1-safe, have a finite number of places and transitions, and also in which each transition has both nonempty pre- and postsets.

2.2 Occurrence Nets

We use $<_F$ ($\leq_F$) to denote the (reflexive) transitive closure of F. Define $\Sigma = \langle S, T, F \rangle$ to be a net and let $x_1, x_2 \in S \cup T$. The nodes x_1 and x_2 are in *conflict*, denoted by $x_1 \# x_2$, if there exist $t_1, t_2 \in T$ such that $t_1 \neq t_2$, ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, $t_1 \leq_F x_1$, and $t_2 \leq_F x_2$.

An occurrence net is a net $N = \langle B, E, F \rangle$ such that:

- $\forall b \in B : |{}^\bullet b| \leq 1$,
- F is acyclic, i.e. the irreflexive transitive closure of F is a partial order,
- N is finitely preceded, i.e. for any node x of the net, the set of nodes y such that $y \leq_F x$ is finite, and
- $\forall x \in B \cup E : \neg(x \# x)$.

The elements of B and E are called *conditions* and *events*, respectively. The set $Min(N)$ denotes the set of minimal elements of $<_F$. In this work the minimal elements will be conditions, and thus $Min(N)$ can be seen as an initial marking.

A *configuration* C of an occurrence net is a set of events satisfying:

- If $e \in C$ then $\forall e' \in E : e' \leq_F e$ implies $e' \in C$ (C is causally closed), and
- $\forall e, e' \in C : \neg(e \# e')$ (C is conflict-free).

A *local configuration* $[e]$ of an event e is the set of events e', such that $e' \leq_F e$. A *level* of an event e is the length i of the longest sequence $e_1, e_2, \ldots, e_i$ of events, such that $e_i = e$, and $e_1 <_F e_2 <_F \ldots <_F e_i$. Thus $level(e) = 1$ when ${}^\bullet e \subseteq Min(N)$.

2.3 Branching Processes

Branching processes are "unfoldings" of net systems and were introduced by Engelfriet [4]. Let $N_1 = \langle S_1, T_1, F_1 \rangle$ and $N_2 = \langle S_2, T_2, F_2 \rangle$ be two nets. A *homomorphism* h is a mapping $S_1 \cup T_1 \mapsto S_2 \cup T_2$ such that: $h(S_1) \subseteq S_2$, $h(T_1) \subseteq T_2$, and for all $t \in T_1$, the restriction of h to ${}^\bullet t$ is a bijection between ${}^\bullet t$ and ${}^\bullet h(t)$, and similarly for $t^\bullet$ and $h(t)^\bullet$. A *branching process* of a net system $\Sigma = \langle S, T, F, M_0 \rangle$ is a tuple $\beta = \langle N', h \rangle$, where $N' = \langle B', E', F' \rangle$ is an occurrence net, and h is a homomorphism from N' to $\langle S, T, F \rangle$ such that: the restriction of h to $Min(N')$ is a bijection between $Min(N')$ and M_0, and $\forall e_1, e_2 \in E'$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $h(e_1) = h(e_2)$, then $e_1 = e_2$. Thus h maps the conditions and events of an occurrence net to the places and transitions of the corresponding net system in a

way which respects the initial marking and the labeling of the transitions and their pre- and postsets.

The marking of Σ associated with a configuration C of β is denoted by $Mark(C) = h((Min(N) \cup C^\bullet) \setminus {}^\bullet C)$. A configuration of the branching process always corresponds to a reachable marking of Σ. It is shown in [4] that a net system has a maximal branching process up to isomorphism, called the *unfolding*. If the net system has some infinite behavior, the unfolding will also be infinite.

2.4 Finite Complete Prefixes

We now define *finite complete prefixes*:

Definition 1. *A finite branching process β of a net system Σ is* a finite complete prefix *if for each reachable marking M of Σ there exists a configuration C of β, such that:*

- $Mark(C) = M$, *and*
- *for every transition t enabled in M there exists a configuration $C \cup \{e\}$ of β, such that $e \notin C$ and $h(e) = t$.*

A finite complete prefix contains all the information about the interleaved reachability graph of the net system. Algorithms to obtain a finite complete prefix given a (finite state) net system are presented in [10,9,19]. The algorithms will mark some events of the prefix β as special *cut-off events*, which we will denote by events marked with crosses in the figures. The intuition behind cut-off events is that they correspond to repetition of behavior found "earlier" in the prefix. Due to space limitations we direct the interested reader to [10,9,19].

3 Compact Finite Complete Prefixes

It is well known that finite complete prefixes can sometimes be exponentially more succinct than the reachability graph of the net system [19]. A simple example of such a family of net systems (with the instance $n = 4$ in Fig. 1) has 2^n reachable markings, while the finite complete prefix is polynomial in the size of the net system. In fact, the finite complete prefixes of this family of net systems are isomorphic to the net system itself. The improved prefix generation algo-

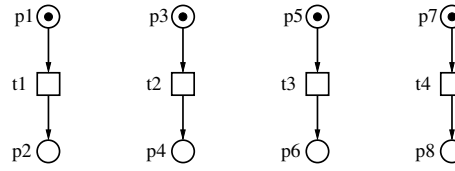


Fig. 1. A 1-safe net system with a compact prefix

rithm by Esparza, Römer, and Vogler [10] guarantees for 1-safe net systems that the number of non-cut-off events of the generated prefix is never larger than the number of reachable markings. What is not to our knowledge presented in the literature is the fact that sometimes the prefix generation algorithm by McMillan [19] (and also the improved version [10]) creates exponentially larger prefixes than are needed to fulfill the semantic prefix completeness criterion.

For an example of such a family of 1-safe Petri net systems, see Fig. 2. This net system is an instance of a binary counter net system with initialization to a "random" initial state (Fig. 2 is a three bit instance, i.e. $n = 3$). The net system acts like a binary counter starting from all low bits, when the initial marking is $M_0' = \{s(c_0), s(b_0l), s(b_1l), s(b_2l)\}$. The contents of the binary counter are consistent when the place $s(c_0)$ is marked, otherwise the carry propagation can be thought to be in progress. The exact behavior of the net system is actually of no interest to us, we are only interested in the sizes of different finite complete prefixes generated from it.

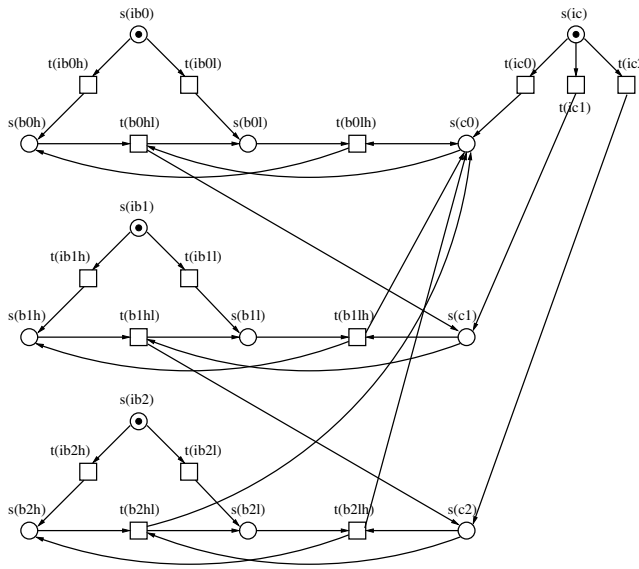


Fig. 2. A three bit binary counter net system

Let us consider what happens when the initial marking is the marking in Fig. 2. We can find the invariants: $M(s(ic)) + \sum_{0 \le i < n} M(s(c_i)) = 1$, and for all $0 \le j < n : M(s(ib_j)) + M(s(b_jh)) + M(s(b_jl)) = 1$. These $n+1$ invariants give an upper bound of $(n+1) \cdot 3^n$ reachable markings. This is also the exact number of reachable markings, which can also be seen by simple static analysis. Namely, firing one (or none) of the transitions of the set $\{t(ic_0), \ldots, t(ic_{n-1})\}$ can set the first invariant to any of $n+1$ values, and also firing one (or none) of the

transitions $\{t(ib_jh), t(ib_jl)\}$ can set the invariant of the bit j into any of three values. Also in the initial state these $n+1$ sets of transitions are concurrently enabled, and thus firing a transition from one set does not disable transitions from other sets. Thus all the $(n+1) \cdot 3^n$ reachable markings are within one concurrent "step" from the initial marking.

We can actually create the finite prefix of Fig. 3 from this net system, and then verify that it fulfills the semantic prefix completeness criterion (Def. 1). We can see that the prefix of Fig. 3 is polynomial in the size of the counter net system of Fig. 2, and that such a "compact prefix" can be constructed for a counter net system of any fixed amount of bits.

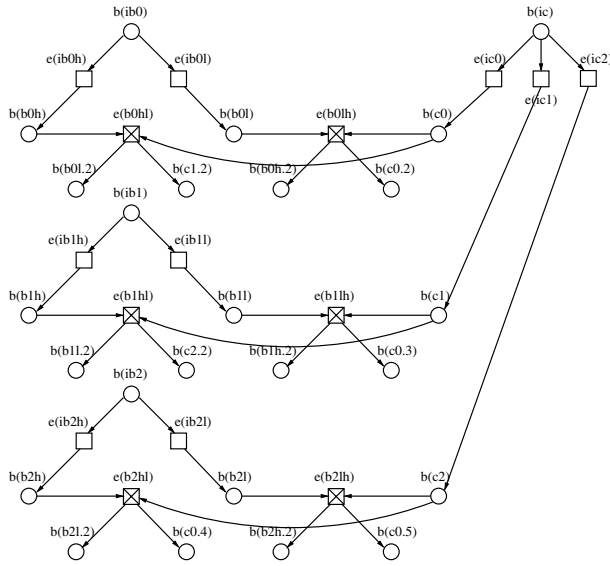


Fig. 3. A finite complete prefix for the three bit counter example

Here we give a proof sketch for the completeness of the finite prefix in Fig. 3. The prefix is identical to the two first levels of the unfolding of the net system of Fig. 2. The first requirement of prefix completeness is fulfilled, as all of the reachable markings can be reached by a configuration containing only events from the first level of the prefix. The second prefix completeness criterion intuitively requires that all the arcs of the reachability graph are present in the prefix. This is the case, because both the first and second levels of this prefix are identical to the unfolding, and thus they contain extensions for all the configurations (of the first level) mentioned in the second completeness criterion.

Note that all the events on the second level are marked as cut-off events, as they introduce no new reachable markings to the prefix. This requires allowing

that the corresponding configuration (see [10]) of a cut-off event is a non-local configuration. Such an idea was already presented in our earlier work [14].

When we consider the sizes of finite complete prefixes generated by the currently employed prefix generation tools, the picture is quite different. We have gathered prefix sizes for small instances of this family of net systems in Table 1. For this family of examples, the McMillan's algorithm [19] and the improvement by Esparza et.al. [10] (implemented in the PEP tool [1]) both generate the same prefixes. While the number of non-cut-off events (the column $|E|-\#c$ of McMillan/ERV Prefix) grows much more slowly than the number of reachable markings, the growth in this column is still exponential (we get the recursion $x_i = 2x_{i-1} + i + 4$, with the initial value $x_2 = 16$). Contrast this with the compact prefix, whose size grows polynomially in the number of bits in the counter. Thus the prefixes generated by the current prefix generation al-

Table 1. Prefix sizes for counter net systems

System			Reachability Graph		McMillan/ERV Prefix				Compact Prefix			
Size	$\|S\|$	$\|T\|$	Markings	Arcs	$\|B\|$	$\|E\|$	#c	$\|E\|-$#c	$\|B\|$	$\|E\|$	#c	$\|E\|-$#c
2	9	10	27	66	43	23	7	16	17	10	4	6
3	13	15	108	351	105	55	16	39	25	15	6	9
4	17	20	405	1620	225	116	30	86	33	20	8	12
5	21	25	1458	6885	453	231	50	181	41	25	10	15
6	25	30	5103	27702	887	449	77	372	49	30	12	18
7	29	35	17496	107163	1721	867	112	755	57	35	14	21

gorithms [19,10] can be exponentially larger than the compact finite complete prefix which we generated using semantic arguments. This construction relied on the special properties the family of net systems under discussion has. We don't know of a practical algorithm to always generate a polynomial prefix when it is allowed by the semantic notion of prefix completeness.

In the rest of this work we adopt the semantic definition of prefix completeness (Def. 1) as the only property a finite complete prefix has. Thus we can use purely semantic arguments, and do not have to consider the peculiarities of a fixed prefix generation algorithm. However, as presented by Table 1, sometimes the current algorithms will generate exponentially larger prefixes. Thus the complexity results we will present do not automatically apply to these prefixes.

4 Complexity of Model Checking with Complete Prefixes

We can see a 1-safe Petri net system as a representation of its (finite, interleaved) reachability graph. Thus in model checking a Petri net we actually interpret the model checking questions on its reachability graph. Because a finite complete prefix is a symbolic representation of the same reachability graph, we can do model checking when a finite complete prefix is given as input. We will now show that many model checking problems for finite complete prefixes of 1-safe

Petri nets are PSPACE-complete in the size of the prefix. This result has been first published in [13], where detailed proofs can be found.

The proof is based on the PSPACE-hardness proof of the reachability problem for 1-safe Petri nets by Jones, Landweber and Lien [15]. The proof involves simulating a Turing machine with a fixed number of tape cells with a 1-safe Petri net. Our proof is based on the variation of this proof by Esparza [6], from which most of the material of the following section is from. We first introduce this proof, because our proof is built on top of it in two steps.

4.1 Reachability with 1-Safe Petri Nets

We use slightly nonstandard notation in this work. We consider Turing machines with finite tape, i.e. they have both a first and a last cell on their tape. As in the standard definition, a move to the left of the first cell results in the machine staying on the first cell. Slightly nonstandard is the handling of the last cell. If the program of the Turing machine tries to move right when being on the last cell, it stays on the last cell. We define the notions of execution and acceptance of a Turing machine in what is in the essence a standard way, see e.g. [20], with only minor notational differences, for the details see [13].

A *Turing machine* is defined to be a tuple $M = \langle Q, \Gamma, \delta, q_0, F\rangle$, where Q is a finite set of states, Γ a finite set of tape symbols (containing a special *blank* symbol #), $\delta : (Q \times \Gamma) \mapsto \mathcal{P}(Q \times \Gamma \times \{R, L\})$ is the (non-deterministic) transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We define the *size of a Turing machine* to be the number of bits needed to encode its transition relation, i.e. $2 \cdot |Q|^2 \cdot |\Gamma|^2$.

We define a *linearly bounded automaton* to be a Turing machine which uses n tape cells for a Turing machine description of size n (i.e. the amount of tape matches the size of the transition relation). We encode the configuration of an automaton as $\langle q, i, w\rangle$, where q is the control state of the automaton, $i \in \{1, \ldots, n\}$ is the current location of the tape head, and $w \in \Gamma^n$ is a string of length n which gives the contents of the n tape cells of the automaton. We call a configuration $\langle q, i, w\rangle$ an *initial configuration* if $q = q_0$.

Many question about the computations of linearly bounded automata are PSPACE-hard. The first one we use is the *empty-tape acceptance problem*:

Definition 2. *EMPTY-TAPE ACCEPTANCE:*
Given a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F\rangle$, does A accept on the empty input?

It is well known that EMPTY-TAPE ACCEPTANCE is PSPACE-complete. Moreover, it remains PSPACE-complete even if we restrict the automaton A to have only one accepting state q_F, see e.g. [6]. We define the size of a 1-safe Petri net system $\Sigma = \langle S, T, F, M_0\rangle$ to be the number of bits needed to encode the flow relation F, i.e. $\mathcal{O}(|S| \cdot |T|)$. The result we use is the following theorem, first proved by Jones, Landweber and Lien [15]:

Theorem 1. *A linearly bounded automaton of size n can be simulated by a 1-safe Petri net system of size $\mathcal{O}(n^2)$. Moreover, there is a deterministic polynomial time procedure in the size of the automaton which constructs this net.*

We now introduce this mapping from a linearly bounded automaton to a 1-safe Petri net system $N(A)$. See Fig. 4 for an example when $|Q| = 3$, $n = 2$ (smaller than the real n to make the figure smaller), and $\Gamma = \{\#, a, b\}$.

Let $A = \langle Q, \Gamma, \delta, q_0, F\rangle$ be a linearly bounded automaton of size n. We denote the set of tape cells with $C = \{c_1, \ldots, c_n\}$. The simulating Petri net $N(A)$ contains a place $s(q)$ for each state $q \in Q$, a place $s(c_i)$ for each cell $c_i \in C$, and a place $s(a, c_i)$ for each pair $a \in \Gamma, c_i \in C$. A token on place $s(q)$ tells that the machine is in state q, a token on $s(c_i)$ marks the current head location, and when the place $s(a, c_i)$ is marked it means that the symbol on tape cell c_i is a.

The transitions of $N(A)$ are obtained from the transition function of A. If $s(q', a', R) \in \delta(q, a)$, then there exists for each cell $c \in C$ a corresponding transition $t(s(q, a, c)w(a')s(q', c'))$, whose input places are $s(q)$, $s(c)$, and $s(a, c)$, and whose output places are $s(q')$, $s(c')$, and $s(a', c)$, where c' is the cell to the right of c, except when c is the last cell, in which case $c' = c$. The left moves are handled similarly, except that now the first cell is an exception, moving left on it will leave the head on the leftmost cell. The initial marking of $N(A)$ has one token on $s(q_0)$, one on $s(c_1)$, and one on each of the places $s(\#, c_i)$, for all $i \in \{1, \ldots, n\}$. The total size of the net system $N(A)$ is $\mathcal{O}(n^2)$ [6].

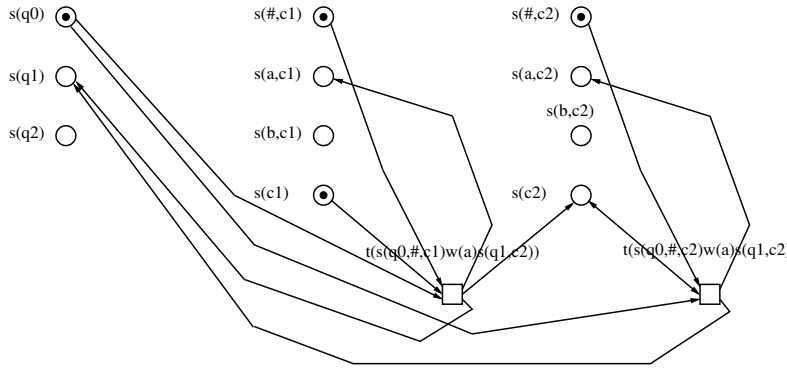


Fig. 4. A part of $N(A)$ simulating a transition $(q_1, a, R) \in \delta(q_0, \#)$

In this work we use *polynomial-time many-one reductions* (i.e. Karp reductions). Thus given a linearly bounded automaton A with a unique accepting state q_F, we can in deterministic polynomial time construct $N(A)$. Now to decide EMPTY-TAPE ACCEPTANCE we need to answer the following problem on $N(A)$: Is there a reachable marking M of $N(A)$, such that $M(s(q_F)) = 1$?

It is easy to see from the semantics of the branching time temporal logic CTL [3], that the question above is equivalent to the CTL model checking ques-

tion: $N(A) \models EF(s(q_F))$, i.e. does the formula $EF(s(q_F))$ hold on the interleaved reachability graph of $N(A)$? (We use place names as atomic propositions, e.g. $s(q_F)$ is true in a marking M iff $M(s(q_F)) = 1$.) Thus CTL model checking is PSPACE-hard in the size of the net system $N(A)$. The model checking problem is also in PSPACE in the size of the 1-safe net system for CTL [6].

4.2 Another PSPACE-Hardness Proof with 1-Safe Petri Nets

We present an alternative PSPACE-hardness proof for CTL model checking with 1-safe Petri nets. This proof was created to make our proof about prefix model checking complexity (to be presented in the next section) easier.

We use another PSPACE-complete problem for linearly bounded automata:

Definition 3. *ARBITRARY-TAPE-STATE ACCEPTANCE:*
Given a linearly bounded automaton $A = \langle Q, \Gamma, \delta, q_0, F\rangle$ with unique accepting state q_F, does there exists an initial configuration on which A accepts?

In other words: Does there exist an accepting execution of A starting from some initial configuration $\langle q_0, i, w\rangle$, where $i \in \{1, \ldots, n\}$ and $w \in \Gamma^n$?

Theorem 2. *ARBITRARY-TAPE-STATE ACCEPTANCE is PSPACE-complete.*

Proof. See [13]. □

Given a linearly bounded automaton A we will now reduce the problem ARBITRARY-TAPE-STATE ACCEPTANCE into the problem of model checking a certain fixed CTL-formula ϕ on a 1-safe net system $C(A)$. The main intuition behind the reduction is that $C(A)$ is a "cheating simulation" of A, namely it has also behaviors which do not correspond to a simulation of an execution of A. The formula ϕ takes care of ignoring the cheating runs of the net system. We construct a 1-safe Petri net, which first "randomly" initializes the system into some initial state, and then starts to simulate the behavior of the automaton A.

We use the net system $N(A)$ as defined in the previous section as the basis of our mapping, add some places and transitions, and change the initial marking to create a net system $C(A)$ (for details, see [13]). See Fig. 5 for an example of the initialization and simulation of the same transition as in Fig. 4. The places $s(nq), s(nc)$, and $s(nc_i)$ for all $i \in \{1, \ldots, n\}$ are new. They are used to mark that the control state, head location, or contents of the tape cell c_i has not been initialized yet, respectively. For each state $q_i \in Q$ there exists a new transition $t(nq_i)$ whose preset is $s(nq)$ and whose postset is $s(q_i)$. For each tape cell $c_i \in C$ there exists a new transition $t(nc_i)$ whose preset is $s(nc)$ and whose postset is $s(c_i)$. Also for each pair (a, c_i), such that $a \in \Gamma, c_i \in C$, there exists a new transition $t(n(a, c_i))$ whose preset is $s(nc_i)$ and whose postset is $s(a, c_i)$. The initial marking is changed to have a token on the new places added to $C(A)$, and no tokens on other places. This denotes the fact that the initialization needs to be done for state, head location, and each tape cell.

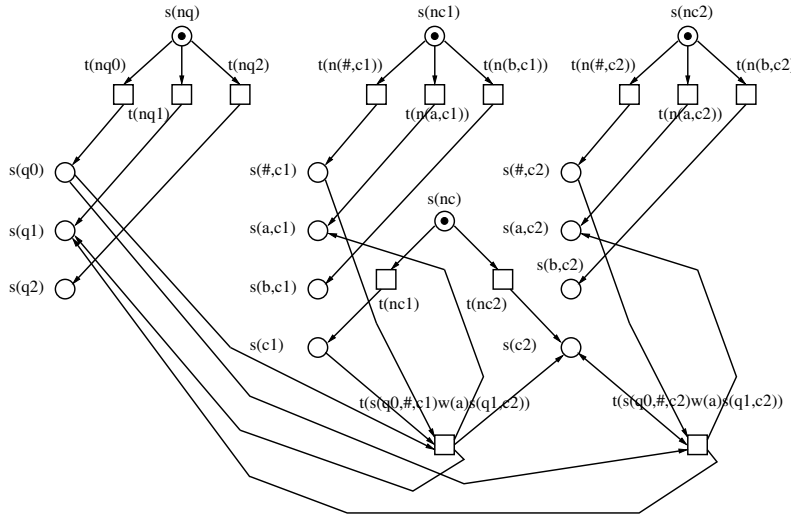


Fig. 5. A cheating Turing machine simulator

Note that we are even initializing the simulation initial state randomly, instead of initializing it to the state q_0. Thus our simulator is a cheating one. Also note that the initialization and the beginning of the simulation are not synchronized. This is needed for the prefix to be created to be a compact one, however, it somewhat complicates the proofs in [13].

Lemma 1. *The net system $C(A)$ is 1-safe.*

Proof. The net system $C(A)$ has the following marking invariants:

- $M(s(nq)) + \sum_{q_i \in Q} M(s(q_i)) = 1$,
- $M(s(nc)) + \sum_{c_i \in C} M(s(c_i)) = 1$, and
- for all $i \in \{1, \ldots, n\}$: $M(s(nc_i)) + \sum_{a \in \Gamma} M(s(a, c_i)) = 1$.

The invariants cover all the places of the net system $C(A)$, thus it is 1-safe. □

Now we can show PSPACE-completeness by model checking the CTL formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$ on the net system $C(A)$.

Lemma 2. *Let A be a linearly bounded automaton with a unique accepting state q_F. It holds that $C(A) \models EF(s(q_0) \wedge EF(s(q_F)))$ iff A has an accepting execution starting from some initial configuration of A.*

Proof. The idea of the proof in one direction is to take an accepting execution of A, and transform it to an execution of $C(A)$, which first fires $n+2$ initialization transitions and then starts simulating the execution of A, giving a witness for the formula ϕ. The other direction is a bit more involved. Whenever $C(A)$ has an execution which is a witness of ϕ, it actually also has an execution which first

fires $n + 2$ initialization transitions, and then starts simulating (an accepting execution of) A. Proving this requires a number of lemmas about (a particular kind of) commutativity between the initialization and simulation transitions of $C(A)$. For more details, see [13]. □

Theorem 3. *Model checking a fixed size CTL formula ϕ is PSPACE-complete in the size of the 1-safe net Σ.*

Proof. To show PSPACE-hardness we use the Lemma 2 to reduce the problem ARBITRARY-TAPE-STATE ACCEPTANCE to the problem of CTL model checking a fixed size formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$ on the net system $C(A)$. The size of $C(A)$ is $\mathcal{O}(n^2)$, i.e. polynomial in the size of A, and the reduction can be done in deterministic polynomial time.

The problem is in PSPACE by Lemma 1, combined with the proof of CTL model checking being in PSPACE in the size of 1-safe net system, see e.g. [6]. □

4.3 Model Checking with Finite Complete Prefixes

We will now make use of the machinery created in the previous sections. We will prove model checking complexity results for algorithms which are given a finite complete prefix of a 1-safe Petri net as the input.

We use the net system $C(A)$ as our starting point, and define the prefix $\beta_C(A)$ to be identical to the first two levels of the unfolding of $C(A)$ (see [13] for the formal definition). For an example of the prefix, see Fig. 6, which is the prefix of the net system in Fig. 5. The prefix $\beta_C(A)$ contains exactly as many

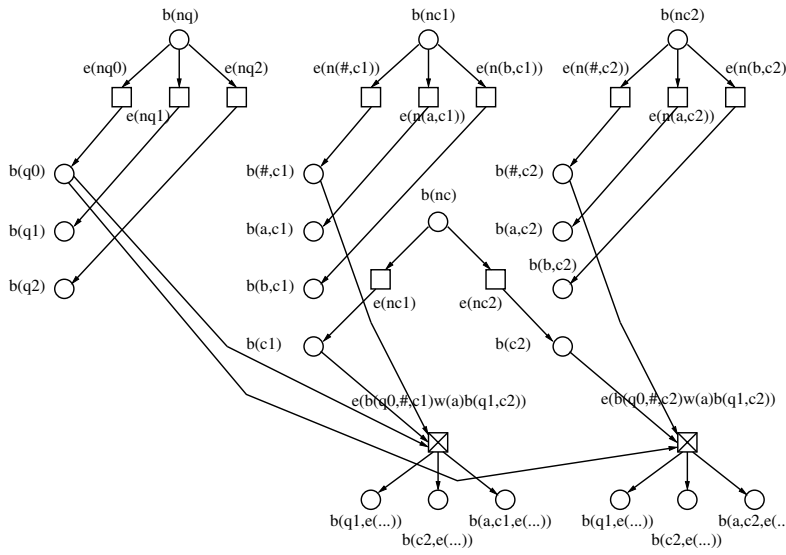


Fig. 6. A finite complete prefix of the cheating Turing machine simulator

events as there are transitions in $C(A)$. Only the conditions in the postsets of the transitions corresponding to the simulation transitions are new, and there are at most $6 \cdot |Q|^2 \cdot |\Gamma|^2$ of them. Therefore the size of the prefix $\beta_C(A)$ is polynomial in the size of $C(A)$ (and thus also in size of A).

Lemma 3. *The prefix $\beta_C(A)$ is a finite complete prefix of the net system $C(A)$.*

Proof. See [13]. □

Now we can present the main result of this work.

Theorem 4. *Model checking a fixed size CTL formula ϕ is PSPACE-complete in the size of a finite complete prefix β of a 1-safe net Σ.*

Proof. See [13] for details. To show PSPACE-hardness we use the reduction used in the proof of Theorem 3, and then reduce this problem further to CTL model checking ϕ with $\beta_C(A)$ by creating $\beta_C(A)$ in deterministic polynomial time from $C(A)$. Thus by Lemma 3 we get the PSPACE-hardness result.

To show that the problem is in PSPACE in the size of the prefix we use the fact that given a prefix β of a 1-safe net system Σ, and a formula ϕ, we can in polynomial space construct a net system Σ' (by folding the acyclic prefix back into a cyclic net system in the labelling respecting way). For this net system it holds that $\Sigma' \models \phi$ iff $\Sigma \models \phi$. Then we CTL model check $\Sigma' \models \phi$ in PSPACE [6] for a total complexity of PSPACE. □

The CTL formula $\phi = EF(s(q_0) \wedge EF(s(q_F)))$ syntactically belongs to all the logics UB$^-$, UB, CTL, and CTL* (see [3], for the UB logics see e.g. [8]). Therefore the PSPACE-hardness result also applies to them.

We will now require without loss of generality that all executions of the automaton A entering the final state q_F will keep on looping back to the final state q_F thus creating an infinite execution in which the final state is repeated.

We can then create the linear temporal logic LTL (see [3]) formula $\psi = \Box(\neg(s(q_0)) \vee \Box(\neg(s(q_F))))$. Now it is easy to see from the semantics of LTL that $C(A) \models \phi$ iff $C(A) \not\models \psi$. A violation of this LTL formula can be expressed by a Büchi automaton, which can be translated into an equivalent linear-time μ-calculus formula (see e.g. [2]). The LTL formula ψ is also a syntactic safety formula, and thus a violation of this property can also be expressed by a deterministic finite automaton [17]. Thus we get a PSPACE-hardness result for LTL model checking, Büchi emptiness checking, linear-time μ-calculus model checking, and safety model checking.

The model checking problems mentioned above are in PSPACE in the size of the 1-safe net system, and thus we can use the proof of Theorem 4 also with them (see [6], for CTL* we create a concurrent program from a 1-safe Petri net in deterministic polynomial time, and then use a similar result presented for concurrent programs in e.g. [16]). Therefore these model checking problems are PSPACE-complete in the size of a finite complete prefix of a 1-safe Petri net.

5 Conclusions

We have shown that model checking a fixed size formula of several temporal logics, including LTL, CTL, and CTL*, is PSPACE-complete in the size of a finite complete prefix of a 1-safe Petri net. This is to be contrasted with the reachability problem, in which a PSPACE-complete problem for 1-safe Petri nets is transformed by the prefix generation process into (a potentially exponentially larger) NP-complete problem, see e.g. [13]. However, such a drop in complexity (assuming NP is easier than PSPACE) does not occur in the case of model checking involving nested temporal modalities. Thus, loosely speaking, with prefixes reachability is easier than "repeated reachability" (see [13]).

Our proof employs a class of 1-safe Petri nets for which it is easy to create a finite complete prefix by using semantic arguments. We have shown that sometimes the prefixes created by current prefix generation algorithms [10] will be exponentially larger than allowed by the semantic completeness criterion (Def. 1). The definition of a suitable prefix minimality criterion, and the creation of a procedure to always obtain these compact prefixes is left for further work.

There are prefix based model checkers which handle nested temporal modalities. The LTL model checker of [21] creates a certain graph, whose size can be exponential in the size of the prefix. The construction employed by the branching time model checker of [5,12] to handle nested temporal modalities is more involved, and relating our work to the results of [12] is left for further study. We would also like to know whether the prefixes generated by [10] have some properties which would allow simpler model checking algorithms than the prefixes fulfilling only the semantic prefix completeness criterion. Finally, for LTL model checking we can change the model checker to take both the net system, and the LTL(-X) formula ψ as input to the prefix generation process. In this approach also the semantic prefix completeness criterion is parameterized by the checked formula, and the model checking can be done in polynomial time in the size of this "product" prefix [7]. The price to pay is a larger prefix. A simpler product method works with safety model checking.

Acknowledgements

Part of this work was done during a visit to Prof. J. Esparza's research group at TU München. The author would like to thank for the visit opportunity, and for discussions about the unfolding method. The author would also like to thank Ilkka Niemelä and Tommi Junttila for critical comments on this work.

References

1. E. Best. Partial order verification with PEP. In *Proceedings of POMIV'96, Workshop on Partial Order Methods in Verification*. American Mathematical Society, July 1996. 114
2. M. Dam. Fixpoints of Büchi automata. In *Proceedings of the 12th International Conference of Foundations of Software Technology and Theoretical Computer Science*, pages 39–50, 1992. LNCS 652. 120

3. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990. 108, 116, 120
4. J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991. 110, 111
5. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994. 121
6. J. Esparza. Decidability and complexity of Petri net problems – An introduction. In *Lectures on Petri Nets I: Basic Models*, pages 374–428. Springer-Verlag, 1998. LNCS 1491. 109, 115, 116, 117, 119, 120
7. J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. In *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'2000)*, July 2000. Accepted for publication. 121
8. J. Esparza and M. Nielsen. Decidability issues for Petri Nets - a survey. *Journal of Information Processing and Cybernetics 30(3)*, pages 143–160, 1994. 120
9. J. Esparza and S. Römer. An unfolding algorithm for synchronous products of transition systems. In *Proceedings of the 10th International Conference on Concurrency Theory (Concur'99)*, pages 2–20. Springer-Verlag, 1999. LNCS 1664. 108, 111
10. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 87–106. Springer-Verlag, March 1996. LNCS 1055. 108, 109, 111, 112, 114, 121
11. J. Feigenbaum, S. Kannan, M. Y. Vardi, and M. Viswanathan. Complexity of problems on graphs represented as OBDDs. *Chigago Journal of Theoretical Computer Science*, 1999(5):1–25, 1999. 108
12. B. Graves. Computing reachability properties hidden in finite net unfoldings. In *Proceedings of 17th Foundations of Software Technology and Theoretical Computer Science Conference*, pages 327–341. Springer-Verlag, 1997. LNCS 1346. 121
13. K. Heljanko. Deadlock and reachability checking with finite complete prefixes. Research Report A56, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999. Licentiate's Thesis. Available at http://www.tcs.hut.fi/pub/reports/A56.ps.gz. 108, 115, 117, 118, 119, 120, 121
14. K. Heljanko. Minimizing finite complete prefixes. In *Proceedings of the Workshop Concurrency, Specification & Programming 1999*, pages 83–95. Warsaw University, Warsaw, Poland, September 1999. 114
15. N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4:277–299, 1977. 115
16. O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, Technion, Israel Institute of Technology, Haifa, Israel, June 1995. 120
17. O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proceeding of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 172–183. Springer-Verlag, 1999. LNCS 1633. 120
18. R. Langerak and E. Brinksma. A complete finite prefix for process algebra. In *Proceeding of 11th International Conference on Computer Aided Verification (CAV'99)*, pages 184–195. Spriger-Verlag, 1999. LNCS 1633. 108
19. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 108, 111, 112, 114
20. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 115
21. F. Wallner. Model checking techniques using net unfoldings. PhD thesis, Technische Universität München, Germany, forthcoming. 121

Verifying Quantitative Properties of Continuous Probabilistic Timed Automata*

Marta Kwiatkowska[1], Gethin Norman[1], Roberto Segala[2], and Jeremy Sproston[2]

[1] University of Birmingham
Birmingham B15 2TT, United Kingdom
{M.Z.Kwiatkowska,G.Norman,J.Sproston}@cs.bham.ac.uk
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
segala@cs.unibo.it

Abstract. We consider the problem of automatically verifying real-time systems with *continuously* distributed random delays. We generalise *probabilistic timed automata* introduced in [19], an extension of the timed automata model of [4], with clock resets made according to continuous probability distributions. Thus, our model exhibits nondeterministic and probabilistic choice, the latter being made according to both discrete and continuous probability distributions. To facilitate algorithmic verification, we modify the standard region graph construction by subdividing the unit intervals in order to *approximate* the probability to within an interval. We then develop a model checking method for continuous probabilistic timed automata, taking as our specification language Probabilistic Timed Computation Tree Logic (PTCTL). Our method improves on the previously known techniques in that it allows the verification of *quantitative* probability bounds, as opposed to qualitative properties which can only refer to bounds of probability 0 or 1.

1 Introduction

Background: In recent years we have witnessed an increase in demand for formal models and verification techniques for real-time systems such as communication protocols, digital circuits with uncertain delay lengths, and media synchronization protocols. Automatic verification of quantitative timing constraints has particularly been subject to significant attention, as indicated by the development of associated software tools [9,11] and their successful application in industrial case studies.

Traditional approaches to real-time systems define behaviour purely in terms of non-determinism. However, it may be desirable to express the relative *likelihood* of the occurrence of certain behaviour. For example, we may wish to model a system in which an event is triggered after a random, *continuously distributed*

* Supported in part by EPSRC grants GR/M04617, GR/M13046 and GR/N22960.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 123–137, 2000.

delay, where the distribution may be e.g. uniform, normal, or exponential. Such a framework would be particularly useful when modelling environments with unpredictable behaviour; for instance, those involving component failure or customer arrivals in a network. Furthermore, we may also wish to refer to the likelihood of certain temporal logic properties being satisfied by such a real-time system, and to have a model checking algorithm for verifying the truth of these assertions. The remit of this paper is to address these issues.

To provide an appropriate foundation for our work, we take the model of *timed automata* [4], a framework for modelling non-deterministic real-time systems and a focus of much attention from both theoretical researchers and verification practitioners alike. A timed automaton takes the form of a finite directed graph equipped with a set of variables referred to as *clocks*. Since clocks are real-valued, the state space of a timed automaton is infinite, and hence verification is performed by constructing a finite-state quotient of the system model, called a *region graph* [3], which is then subject to established model checking techniques. Recently, we have shown that this region graph construction can also be applied to timed automata augmented with *discrete* probability distributions [19]. This result provides a method for verifying such probabilistic timed automata against PTCTL. This result relies on the fact that all of the states encoded into a single region satisfy the same formulae. However, if our system model admits continuously distributed random delays, the latter property does not hold.

Motivating example: We illustrate why the region graph approach does not work with continuously distributed random delays by means of the example below due to Rajeev Alur [1]. Suppose in state s, at time $t = 0$, a clock d is set to values in the interval $(0, 1)$ according to some continuous density function. Now suppose that, at time $t < 1$, a transition occurs to state s' where d remains scheduled and another clock d' is newly scheduled, again set to values in the interval $(0, 1)$. Consider the three possible relationships between the clocks d and d' in state s':

$$(1)\ d' < d \qquad (2)\ d' = d \qquad (3)\ d' > d$$

The region-based approach only encodes the information that: (2) has probability 0, while (1) and (3) both have positive probability. However, the actual probabilities *depend on the value* of t (when the transition from s to s' is made).

Therefore, to perform any *exact* probability calculations with respect to continuous probabilistic real-time systems, we will require an *infinite* model, since, as can be seen in the example above, for each different value of t the probabilities of (1) and (3) will differ. On the other hand, a *finite* model is required for decidable automatic verification.

Main contribution: Our key proposal is to *refine* the region graph construction by subdividing the equivalence classes of clock values. In particular, we split the unit intervals into n subintervals of equal size. Intuitively, this process increases the granularity of the partitioning in order for the region graph to more accurately retain information concerning the continuous random delays. However, note that, for finite n, this will constitute an *approximation* of the exact probability values involved; we supplement such approximate answers with

an estimate of the *error*. Further refinements into yet smaller intervals yields better approximations of the probability bounds.

The main technical challenges in our approach are threefold. Firstly, we must define the probability measure and σ-algebra of the infinite paths in the presence of continuously distributed delays and dense time. Secondly, we need to show how the refined region graph may be constructed, where the particular difficulty is due to not knowing the relative order of clocks which are set continuously at random. Thirdly, we have to estimate the error caused by the finite approximation.

Related work: There are many probabilistic verification frameworks, see e.g. [6,16,13,14], most of which only handle discrete probability and time. Our results relate to those of [2,12], which concern the model checking of probabilistic real-time systems with continuous random delays against qualitative properties that can only refer to probability bounds of 0 or 1. In [7], a quantitative model checking procedure for continuous time Markov chains is presented. Recently a method for approximating continuous Markov processes has been proposed in [15], but the relationship between their approach and ours is not yet known.

2 Preliminaries

Throughout, we use standard notation from timed automata, based on [3]. Labelled paths are non-empty finite or infinite sequences of the form: $\omega = \sigma_0 \xrightarrow{l_0} \sigma_1 \xrightarrow{l_1} \sigma_2 \xrightarrow{l_2} \cdots$ where σ_i are states and l_i are labels for transitions. The first state of ω is denoted by $\mathit{first}(\omega)$. If ω is finite then the last state of ω is denoted $\mathit{last}(\omega)$. The length of a path is defined in the standard way (∞ if the path is infinite) and is denoted $|\omega|$. The prefix relation on paths is denoted by $\leq$ and the concatenation by juxtaposition. If $k \in \mathbb{N}$ then $\omega(k)$ denotes the k-th state, $\mathit{step}(\omega, k)$ the label of the k-th step, and $\omega^{(k)}$ denotes the k-th prefix of ω.

We assume some familiarity with probability and measure theory, see e.g. [5]. Consider a set Ω. A *σ-field* on Ω, denoted $\mathcal{F}$, is a field closed under countable union. The elements of a σ-field are called the *measurable sets*, and $(\Omega, \mathcal{F})$ is called a *measurable space*. Let $(\Omega, \mathcal{F})$ be a measurable space. A function $P : \mathcal{F} \to [0, 1]$ is a *probability measure* on $(\Omega, \mathcal{F})$, and $\mathcal{P} = (\Omega, \mathcal{F}, P)$ a *probability space*, if P satisfies the following properties: $P(\Omega) = 1$, and if $A_1, A_2, \ldots$ is a disjoint sequence of elements of $\mathcal{F}$, then $P(\cup_i A_i) = \sum_i P(A_i)$.

A *continuous density function* (cdf) on $\mathbb{R}$ is a function f such that $f(x) \geq 0$ for all $x \in \mathbb{R}$ and $\int_{-\infty}^{+\infty} f(x)dx = 1$. Furthermore, f has *support* $A \subseteq \mathbb{R}$ if $f(x) = 0$ for all $x \in \mathbb{R} \setminus A$. We define a cdf f to be *positive bounded* if its support lies within a closed interval of $\mathbb{R}^{\geq 0}$. We denote by PB the set of positive bounded cdfs, and the set of *discrete probability distributions* over a (finite) set S by $\mu(S)$.

2.1 Dense Markov Processes

Definition 1. *A* dense Markov Process M *is a tuple* $(Q, \mathcal{F}, q_0, \{P_q\}_{q \in Q})$*, where* Q *is a set of* states*,* $\mathcal{F}$ *is a* σ*-field over* Q*,* q_0 *is the* initial state*, and each* P_q *is a probability measure on* $(Q, \mathcal{F})$*.*

Observe that we do not impose any limit on the cardinality of Q and that the probability spaces associates with the states are not necessarily discrete. For notational convenience we denote the dense Markov process $(Q, q, \mathcal{F}, \{P_{q'}\}_{q' \in Q})$ by M_q. Our objective is to define a probability space, $\mathcal{P}_{M_q} = (\Omega_{M_q}, \mathcal{F}_{M_q}, P_{M_q})$, for the infinite sequences of states that can be generated by a dense Markov process.

The sample set Ω_{M_q} is the set of infinite sequences qQ^ω of states starting in q. For the σ-field we generalise the cone construction for ordinary Markov processes. The generalisation of a cone is a set of sequences of Ω_{M_q} that extend some appropriate set of finite sequences. Formally, given a dense Markov process M_q and a finite sequence $X_1 X_2 \dots X_k$ of elements of $\mathcal{F}$, the set of sequences

$$B_{X_1 X_2 \dots X_k} = \{qq_1 q_2 \dots q_k \alpha \mid \alpha \in Q^\omega \text{ and } q_i \in X_i \text{ for all } 1 \leq i \leq k\}$$

is called a *basic set*. Special basic sets are $B_\epsilon = \Omega_{Mq}$ (ϵ denotes the empty sequence) and $B_\perp = \emptyset$. We use β to range over sequences of elements of $\mathcal{F}$.

The next step is to assign a measure to basic sets. It turns out that we cannot assign a measure to all basic sets in general. We define basic measurable sets together with their measures by induction.

Definition 2 (Basic measurable sets). *The basic sets $B_\perp$ and B_ϵ are measurable. The measure of $B_\perp$ is $P_{M_q}[B_\perp] = 0$ and the measure of B_ϵ is $P_{M_q}[B_\epsilon] = 1$.*

A basic set $B_{X\beta}$ is a basic measurable set if

1. *B_β is a basic measurable set; and*
2. *the function $f_{X\beta}$ that maps the state q to $P_{M_q}[B_{X\beta}]$ is measurable from $(Q, \mathcal{F})$ to the Borel σ-field over the interval $[0, 1]$, where $P_{M_q}[B_{X\beta}]$ is defined to be $\int_Q f_\beta I_X dP_q$ and I_X denotes the indicator function of X.*

Note that in the integral above f_β is measurable because B_β is a basic measurable set, and I_X is measurable because $X \in \mathcal{F}$. Thus the above integral is well defined by [5, Theorem 1.5.9].

Following an argument similar to [20], we can show that the measure P_{M_q} is σ-additive on the basic measurable sets. If all basic sets are basic measurable, then we can generate the minimum field that contains the basic measurable sets and show that there is a unique extension of the measure P_{M_q}. Thus, we can simply define the σ-field $\mathcal{F}_{M_q}$ to be the σ-field generated by the basic measurable sets, and extend the measure P_{M_q} using [5, Theorem 1.3.6].

3 Definition of the Model

Let AP be a set of atomic propositions. A *clock* x is a non-negative real-valued variable which increases at the same rate as real-time. Let $\mathcal{X}$ be a set of clocks, and let $\nu : \mathcal{X} \to \mathbb{R}^{\geq 0}$ be a function assigning a non-negative real value to each of the clocks in this set. Such a function is called a *clock assignment*. For any

$X \subseteq \mathcal{X}$ and $t \in \mathbb{R}^{\geq 0}$, we write $\nu[X := t]$ for the clock assignment that assigns t to all clocks in X, and agrees with ν for all clocks in $\mathcal{X} \setminus X$. In addition, $\nu + t$ denotes the clock assignment for which all clocks x in $\mathcal{X}$ take the value $\nu(x) + t$.

As with standard timed automata [4,17], the conditions for progress between nodes of the graph are described in terms of *clock constraints*.

Definition 3 (Clock Constraints). *Let X be a set of clocks. The set of* clock constraints *of clocks in X, $\mathcal{C}_X$, is defined inductively by the syntax:*

$$\zeta ::= x \leq k \,|\, x \geq k \,|\, x - y \leq k \,|\, x - y \geq k \,|\, \neg\zeta \,|\, \zeta \vee \zeta,$$

where $x, y \in X$ and $k \in \mathbb{N}$.

We say that a clock assignment ν *satisfies* the clock constraint ζ (also denoted $\nu \models \zeta$) if substitution of each $x \in \mathcal{X}$ by $\nu(x)$ results in ζ resolving to true.

Definition 4 (Continuous Probabilistic Timed Automaton). *A continuous probabilistic timed automaton is a tuple $G = (\mathcal{S}, \bar{s}, L, \mathcal{X}, dens, inv, prob, \langle\tau_s\rangle_{s\in\mathcal{S}})$ consisting of*

- *a finite set $\mathcal{S}$ of nodes, including a initial node $\bar{s}$;*
- *a function $L : \mathcal{S} \to 2^{\mathrm{AP}}$ assigning to each node of the graph the set of atomic propositions that are true in that node;*
- *a finite set $\mathcal{X}$ of clocks;*
- *a partial function $dens : \mathcal{S} \times \mathcal{X} \to PB$ assigning to pairs of nodes and clocks a positive bounded density function;*
- *a function $inv : \mathcal{S} \to \mathcal{C}_{\mathcal{X}}$ assigning to each node an invariant condition;*
- *a function $prob : \mathcal{S} \to \mathcal{P}_{fn}(\mu(\mathcal{S}))$, assigning to each node a (finite non-empty) set of discrete probability distributions on $\mathcal{S}$;*
- *a family of functions $\langle\tau_s\rangle_{s\in\mathcal{S}}$ where, for any $s \in \mathcal{S}$, $\tau_s : prob(s) \to \mathcal{C}_{\mathcal{X}}$ assigns an enabling condition to each discrete probability distribution in $prob(s)$.*

Continuous probabilistic timed automata generalise the probabilistic timed automata of [19] through the addition of the partial function *dens*. Whenever defined for a node s and a clock x, this function yields a cdf, say f, which captures the *reset* of x upon entry into s. Such a reset results in a *random* assignment to x (for f a general cdf in PB) or an assignment of an *exact value* (if f is a point distribution). If *dens* is undefined, the clocks keep their old values, as in [2].

The behaviour of the model is as follows. It starts in node $\bar{s}$ with all clocks in $\mathcal{X}$ initialized to 0. The values of all the clocks increase uniformly with time. At any point in time, if the system is in node s then it can behave in one of two ways depending on the values of the clocks in $\mathcal{X}$. It can either let *time advance* such that the invariant condition $inv(s)$ does not become violated, or make a *state transition*, subject to certain conditions given below. State transitions are instantaneous, and generalise the state transitions of the (discrete-)probabilistic timed automata of [19] in the following sense:

- a distribution $p_s \in prob(s)$, whose corresponding enabling condition $\tau_s(p_s)$ is satisfied by the current values of the clocks, is selected *nondeterministically*;

– then, supposing p_s is chosen, for any $s' \in \mathcal{S}$, with probability $p_s(s')$ the system will make a transition to node s' and reassign values to all the clocks x for which $dens(s', x)$ is defined according to the cdfs given by $dens(s', x)$.

For notational convenience, for each node $s \in \mathcal{S}$, we denote by $O(s)$ the set of clocks x for which $dens(s, x)$ is not defined, i.e. those clocks that keep their old value when s is reached, and by $N(s)$ the set of clocks x for which $dens(s, x)$ is defined, i.e. those clocks that are reassigned a new value when s is reached.

Let G be a continuous probabilistic timed automaton. We now define formally the behaviour of G as a probabilistic timed structure. We let $\Gamma(G)$ denote the set of all clock assignments for all the clocks in $\mathcal{X}$.

Definition 5 (State). *A* state *of G is a pair $\langle s, \nu \rangle$ where $s \in \mathcal{S}$, $\nu \in \Gamma(G)$ such that $inv(s)$ is satisfied by ν.*

Definition 6 (Path). *A path of G is an infinite or finite sequence*

$$\omega = \langle s_0, \nu_0 \rangle \xrightarrow{t_0, p_0} \langle s_1, \nu_1 \rangle \xrightarrow{t_1, p_1} \langle s_2, \nu_2 \rangle \xrightarrow{t_2, p_2} \cdots$$

such that, for each $i \in \mathbb{N}$:

1. *$s_i \in \mathcal{S}$, $\nu_i \in \Gamma(G)$, $t_i \in \mathbb{R}^{\geq 0}$ and $p_i \in prob(s_i)$;*
2. *the invariant condition $inv(s_i)$ is satisfied by $(\nu_i + t)$ for all $0 \leq t \leq t_i$;*
3. *the clock assignment $(\nu_i + t_i)$ satisfies $\tau_{s_i}(p_i)$;*
4. *$p_i(s_{i+1}) > 0$, $\nu_{i+1}(x)$ is in the support of $dens(s_{i+1}, x)$ for all $x \in N(s_{i+1})$ and $\nu_{i+1}(x) = \nu_i(x) + t_i$ for all $x \in O(s_{i+1})$.*

For all $0 \leq i \leq |\omega|$, define $\mathcal{T}_\omega(i)$, the elapsed time *until the i^{th} transition, as follows: put $\mathcal{T}_\omega(0) = 0$, and for any $1 \leq i \leq |\omega|$, let $\mathcal{T}_\omega(i) = \sum_{k=0}^{i-1} t_k$.*

Consider an infinite path ω of G. A *position* of ω is a pair (i, t'), where $i \in \mathbb{N}$ and $t' \in \mathbb{R}$ such that $0 \leq t' \leq t_i$. The *state at position* (i, t'), denoted by $\omega(i + t')$, is given by $\langle s_i, \nu_i + t' \rangle$. Given a path ω, $i, j \in \mathbb{N}$ and $t, t' \in \mathbb{R}$ such that $t \leq t_i$ and $t' \leq t_j$, then we say that the position (j, t') *precedes* the position (i, t), written $(j, t') \prec (i, t)$, if and only if $j < i$, or $j = i$ and $t' < t$.

For simplicity, as in [19], we add time successor transitions from each state of the probabilistic timed structure determined by G in which time may diverge. We omit the details of this approach to time divergence from this extended abstract.

Due to the presence of both non-deterministic and probabilistic choice, we use the notion of an adversary, based on e.g. [8]. The role of an adversary is to select, for each finite path of G, the time point t and one of the probability distributions p enabled in the last state of the path.

Definition 7 (Adversary of G). *An* adversary *(or scheduler) of G is a function A mapping every finite path ω of G to a pair (t, p) where $t \in \mathbb{R}^{\geq 0}$ and $p \in \mu(\mathcal{S})$ such that if $last(\omega) = \langle s, \nu \rangle$ then $p \in prob(s)$, $\nu + t'$ satisfies $inv(s)$ for all $0 \leq t' \leq t$, and $\nu + t$ satisfies $\tau_s(p)$.*

For an adversary A of a continuous probabilistic timed automaton G we define the following sets of paths: $Path^A_{fin}\langle s,\nu\rangle$ ($Path^A_{ful}\langle s,\nu\rangle$) is the set of finite (infinite) paths such that $step(\omega,i) = A(\omega^{(i)})$ for all $1 \leq i < |\omega|$ and $first(\omega) = \langle s,\nu\rangle$.

We now turn to the definition of a probability space over the set of infinite paths $Path^A_{ful}\langle s,\nu\rangle$ of a given adversary A and state $\langle s,\nu\rangle$. When we use an adversary to resolve the nondeterminism we obtain a dense Markov process, whose path space is defined in Section 2.1. Each element of the sample set is an infinite chain of paths; however, it is easy to see that such chains can be replaced by their limits under prefix, i.e. an infinite path. We denote by $\mathcal{P}_{A,\langle s,\nu\rangle}$ the probability space over $Path^A_{ful}\langle s,\nu\rangle$.

Below we give an idea of how an adversary A and state $\langle s,\nu\rangle$ generate a dense Markov process $(Q,\mathcal{F},q_0,\{P_q\}_{q\in Q})$. The set of states Q is the set $Path^A_{fin}\langle s,\nu\rangle$ and the initial state q_0 is $\langle s,\nu\rangle$. The σ-field $\mathcal{F}$ is the σ-field generated by all the sets of the form

$$C_{q,s',\mathcal{I}} = \{q \xrightarrow{t,p} \langle s',\nu'\rangle \in Q \mid \nu'(x) \in \mathcal{I}(x) \text{ for all } x \in N(s')\},$$

where $q \in Q$, $s' \in \mathcal{S}$, $A(q) = (t,p)$ and $\mathcal{I}$ denotes a function mapping clocks to closed intervals. Finally, if $q \in Q$ and $A(q) = (t,p)$, then P_q is defined on the sets $C_{q',s',\mathcal{I}}$ as follows:

$$P_q(C_{q',s',\mathcal{I}}) = \begin{cases} p(s') \cdot \left(\prod_{x\in N(s')} \int_{\mathcal{I}(x)} dens(s',x)\,dx \right) & \text{if } q = q' \\ 0 & \text{otherwise} \end{cases}$$

and then extended to $\mathcal{F}$ using [5, Theorem 1.3.6].

4 Probabilistic Timed Computation Tree Logic (PTCTL)

We now introduce Probabilistic Timed Computation Tree Logic (PTCTL) as our logic for the specification of properties of probabilistic timed automata. Before we can define our logic formally, we will need to appropriately restrict the notion of an adversary of a continuous probabilistic timed automaton G. Due to the unlimited power that we have given to an adversary, it is easy to provide adversaries that would not guarantee the measurability of trivial events such as the occurrence of a single action. On the other hand, such adversaries would be extremely unnatural, and therefore we think it reasonable to rule out such pathological adversaries by definition. Thus, for the rest of this paper, we consider only *feasible* adversaries, that is, those that ensure the *measurability* of all the events identified by PTCTL formulae.

A further restriction on adversaries that we shall require is that of *time-divergence*; it is commonly imposed in real-time systems so that unrealisable behaviour (i.e. corresponding to time not advancing beyond a time bound) is disregarded during analysis. We say that an infinite path ω is *divergent* if for any $t \in \mathbb{R}^{\geq 0}$, there exists $j \in \mathbb{N}$ such that $\mathcal{T}_\omega(j) > t$.

Definition 8 (Divergent adversary). *An adversary A for a continuous probabilistic timed automaton G is* divergent *if and only if for each state $\langle s, \nu\rangle$ of G the probability $P_{A,\langle s,\nu\rangle}$ of the divergent paths of $Path^A_{ful}\langle s, \nu\rangle$ is 1. Let $\mathcal{A}_G$ denote the set of all divergent adversaries of G.*

We now define the syntax and semantics of PTCTL. We have omitted the treatment of reset quantifiers and clock constraints, the addition of which is straightforward, see [19].

Definition 9 (Syntax of PTCTL**).** *The syntax of* PTCTL *is defined as follows:*

$$\phi ::= \texttt{true} \;\mid\; a \;\mid\; \phi \wedge \phi \;\mid\; \neg\phi \;\mid\; [\phi\, \exists\, \mathcal{U}_{\sim k}\, \phi]_{\sqsupseteq\delta} \;\mid\; [\phi\, \forall\, \mathcal{U}_{\sim k}\, \phi]_{\sqsupseteq\delta}$$

where $a \in$ AP, $k \in \mathbb{N}$, $\delta \in [0,1]$, $\sim \in \{\leq, <, \geq, >\}$, and $\sqsupseteq$ is either $\geq$ or $>$.

Definition 10 (Satisfaction Relation). *For any continuous probabilistic timed automaton G, state $\langle s, \nu\rangle$ of G, set of divergent adversaries $\mathcal{A}_G$ of G, and* PTCTL *formula ϕ, the satisfaction relation $\langle s, \nu\rangle \models_{\mathcal{A}_G} \phi$ is defined inductively as follows:*

$$\begin{array}{ll}
\langle s,\nu\rangle \models_{\mathcal{A}_G} \texttt{true} & \text{for all } s \in \mathcal{S} \text{ and } \nu \in \Gamma(G) \\
\langle s,\nu\rangle \models_{\mathcal{A}_G} a & \Leftrightarrow a \in L(s) \\
\langle s,\nu\rangle \models_{\mathcal{A}_G} \phi_1 \wedge \phi_2 & \Leftrightarrow \langle s,\nu\rangle \models_{\mathcal{A}_G} \phi_i \text{ for all } i \in \{1,2\} \\
\langle s,\nu\rangle \models_{\mathcal{A}_G} \neg\phi & \Leftrightarrow s \not\models_{\mathcal{A}_G} \phi \\
\langle s,\nu\rangle \models_{\mathcal{A}_G} [\phi_1\, \exists\, \mathcal{U}_{\sim k}\, \phi_2]_{\sqsupseteq\delta} & \Leftrightarrow P_{A,\langle s,\nu\rangle}\{\omega \in Path^A_{ful}\langle s,\nu\rangle | \omega \models_{\mathcal{A}_G} \phi_1\, \mathcal{U}_{\sim k}\, \phi_2\} \sqsupseteq \delta \\
 & \quad \text{for some } A \in \mathcal{A}_G \\
\langle s,\nu\rangle \models_{\mathcal{A}_G} [\phi_1\, \forall\, \mathcal{U}_{\sim k}\, \phi_2]_{\sqsupseteq\delta} & \Leftrightarrow P_{A,\langle s,\nu\rangle}\{\omega \in Path^A_{ful}\langle s,\nu\rangle | \omega \models_{\mathcal{A}_G} \phi_1\, \mathcal{U}_{\sim k}\, \phi_2\} \sqsupseteq \delta \\
 & \quad \text{for all } A \in \mathcal{A}_G \\
\omega \models_{\mathcal{A}_G} \phi_1\, \mathcal{U}_{\sim k}\, \phi_2 & \Leftrightarrow \text{there exists a position } (i,t) \text{ of } \omega \text{ such that} \\
 & \quad \mathcal{T}_\omega(i) + t \sim k,\ \omega(i+t) \models_{\mathcal{A}_G} \phi_2, \text{and for all} \\
 & \quad \text{positions } (j,t') \text{ of } \omega \text{ such that } (j,t') \prec (i,t), \\
 & \quad \omega(j+t') \models_{\mathcal{A}_G} \phi_1 \vee \phi_2\ .
\end{array}$$

Note that the feasibility condition we impose on adversaries ensures that the set $\{\omega \,|\, \omega \in Path^A_{ful}(\langle s,\nu\rangle)\ \&\ \omega \models_{\mathcal{A}_G} \phi_1\, \mathcal{U}_{\sim k}\, \phi_2\}$ is measurable with respect to the probability space $P_{A,\langle s,\nu\rangle}$ induced by A and $\langle s,\nu\rangle$.

5 The Refined Region Graph

As already observed, the standard region construction applied to a continuous probabilistic timed automaton fails in the case of quantitative probabilistic temporal properties. We propose to quotient over *smaller* intervals of clock values, and to this end subdivide each unit interval into n intervals of the same size for some $n \in \mathbb{N}$. The intuition is that, as we subdivide into smaller regions, we obtain an improvement of the minimum/maximum probability bounds, which in the limit tend to the exact bounds as the number of subdivisions increases.

We deal with the inevitable loss of information caused by the finiteness of the construction by providing a bound on the error.

We first refine the equivalence relation of [4] to intervals of size $\frac{1}{n}$.

Definition 11. *For any $x \in \mathcal{X}$, let k_x be the largest constant x is compared to in any of the invariant and enabling conditions of G. For any $\nu \in \Gamma(G)$ and $x \in \mathcal{X}$, define x to be* relevant *for ν if $\nu(x) \leq k_x$.*

Definition 12 (Clock equivalence). *For clock assignments ν and ν' in $\Gamma(G)$ and $n \in \mathbb{N}$, $\nu \cong_n \nu'$ if and only if the following conditions are satisfied:*

1. *$\forall x \in \mathcal{X}$ either $\lfloor n \cdot \nu(x) \rfloor = \lfloor n \cdot \nu'(x) \rfloor$ or x is not relevant for ν and ν';*
2. *$\forall x, x' \in \mathcal{X}$ relevant for ν:*
   (i) $fract(\nu(x)) < fract(\nu(x'))$ if and only if $fract(\nu'(x)) < fract(\nu'(x'))$.
   (ii) $fract(\nu(x)) > fract(\nu(x'))$ if and only if $fract(\nu'(x)) > fract(\nu'(x'))$.

Let $[\nu]_n$ denote the equivalence class to which ν belongs under $\cong_n$. The following lemma allows us to extend the notion of satisfaction of clock constraints to equivalence classes of clocks.

Lemma 1 ([4]). *Let $\nu, \nu' \in \Gamma(G)$ such that $\nu \cong_n \nu'$. Then, for any clock constraint $\zeta \in \mathcal{C}_{\mathcal{X}}$, ν satisfies ζ if and only if ν' satisfies ζ.*

We now define a probabilistic graph $R_n(G, \phi)$ (where ϕ is a PTCTL formula) whose vertices are pairs consisting of the nodes of G and the equivalence classes with respect to $\cong_n$. As in [3], to improve the complexity of the model checking algorithm, we keep track of the time elapsed when passing through sequences of regions by adding an extra clock $\mathbf{x}$ to $\mathcal{X}$ and setting $k_{\mathbf{x}}$ to be the maximal time-bound appearing in the formula ϕ. We start with some preliminary definitions following the construction in [3,19].

Definition 13. *Let α and β be distinct equivalence classes of $\Gamma(G)$.*

- *The equivalence class β is said to be the* successor *of α, denoted $succ(\alpha)$, if for all $\nu \in \alpha$, there exists $t > 0$ such that $\nu + t \in \beta$ and $\nu + t' \in \alpha \cup \beta$ for all $0 \leq t' < t$.*
- *The class α is said to be an* invariant class of s *if $succ(\alpha)$ violates the invariant condition $inv(s)$.*
- *The class α is an* end class *if, for all $x \in \mathcal{X}$, x is not relevant for α. If α is an end class then, for any $s \in \mathcal{S}$, $\langle s, \alpha \rangle$ is an* end region.

Thus, if we are in an *invariant class* of s then we cannot let time advance sufficiently to move into a new equivalence class without the invariant condition being violated. If we are in an *end class* then we can remain in this region and let time diverge.

The next step is to define the transition relation over regions. As in the standard approach, there are two types of transitions, due to passage of time and change of state respectively, which we consider in turn.

Transitions due to *passage of time* are straightforward using Definition 13: the region that can be reached from $\langle s, \alpha \rangle$ due to passage of time is $\langle s, succ(\alpha) \rangle$. *State*

transitions are more complex to deal with. Suppose that we are in a region $\langle s, \alpha \rangle$ and a state transition occurs. Then, by definition of the model, the following two choices are made in succession:

- a discrete probability distribution $p_s \in prob(s)$, where $p_s \in \mu(\mathcal{S})$, such that the enabling condition $\tau_s(p_s)$ is satisfied by α, is selected non-deterministically;
- then, supposing p_s is chosen, a transition is made according to p_s.

In order to establish which equivalence classes the system moves to, we consider what happens to the values of all the clocks when the transition is made. Consider a transition from a region $\langle s, \alpha \rangle$ to some node s'. To understand the possible equivalence classes of clock assignments associated with s', we consider the equivalence classes separately for the clocks of $O(s')$, denoted α'_O and of $N(s')$, denoted α'_N.

The equivalence class α'_O is the restriction of α to the clocks of $O(s')$, since the clocks of $O(s')$ remain unchanged. The clocks of $N(s')$ are assigned new values at random, and thus the new equivalence class α'_N is determined by a probability distribution that can be computed through simple integrations. We call $P_{N(s')}$ the *joint probability measure* for the clocks of $N(s')$.

The problem is how to combine the equivalence classes of α'_O and α'_N to obtain a unique equivalence class of clock assignments, since we have no way of stating the relative orders between the clocks in $O(s')$ and $N(s')$. Indeed, there are several possible equivalence classes that work correctly. Since we have no way of determining which one is correct, we introduce a *nondeterministic choice* between them all, and consequently *an error* which we analyze in Section 6. The definition below can be used to determine all the possible equivalence classes that are consistent with α'_O and α'_N.

Definition 14. *If α_1 and α_2 are equivalence classes of clock assignments defined on some subset of clocks X_1 and X_2 respectively such that $X_1 \cap X_2 = \emptyset$, then we let $\alpha_1 \cup \alpha_2$ be the set of equivalence classes over $X_1 \cup X_2$ such that $\gamma \in \alpha_1 \cup \alpha_2$ if and only if $\gamma \restriction X_1 = \alpha_1$, $\gamma \restriction X_2 = \alpha_2$, where $\restriction$ denotes restriction.*

Based on the discussion above, we introduce the notion of a *union region*, which is a triple $\langle s, \alpha_O, \alpha_N \rangle$, where α_O is an equivalence class of $O(s)$, and α_N is an equivalence class of $N(s)$.

We are now ready to formulate the region graph for a continuous probabilistic timed automaton G and PTCTL formula ϕ.

Definition 15 (Region Graph). *The region graph $R_n(G, \phi)$ is defined to be the graph $(V, Rstep)$. The vertex set V is the set of regions and union regions (satisfying the corresponding invariant condition). The probabilistic edge function $Rstep : V \to \mathcal{P}_f(\mu(V))$ consists of three types of steps:*

- (**passage of time**) *if $\langle s, \alpha \rangle \in V$ and α is not an invariant class of s, then the point distribution over $\langle s, succ(\alpha) \rangle$ is an element of $Rstep\langle s, \alpha \rangle$.*

- (**transitions of** G) *if* $\langle s, \alpha\rangle \in V$, $p_s \in prob(s)$ *and* $\tau_s(p_s)$ *is satisfied by* α, *then the distribution* p *such that the probability of each union region* $\langle s', \alpha'_1, \alpha'_2\rangle$ *is* $p_s(s')P_{N(s')}(\alpha'_2)$ *when* $\alpha'_1 = \alpha \restriction O(s)$ *and is 0 otherwise is an element of* $Rstep\langle s, \alpha\rangle$.
- (**division**) *if* $\langle s, \alpha_1, \alpha_2\rangle \in V$, *then for each* $\alpha' \in \alpha_1 \cup \alpha_2$, *the point distribution over* $\langle s, \alpha'\rangle$ *is an element of* $Rstep\langle s, \alpha_1, \alpha_2\rangle$.

The definition of a path for a region graph is similar to the definition of a path for a continuous probabilistic timed automaton with the exception that the labels of the arrows do not contain time values.

Definition 16 (Adversary of R_n). *A (randomized) adversary* B *on the region graph is a function* B *mapping every finite path* π *of* $R_n(G, \phi)$ *to a distribution over* $Rstep(last(\pi))$.

The definition of a probability space $\mathcal{P}_{B,v}$ on $Path^B_{ful}(v)$, given a randomized adversary B and a region v, is standard [10,20]. The definition of divergent adversaries can also be adapted easily to region graphs (see [19]).

6 Model Checking Continuous Probabilistic Timed Automata

The aim of this paper is to extend the result of [19], which we now recall. Suppose G is a *discrete* probabilistic timed automaton and the mapping $\phi \mapsto \Phi$ from PTCTL to PBTL [8] is as defined in [19]. Then, for any $\langle s, \nu\rangle \in G$ and $\phi \in$ PTCTL, we have

$$\langle s, \nu\rangle \models_{\mathcal{A}_G} \phi \text{ if and only if } R_1\langle s, \nu\rangle \models_{\mathcal{A}_{R_1}} \Phi$$

where $R_1\langle s, \nu\rangle$ denotes the unique state $\langle s', \alpha'\rangle \in R_1(G, \phi)$ of the region graph such that $s' = s$ and $\alpha = [\nu[\mathbf{x} := 0]]_1$. In particular, the formula $[\phi_1 \,\forall\mathcal{U}_{\sim k}\, \phi_2]_{\sqsupseteq\delta}$ is mapped to $[\Phi_1 \,\forall\mathcal{U}\, (\Phi_2 \wedge a_{\mathbf{x}\sim k})]_{\sqsupseteq\delta}$, where $a_{\mathbf{x}\sim k}$ is the atomic proposition which encodes the time bound subscript $\sim k$, and labels a region $\langle s, \alpha\rangle$ if and only if $\alpha \models \mathbf{x} \sim k$. By abuse of notation, we abbreviate $\Phi_1 \,\mathcal{U}\, (\Phi_2 \wedge a_{\mathbf{x}\sim k})$ to $\Phi_1 \,\mathcal{U}_{\sim k}\, \Phi_2$.

Let G be a continuous probabilistic timed automaton, ϕ be a PTCTL formula, and $R_n(G, \phi)$ be the region graph for G and ϕ with each unit interval refined into n parts. The region graph $R_n(G, \phi)$ does not preserve the validity of ϕ in general since its construction does not preserve the probabilities of events. In particular, it is not the case that a state $\langle s, \nu\rangle \in G$ satisfies ϕ if and only if the corresponding state $R_n\langle s, \nu\rangle$ of $R_n(G, \phi)$ satisfies Φ.

To understand the problem better, let $\phi = [\phi_1 \,\forall\, \mathcal{U}_{\sim k}\, \phi_2]_{\geq\delta}$. Suppose that there is a known upper bound λ on the error that we introduce by evaluating the probability of $\Phi_1 \mathcal{U}_{\sim k} \Phi_2$ on $R_n\langle s, \nu\rangle$ rather than on $\langle s, \nu\rangle$ (see Section 6.1 for the method to compute λ), and that the minimum, over all $B \in \mathcal{A}_{R_n(G,\phi)}$, of the probability of the paths of B starting from $R_n\langle s, \nu\rangle$ and satisfying $\Phi_1 \mathcal{U}_{\sim k} \Phi_2$ is p_1. Then we can deduce that, from $\langle s, \nu\rangle$, there exists an adversary for which the probability of paths from $\langle s, \nu\rangle$ satisfying $\phi_1 \,\mathcal{U}_{\sim k}\, \phi_2$ is in the interval $[p_1, p_1 + \lambda]$.

If $\delta \leq p_1$, then we can conclude that ϕ is valid; if $\delta > p_1 + \lambda$, then we can conclude that ϕ is not valid. If δ is in the interval $(p_1, p_1 + \lambda]$, then Φ is valid in $R_n\langle s, \nu\rangle$; however, ϕ may or may not be valid in $\langle s, \nu\rangle$. In this case we have three possible choices for how to proceed:

1. consider a more refined graph in the hope of solving the uncertainty;
2. say that we do not know the correct answer ("don't know");
3. say that the formula is valid and warn the user that there may be an error of λ in the determination of the probability bound.

The first case has obvious complexity implications. In the second case we need to deal with a three-valued logic, which would involve propagating the "don't know" values to the higher levels of the parse tree of the PTCTL formula in question. In the third case the difficulty is that we cannot quantify the propagation of the error to super-formulae of ϕ. This is because in the worst case we may estimate wrongly the validity of ϕ on most of the states of G. Thus, the only thing that we can say safely in this case is that at each level we may be wrong by some value λ in the estimation of probabilities.

The results we obtain allow us to adopt the third solution, namely, to calculate an interval of probabilities to which the actual probability bound belongs, together with an estimate of error, for a given number of subdivisions n, and refine the region graph further in case "don't know" outcomes have resulted.

6.1 Main Results

Before we can state our results we need some auxiliary definitions. For the rest of the discussion we fix a continuous probabilistic timed automaton G and a formula ϕ. Let s, s' be nodes of G and let α, α' be sets of clock assignments for the clocks of G. We say that $\langle s, \alpha\rangle$ is *contained* in $\langle s', \alpha'\rangle$, denoted $\langle s, \alpha\rangle \leq \langle s', \alpha'\rangle$, if $s = s'$ and $\alpha \subseteq \alpha'$. Given two region graphs R_m, R_n, we say that R_m *refines* R_n, denoted by $R_m \leq R_n$, if each region of R_n is contained in a region of R_m. The next lemma implies that the probability bounds do not increase with further subdivisions of the region graph.

Lemma 2. *$R_m(G, \phi) \leq R_n(G, \phi)$ if n divides m.*

The next notion plays an important part in estimating the error. Let R_m, R_n be two region graphs such that $R_m \leq R_n$, and $v = \langle s, \alpha_1, \alpha_2\rangle$ a union region of R_m. We say that v is *homogeneous* with respect to R_n if there exists a unique region v' of R_n that contains each region of $\{\langle s, \gamma\rangle \mid \gamma \in \alpha_1 \cup \alpha_2\}$.

Fix a refined region graph R_n for G, ϕ and some $n \in \mathbb{N}$. For any $A \in \mathcal{A}_G$, $\langle s, \nu\rangle \in G$, $B \in \mathcal{A}_{R_n}$, $\langle s, \alpha\rangle \in R_n$ and $\phi_1, \phi_2 \in$ PTCTL we let:

$$P^A_{\phi_1 \mathcal{U}_{\sim k} \phi_2}\langle s, \nu\rangle \stackrel{\text{def}}{=} P_{A,\langle s,\nu\rangle}\{\omega \mid \omega \in Path^A_{ful}\langle s, \nu\rangle \text{ and } \omega \models_{\mathcal{A}_G} \phi_1 \,\mathcal{U}_{\sim k}\, \phi_2\}$$
$$P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle \stackrel{\text{def}}{=} P_{B,\langle s,\alpha\rangle}\{\pi \mid \pi \in Path^B_{ful}\langle s, \alpha\rangle \text{ and } \pi \models_{\mathcal{A}_{R_n}} \Phi_1 \,\mathcal{U}_{\sim k}\, \Phi_2\}.$$

Suppose that $\phi_1, \phi_2 \in$ PTCTL are such that for any $\langle s, \nu\rangle \in G$:

$$\langle s, \nu\rangle \models_{\mathcal{A}_G} \phi_1 \Leftrightarrow R_n\langle s, \nu\rangle \models_{\mathcal{A}_{R_n}} \Phi_1 \text{ and } \langle s, \nu\rangle \models_{\mathcal{A}_G} \phi_2 \Leftrightarrow R_n\langle s, \nu\rangle \models_{\mathcal{A}_{R_n}} \Phi_2.$$

Then we can show the following correspondence holds between adversaries of the automaton G and its region graph R_n (see [18]).

Proposition 1. *For any $A \in \mathcal{A}_G$, $\langle s, \nu\rangle \in G$ and $n \in \mathbb{N}$, there exists $B \in \mathcal{A}_{R_n}$ such that $P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle = P^A_{\phi_1 \mathcal{U}_{\sim k} \phi_2}\langle s, \nu\rangle$ where $R_n\langle s, \nu\rangle = \langle s, \alpha\rangle$.*

Proposition 2. *For any $n \in \mathbb{N}$, $B \in \mathcal{A}_{R_n}$, $\langle s, \alpha\rangle \in R_n$ and $\langle s, \nu\rangle \in G$ with $R_n\langle s, \nu\rangle = \langle s, \alpha\rangle$, there exists $A \in \mathcal{A}_G$ such that $P^A_{\phi_1 \mathcal{U}_{\sim k} \phi_2}\langle s, \nu\rangle$ and $P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle$ differ by at most the probability of reaching a non-homogeneous region before satisfying or violating $\Phi_1 \mathcal{U}_{\sim k} \Phi_2$.*

As a corollary of Proposition 1 and Proposition 2, we obtain the following crucial correspondence between the probability bounds calculated on R_n and those on G. Its importance is in stating that the probabilities of a PTCTL until formula over the divergent adversaries of G are bounded by the probabilities for the corresponding PBTL formula over the divergent adversaries of the region graph. The latter probability calculation is standard and proceeds via reduction to a linear programming problem [10,8]. Moreover, since the difference in these values can be no more than the probability of reaching a non-homogeneous region before satisfying or violating $\Phi_1 \mathcal{U}_{\sim k} \Phi_2$, this yields the estimate of error. This error can also be calculated by standard methods [10,8].

Corollary 1. *For any $\langle s, \nu\rangle \in G$ and $n \in \mathbb{N}$, if $R_n\langle s, \nu\rangle = \langle s, \alpha\rangle$ and the maximum probability of reaching a non-homogeneous region before satisfying or violating $\Phi_1 \mathcal{U}_{\sim k} \Phi_2$ from $R_n\langle s, \nu\rangle$ is λ, then*

$$\inf_{A \in \mathcal{A}_G} P^A_{\phi_1 \mathcal{U}_{\sim k} \phi_2}\langle s, \nu\rangle \in \left[\min_{B \in \mathcal{A}_{R_n}} P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle, \min_{B \in \mathcal{A}_{R_n}} P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle + \lambda\right]$$

$$\sup_{A \in \mathcal{A}_G} P^A_{\phi_1 \mathcal{U}_{\sim k} \phi_2}\langle s, \nu\rangle \in \left[\max_{B \in \mathcal{A}_{R_n}} P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle - \lambda, \max_{B \in \mathcal{A}_{R_n}} P^B_{\Phi_1 \mathcal{U}_{\sim k} \Phi_2}\langle s, \alpha\rangle\right] .$$

6.2 Example

We illustrate the working of our method with the help of an example. Consider the automaton H in Figure 1. From s_0 we enable a transition that moves to s_1 and sets x uniformly in the interval $[0, 1]$. From s_1 we enable two transitions: one transition, T_1, moves to node s_2 and sets y uniformly in the interval $[0, 1]$, while the other transition, T_2, moves to s_3 with probability $\frac{2}{3}$ and to s_4 with probability $\frac{1}{3}$. From s_2 we enable a transition to s_4 if $y > x$. We consider the upper bound on the probability of reaching of s_4, i.e. the formula $[\mathtt{true}\, \forall\, \mathcal{U}_{\geq 0}\ a_{s_4}]_{\sqsupseteq \delta}$. The adversary that gives the highest probability ($\frac{5}{9}$) is obtained by scheduling T_1 immediately in s_1 if $x < \frac{2}{3}$ and T_2 otherwise.

From s_0 to s_1 the possible regions that can be reached are $\frac{k}{n} < x < \frac{k+1}{n}$ for $k = 0, \ldots, n-1$, each with probability $\frac{1}{n}$. In the region $\langle s_1, \frac{k}{n} < x < \frac{k+1}{n}\rangle$ there is a choice between letting time advance, taking the transition T_1 or the transition T_2. It follows that the maximum probability of reaching s_4 equals: $\sum_{k=0}^{n-1} \frac{1}{n} \cdot \max\left(\frac{n-k}{n}, \frac{1}{3}\right)$.

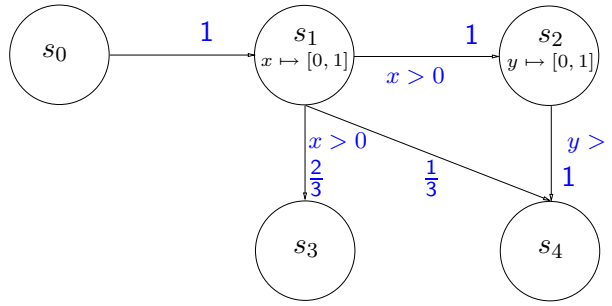


Fig. 1. The continuous probabilistic timed automaton H

To reach a non-homogeneous region, y must be set, then supposing x has been set already, this has probability $\frac{1}{n}$. Therefore, the maximum probability of reaching a non-homogeneous region is $\frac{1}{n}$, which yields the following:

- R_1: upper bound is 1, error $\frac{4}{9}$ and estimate of error 1;
- R_2: upper bound is $\frac{3}{4}$, error $\frac{7}{36}$ and estimate of error $\frac{1}{2}$;
- R_4: upper bound is $\frac{31}{48}$, error $\frac{13}{144}$ and estimate of error $\frac{1}{4}$;
- R_{100}: upper bound is $\frac{5589}{10000}$, error $\frac{301}{90000}$ and estimate of error $\frac{1}{100}$.

7 Conclusions

We have proposed a model checking method for continuous probabilistic timed automata against PTCTL specifications. In the formalism we propose, we can specify timing properties such as "at least 80% of packets will be delivered within k units of time assuming the packets arrive according to f" where f is a continuous-time probability distribution (for example, uniform or normal) with support within a closed interval of $\mathbb{R}^{\geq 0}$. The model checking algorithm runs on a finite region-like graph, obtained through subdividing the unit intervals. We show how to approximate the probability to within an interval, where approximations improve with further subdivisions, and estimate the error of the approximation.

It is known that the complexity of the verification of real-time systems is expensive, and the method proposed here is no exception. Research into improving the complexity of our procedure, for example using symbolic methods, would be necessary before it can be applied to real-world problems.

Acknowledgements

We thank Pedro D'Argenio for pointing out a flaw in our previous attempt to solve this problem. We also thank the anonymous referees for their helpful comments.

References

1. R. Alur. Private communication. 1998. 124
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. In *Proc. ICALP'91*, volume 510 of *LNCS*. Springer, 1991. 125, 127
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1), 1993. 124, 125, 131
4. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126, 1994. 123, 124, 127, 131
5. R. B. Ash. *Real Analysis and Probability*. Academic Press, 1972. 125, 126, 129
6. C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *Proc. ICALP'97*, volume 1256 of *LNCS*. Springer, 1997. 125
7. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *CONCUR'99*, volume 1664 of *LNCS*. Springer, 1999. 125
8. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11, 1998. 128, 133, 135
9. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of UPPAAL. In *Proc. International Workshop on Software Tools for Technology Transfer*, 1998. 123
10. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *FST and TCS*, volume 1026 of *LNCS*. Springer, 1995. 133, 135
11. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In *Proc. CAV'98*, volume 1427 of *LNCS*. Springer, 1998. 123
12. P. D'Argenio, J.-P. Katoen, and E. Brinksma. Specification and analysis of soft real-time systems: Quantity and quality. In *Proc. IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1999. 125
13. L. de Alfaro. How to specify and verify the long-run average behaviour of probabilistic systems. In *Proc. LICS'98*. IEEE Computer Society Press, 1998. 125
14. L. de Alfaro. Stochastic transition systems. In *Proc. CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998. 125
15. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Approximating labelled Markov processes. To appear in LICS'2000. 125
16. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(4), 1994. 125
17. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 1994. 127
18. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of quantitative properties of continuous probabilistic real-time automata. Technical Report CSR-00-06, University of Birmingham, 2000. 135
19. M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. Technical Report CSR-00-02, University of Birmingham, 2000. Accepted for a Special Issue of Theoretical Computer Science. Preliminary version of this paper appeared in Proc. ARTS'99, LNCS vol 1601, 1999. 123, 124, 127, 128, 130, 131, 133
20. R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995. 126, 133

The Impressive Power of Stopwatches*

Franck Cassez[1] and Kim Larsen[2]

[1] IRCCyN/CNRS UMR 6597, France
Franck.Cassez@irccyn.ec-nantes.fr
[2] Dep. of Computer Science, Aalborg University, Denmark
kgl@cs.auc.dk

Abstract. In this paper we define and study the class of *stopwatch automata* which are timed automata augmented with *stopwatches* and *unobservable behaviour*. In particular, we investigate the expressive power of this class of automata, and show as a main result that any finite or infinite *timed language* accepted by a *linear hybrid automaton* is also acceptable by a stopwatch automaton. The consequences of this result are two-fold: firstly, it shows that the seemingly minor upgrade from timed automata to stopwatch automata immediately yields the full expressive power of linear hybrid automata. Secondly, reachability analysis of linear hybrid automata may effectively be reduced to reachability analysis of stopwatch automata. This, in turn, may be carried out using an easy (over-approximating) extension of the efficient reachability analysis for timed automata to stopwatch automata. We report on preliminary experiments on analyzing translations of linear hybrid automata using a stopwatch-extension of the real-time verification tool UPPAAL.

1 Introduction

Hybrid systems. Hybrid systems [ACH+95,AHH96,Hen96] are a strong extension of timed automata [AD94] used to model systems which combine *discrete* and *continuous* evolutions. The *reachability* and *ω-language emptiness* problems (RP and LEP) are key problems for the verification of hybrid automata: these problems are decidable for *timed automata* (TA) [AD94] (and PSPACE-complete) but not for *linear hybrid automata* (LHA) [ACH+95] for which the reachability problem is only semi-decidable. Decidability of RP and LEP have been extensively studied for subclasses of hybrid automata [KPSY93,AR95,Cer92,JP94,AP94] [HKPV98]. We investigate here a related issue which is the characterization of the expressive power of various subclasses of hybrid automata.

Related work. Concerning expressiveness of timed automata as timed language acceptors, it was proven in [AD94] that Timed Muller Automata (TMA) are as

* This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid systems) and by the FIREworks Esprit WG 23531; it was carried out while the first author was visiting BRICS@Aalborg during the fall 1999.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 138–152, 2000.

expressive as Timed Büchi Automata (TBA), which in turn are more expressive than Deterministic TMA (DTMA), which are themselves more expressive than Deterministic TBA (DTBA).

More recent results concern the expressiveness of clocks and the power of (silent) τ-transitions:

- In [HKWT95], Henzinger and al. investigated the power of timing restrictions on finite automata and showed that clock constraints together with time divergence enables one to express Büchi, Muller, Streett, Rabin acceptance and fairness conditions for finite automata; In [ACH94] Alur and al. studied a variety of (un)timed equivalences for timed automata and the distinguishing power of clocks as observers.
- there are also papers devoted to the expressive power of τ-transitions for timed automata; in [BGP96] it is proven that τ-transitions strictly increase the power of timed automata (as timed language acceptors) if they reset clocks; moreover timed automata with τ-transitions are more robust than timed automata without in the sense that any language recognized when the time domain in $\mathbb{N}$ remains recognizable when the domain is $\mathbb{R}$. The expressive power of τ-transitions which reset clocks is settled in [DGP97]; [BDGP98] is a synthesis of the two above mentioned papers.
- other relevant results [HK97,HK96] concern a subclass of hybrid automata: *rectangular hybrid automata* (RHA); in [HKPV98] it is shown that initialized RHA (IRHA) are equivalent to timed automata and thus RP and LEP are decidable for this subclass. The initialization property states that whenever a slope of a variable changes it must be reset. The authors showed that relaxing either the rectangular or the initialization assumptions leads to undecidability of RP and LEP, thus proving that IRHA are at the border or decidability and undecidability.

Our contribution. We compare the expressive power of linear hybrid automata (LHA) and certain of its subclasses. More precisely, we show that, in terms of expressiveness, one important class is the one obtained by a simple addition of stopwatches to the class of timed automata (TA)[1]: we refer to the resulting class as the class of *stopwatch automata* (SWA).

Extending hybrid automata with *unobservable* timed transitions[2], we prove that SWA with unobservable delays are as expressive as LHA. That is, every (ω-) language accepted by a LHA is also accepted by some SWA with unobservable delays. We consider this result interesting for two reasons:

1. it indicates that undecidability of RP and LEP originates from the ability to stop time, and
2. it has a practical application to verification of linear hybrid systems.

The application to verification is based on an easy extension of algorithms for model-checking (safety properties of) TA to (over-approximating) algorithms

[1] here TA refers to timed automata with simple constraints as defined in [AD94].
[2] i.e. some durations are unobservable for this class of hybrid automata.

for model-checking SWA. In particular, this extension may apply the full range of efficient data-structures [LLPY97,ABK+97,BLP+99] currently applied in verification tools for timed automata [Yov97,LPY97]. Analysis of a LHA may now be reduced to a similar analysis on the equivalent SWA, avoiding the need for representing and manipulating general polyhedra.

Outline of the paper. In the next section 2 we give the definitions of LHA and of the subclasses of hybrid automata we will be interested in and we define *unobservable delays*. Section 3 states the main result: the fact that SWA have the same expressive power as LHA. In Section 4 we focus on the practical interest of this equivalence and give examples of LHA we are able to verify using our translation and the SWA-extension of UPPAAL. Finally we conclude in section 5.

2 Linear Hybrid Automata

2.1 Preliminaries

For a given alphabet Σ, Σ^* denotes the set of finite words and Σ^ω the set of infinite words over Σ. Also $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. We also use the set of booleans $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$.

A *valuation* is an element of $\mathbb{R}^V$, where V is a finite set of variables. If $|V| = n$, a valuation v can be interpreted as a vector $\overline{v}$ of $\mathbb{R}^n$. If $V' \subseteq V$ and $\nu \in \mathbb{R}^V$, we denote by $proj_{V'}(\nu)$ the valuation $\nu' \in \mathbb{R}^{V'}$ defined by $\nu'(x) = \nu(x), \forall x \in V$.

A *linear expression* $\phi(\overline{v})$ over V is of the form $\sum a_i v_i$ with $a_i \in \mathbb{Z}, v_i \in V$. A *linear constraint* is a propositional formula using the connectives $\vee, \wedge, \neg$ over atomic formulæ of the form $\phi(\overline{x}) \bowtie c$, where $\bowtie \in \{<, =, >\}$, $\phi(\overline{x})$ is a linear expression and $c \in \mathbb{N}$. $\mathcal{LC}(V)$ is the set of linear constraints. If we restrict the linear expressions to one of the simple forms $v - v' \bowtie c$ or $v \bowtie c$, $v, v' \in V$ we obtain the set $\mathcal{SC}(V)$ of *simple constraints* over V. A *linear assignment* over V is of the form $\overline{v} := A.\overline{v} + \overline{b}$ where A is a $n \times n$ matrix with coefficients in $\mathbb{Z}$ and $\overline{b}$ is a vector of $\mathbb{Z}^n$. $\mathcal{LA}(V)$ is the set of linear assignments over V. A *simple assignment* is such that all entries of A are either 0 or 1, and every row of A has at most one non-null coefficient.

Given a valuation v and a constraint γ, the boolean value $\gamma(v)$ describes whether γ is satisfied by v or not.

A *continuous change* of the variables is defined w.r.t. an element d of $\mathbb{Z}^V$ corresponding to the first derivative of each variable: given $t \in \mathbb{R}_{\geq 0}$, the valuation $v + d.t$ is defined by $(v + d.t)(x) = v(x) + d(x).t$.

2.2 Hybrid Automata

Hybrid automata [ACH+95,AHH96,Hen96] are used to model systems which combine *discrete* and *continuous* evolutions. For general hybrid systems the activities can be any continuous functions. However, we restrict our attention to

the subclass of *linear* hybrid systems[3]. Moreover we assume that the initial value of each variable is 0 (the corresponding initial valuation is denoted v_0).

Definition 1 (Linear hybrid automaton). *A* linear hybrid automaton *(in the sequel LHA)* $\mathcal{H}$ *is a 7-tuple* $(N, l_0, V, A, E, Act, Inv)$ *where:*

- N *is a finite set of* locations,
- $l_0 \in N$ *is the* initial location, v_0 *is the* initial valuation,
- V *is a finite set of real-valued* variables,
- A *is a finite set of* actions,
- $E \subseteq N \times \mathcal{LC}(V) \times A \times \mathcal{LA}(V) \times N$ *is a finite set of* edges*;* $e = \langle l, \gamma, a, \alpha, l' \rangle \in E$ *represents an edge from the location* l *to the location* l' *with the linear guard* γ*, the label* a *and the linear assignment* α*.*
- $Act \in \left((\mathbb{Z} \times \mathbb{Z})^V\right)^N$ *where* $Act(l)(x) = [u_1, u_2]$ *means that the first derivative of* x *in location* l *lies in the compact bounded interval* $[u_1, u_2]$ *of* $\mathbb{Z}$*.*
- $Inv \in \mathcal{LC}(V)^N$ *assigns a linear* invariant *to any location.*

To this standard definition we add the following features: the set of locations N *is partitioned into two subsets* $N_o \cup N_u$*;* N_o *(resp.* N_u*) is the set of locations where time-elapsing is* observable *(resp.* unobservable*). We also denote the unobservable action* τ *and consider hybrid automata with* τ *moves. We denote* A^τ *the set* $A \cup \{\tau\}$ *and* Δ^τ *the set* $\mathbb{R}_{\geq 0} \cup \{\tau(t), t \in \mathbb{R}_{\geq 0}\}$ □

Example 1 (Water-level monitor [ACH+95]). As a running example, we consider the water-level monitor described in [ACH+95] and illustrated [4] in Figure 1. The aim is to control the water level in a tank with a monitor. A pump can be turned on and off to control the level. When the pump is off (locations ℓ_2 and ℓ_3) the water level falls by two cms per second; when the pump is on (locations ℓ_0 and ℓ_1), the level rises by one cm per second. The delay to turn the pump on and off is 2 time units (measured by the variable x). Time elapsing is observable in each location of the water level monitor. □

2.3 Semantics

Definition 2. *The semantics of a hybrid automaton* $\mathcal{H} = (N, l_0, V, \Sigma, E, Act, Inv)$ *is a timed transition system* $S_{\mathcal{H}} = (Q, q_0, \Sigma, \rightarrow)$ *where* $Q = N \times \mathbb{R}^V$*,* $q_0 = (l_0, v_0)$ *is the initial state* ($v_0(x) = 0$, $\forall x \in V$) *and* $\rightarrow$ *is defined by:*

$$\langle l, v \rangle \xrightarrow{a} \langle l', v' \rangle \text{ iff } \exists\, (l, \gamma, a, \alpha, l') \in E \text{ s.t. } \begin{cases} \gamma(v) = \mathsf{tt},\ v' = \alpha(v) \text{ and} \\ Inv(l')(v') = \mathsf{tt} \end{cases}$$

$$\langle l, v \rangle \xrightarrow{e} \langle l', v' \rangle \text{ iff } \begin{cases} e = d \text{ if } l \in N_o, \quad e = \tau(d) \text{ if } l \in N_u \\ l = l' \text{ and } \exists d \in Act(l) \text{ s.t. } v' = v + d.t \text{ and} \\ \forall\, 0 \leq d' \leq d, v + d'.t \in Inv(l) \end{cases}$$

$\tau(d)$ stands for an unobservable delay of duration d. □

[3] The example of the thermostat in [ACH+95] is a hybrid automata which is *not* linear.

[4] automata designed with GasTeX (http://www.liafa.jussieu.fr/~gastin/gastex).

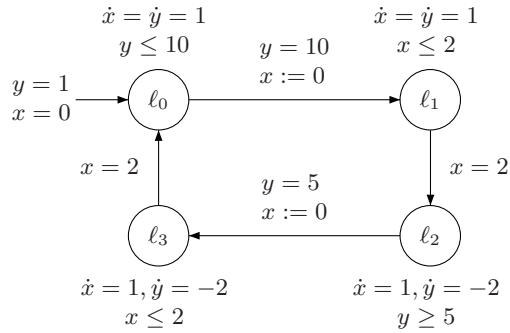


Fig. 1. The water-level monitor

A *run* from a state s_0 of a linear hybrid automaton $\mathcal{H}$ is a sequence of alternating discrete and continuous transitions of $S_{\mathcal{H}}$:

$$\rho = s_0 \xrightarrow{\delta_0} s'_0 \xrightarrow{a_0} \dots s_i \xrightarrow{\delta_i} s'_i \xrightarrow{a_i} \dots$$

where $a_i \in A^\tau$ and $\delta_i \in \Delta^\tau$. Intuitively time δ_{i+1} elapses between the actions a_i and a_{i+1}. We also introduce the following derived notations:

$$\begin{array}{l} s \stackrel{0}{\Longrightarrow} s' \text{ iff } \exists d \in \mathbb{R}_{\geq 0}\, s.t.\, s \xrightarrow{\tau(d)} s' \\ s \stackrel{\varepsilon}{\Longrightarrow} s' \text{ iff } s(\xrightarrow{\tau} \cup \stackrel{0}{\Longrightarrow})^* s' \\ s \stackrel{d}{\Longrightarrow} s' \text{ iff } s \stackrel{\varepsilon}{\Longrightarrow} \xrightarrow{d_1} \stackrel{\varepsilon}{\Longrightarrow} \xrightarrow{d_2} \cdots \stackrel{\varepsilon}{\Longrightarrow} \xrightarrow{d_n} s' \text{ with } d = \sum_i d_i \\ s \stackrel{a}{\Longrightarrow} s' \text{ iff } s \stackrel{\varepsilon}{\Longrightarrow} \xrightarrow{a} s' \end{array}$$

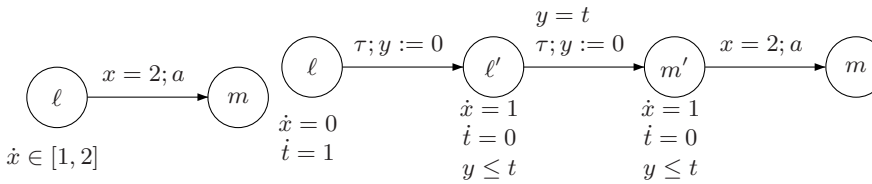
2.4 Timed Languages and Bisimulation for Hybrid Automata

The definition of hybrid automata may be extended with two sets of states: $F \subseteq N$ being a set of *final* states and $R \subseteq N$ being the set of *repeated* states.

The finite (resp. infinite) run ρ is *accepting* if ρ ends in a final state (resp. if there are infinitely many states from R in ρ). With every finite (resp. infinite) run we associate a finite (resp. infinite) τ*-timed word*: with every action a_i we associate a *timestamp* t_i, which is the accumulated observable time since the initial instant. Formally $t_i = \sum_{k=0}^{i} \{\delta_k \in \mathbb{R}_{\geq 0}\}$. Thus the run ρ accepts the τ-timed word $(a_0, t_0) \dots (a_n, t_n) \dots \in (A^\tau \times \mathbb{R}_{\geq 0})^\infty$ and since τ-transitions are unobservable we remove all pairs (τ, t) to obtain a *timed word* $(a_{i_0}, t_{i_0}) \dots (a_{i_n}, t_{i_n}) \dots \in (A \times \mathbb{R}_{\geq 0})^\infty$

The *timed language* $\mathcal{L}(\mathcal{H})$ accepted by the hybrid automaton $\mathcal{H}$ is the set of timed words which have accepting runs from the initial state of $S_{\mathcal{H}}$.

A symmetric relation $B \subseteq Q \times Q$ is a *weak timed bisimulation* if whenever $(s, t) \in B$, the following holds: (1) $s \in F \Leftrightarrow t \in F$ (resp. $s \in R \Leftrightarrow t \in R$), and (2) whenever $s \stackrel{d}{\Longrightarrow} \xrightarrow{a} s'$ then $t \stackrel{d}{\Longrightarrow} \xrightarrow{a} t'$ for some t' with $(s', t') \in B$. We say



(a) Initial LHA: H (b) Corresponding SWA: H'

Fig. 2. Two equivalent linear hybrid automata, H and H'.

that s and t are *weakly timed bisimilar* ($s \approx t$) provided (s, t) is contained in some weak timed bisimulation B. It follows easily that our chosen definition of weak timed bisimilarity between states s and t implies equality in terms of timed words accepted.

Example 2. Consider the linear hybrid automaton H in Figure 2. Assuming that m is final and both ℓ and m are observable, the timed language accepted by H is the set $\{(a, d) \mid 1 \leq d \leq 2\}$. For the linear hybrid automaton H' a typical run is of the form[5]:

$$\begin{aligned}\langle \ell, t = 0, x = 0, y = 0\rangle &\xrightarrow{d}\xrightarrow{\tau} \langle \ell', t = d, x = 0, y = 0\rangle \\ &\xrightarrow{\tau(d)}\xrightarrow{\tau} \langle m', t = d, x = d, y = 0\rangle \\ &\xrightarrow{\tau(d')} \langle m', t = d, x = 2, y = d'\rangle \\ &\xrightarrow{a} \langle m, t = d, x = 2, y = d'\rangle\end{aligned}$$

for $d' \leq d$ and $d' + d = 2$. Assuming again that m is accepting, and removing the unobservable part of the above run, yields (a, d) as an accepted timed word. In fact it is not hard to see, that the timed words of the form (a, d), where $1 \leq d \leq 2$, are precisely the words accepted by H'; hence that $\mathcal{L}(H) = \mathcal{L}(H')$. □

Remark 1. Invariants are not useful when considering timed language acceptance for hybrid automata: we may remove every invariant[6], by translating every transition $\langle l, \gamma, a, \alpha, l'\rangle$ into two consecutive transitions: $\langle l, \gamma \wedge Inv(l), \tau, \alpha.[t := 0], m\rangle$ $\langle m, t = 0 \wedge Inv(l'), a, Id, l'\rangle$, where t is a fresh clock with $\dot{t} = 1$ in m. Leaving the set of final (resp. repeated) states unchanged we obtain the same accepted language. In the sequel we will not consider invariants anymore but assume that this simple translation has already been done for any LHA. □

[5] assuming that ℓ, m are observable and ℓ', m' are unobservable.

[6] this works only for convex invariants as pointed out by an anonymous referee; for a union of convex invariants in ℓ we may add copies of ℓ with convex invariants and apply our translation schema.

2.5 Subclasses of LHA

In the sequel we will consider the following subclasses of LHA (or extended classes of TA)[7]:

TA: the class of timed automata is the one defined in [AD94]; that is, only simple constraints are allowed, the derivatives of the variables (clocks) are all equal to 1 and only linear assignments $(A, \overline{b})$ with $A = 0$ and $\overline{b} \geq 0$ are allowed.

SWA: the class of TA extended with stopwatches; that is the derivative of a variable in a location can be either 0 or 1.

LSWA: the class of LHA, where the derivative of a variable in a location can be either 0 or 1. Alternatively, this class is the extension of SWA which allows linear constraints and assignments.

The set of timed languages accepted by a class C of hybrid automata is denoted $\mathbf{TL}_C$.

Remark 2. For SWA we can allow assignments of the form $x := x' + k$ as they can be written as a shorthand for (1) reset x; (2) let an unobservable delay of $x' + k$ elapse while stopping all the other variables. We will use this shorthand in section 3.1. Notice that allowing this type of assignments in TA make RP and LEP undecidable [BDFP00].

We also point out that stopwatches allow us to use unbounded integers: it suffices to use a variable which is stopped in every location. □

3 From LHA to SWA

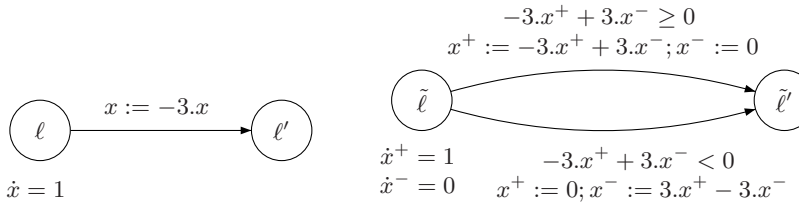
In this section we will show that every language accepted by a LHA is also accepted by a SWA with unobservable delays.

Theorem 1 (Expressiveness of SWA). *The classes LHA and SWA are equally expressive in the sense that* $\mathbf{TL}_{SWA} = \mathbf{TL}_{LHA}$. □

The proof of theorem 1 is implied by the following two proof steps: (1) $\mathbf{TL}_{SWA} = \mathbf{TL}_{LSWA}$, and (2) $\mathbf{TL}_{LSWA} = \mathbf{TL}_{LHA}$. According to remark 1 we will only consider LHA with $\mathtt{tt}$ invariant in every location.

3.1 From LSWA to SWA

Let $L = (N, l_0, V, \Sigma, E, Act, Inv)$ be a LSWA (with $\mathtt{tt}$ invariants). We show how to translate the linear guards and linear assignments of L into simple guards and simple assignments thus obtaining a SWA. However, to prepare these translations, we first transform L in order to ensure that no (stopwatch) variable will ever obtain a negative value.



(a) Initial LHA: L (b) Corresponding SWA: $\tilde{L}$

Fig. 3. Translating assignments to avoid negative values

Avoiding negative values. As L is a LSWA, the stopwatch variables are guaranteed never to *decrease* in any location. However, variables may obviously be assigned negative values when transitions are taken (a simple example is given in Figure 3). First, any transition $\langle \ell, a, g, x := \phi(\overline{x}), \ell' \rangle$[8] is split into two transitions $\langle \tilde{\ell}, a, g \wedge \phi(\overline{x}) \geq 0, x := \phi(\overline{x}), \tilde{\ell'} \rangle$ and $\langle \tilde{\ell}, a, g \wedge \phi(\overline{x}) < 0, x := \phi(\overline{x}), \tilde{\ell'} \rangle$. Obviously, this translation does not alter the behavior of L but provides the knowledge of the sign of the value assigned to x. Now for each variable x we introduce two new (stopwatch) variables x^+ and x^- intending that $x = x^+ - x^-$ and $x^+ \geq 0$ and $x^- \geq 0$ will hold invariantly. To ensure this, we do the following:

1. in each location l of L where $\dot{x} = 1$ we set $(\dot{x}^+, \dot{x}^-) = (1, 0)$; if $\dot{x} = 0$ we set $(\dot{x}^+, \dot{x}^-) = (0, 0)$;
2. we add the (obvious) assignments
   - $(x^+, x^-) := (\phi(\overline{x}), 0)$ for transitions where $\phi(\overline{x}) \geq 0$ is in the guard, and
   - $(x^+, x^-) := (0, -\phi(\overline{x}))$ for transitions with guard $\phi(\overline{x}) < 0$.

Finally, we may completely remove the old variable x by replacing it with $x^+ - x^-$ in all terms (in guards or assignments). Figure 3 shows an example of the transformation.

Translating linear guards. Now we focus on transforming linear guards in transitions into simple ones. Thus consider a general transition $\langle l, a, \phi(\overline{x}) \bowtie c, \alpha, l' \rangle$: a is the label, and α the linear assignment; the guard $\phi(\overline{x}) \bowtie c$ is a linear constraint over V with $\bowtie \in \{<, =, >\}$. Notice that it is possible to rewrite this guard in the form $\sum_{i=1}^{n} a_i x_i - \sum_{i=n+1}^{2n} a_i x_i \bowtie c'$ with $c' \in \mathbb{N}$ and $\mathbb{Z} \ni a_i \geq 0$. We demonstrate how to compute these two sums with two new fresh variables u and v. After this it only remains to replace the guard by the simple $u - v \bowtie c'$.

The translation[9] works as follows (see Figure 4[10]):

[7] the notion of unobservable delay is included in all of those classes.
[8] For simplicity we only consider transitions with a single assignment.
[9] $\dot{V} = 0$ means $\forall v \in V, \dot{v} = 0$.
[10] when no label is written on a transition this is a silent τ-transition.

1. we introduce the new stopwatch variables u and v to sum up the terms $a_i x_i, i \in [1 \dots n]$ and $a_i x_i, i \in [n+1 \dots 2n]$ respectively,
2. between the locations $\tilde{l}$ and $\tilde{l}'$ we introduce new locations $\tilde{l}_i$ where time elapsing is unobservable to perform the computations of u and v.

Note that this translation only works because we (by over previous preparation step) may assume that x_i will not have non-negative values: in particular in location $\tilde{l}_i$ we let time x_i elapse which is possible only if $x_i \geq 0$.

Let $\tilde{V} = V \cup \{u, v, t\}$ and $\nu, \nu' \in \mathbb{R}^V$, $\tilde{\nu}, \tilde{\nu}' \in \mathbb{R}^{\tilde{V}}$ s.t. $proj_V(\tilde{\nu}) = \nu$; we then have the following: $\langle \tilde{l}, \tilde{\nu} \rangle \stackrel{d}{\Longrightarrow} \stackrel{a}{\longrightarrow} \langle \tilde{l}', \tilde{\nu}' \rangle$ iff $\langle l, \nu \rangle \stackrel{d}{\Longrightarrow} \stackrel{a}{\longrightarrow} \langle l', \nu' \rangle$ whenever $proj_V(\tilde{\nu}') = \nu'$; thus $\langle \tilde{l}, \tilde{\nu} \rangle$ and $\langle l, \nu \rangle$ are weakly timed bisimilar and consequently accept the same timed words.

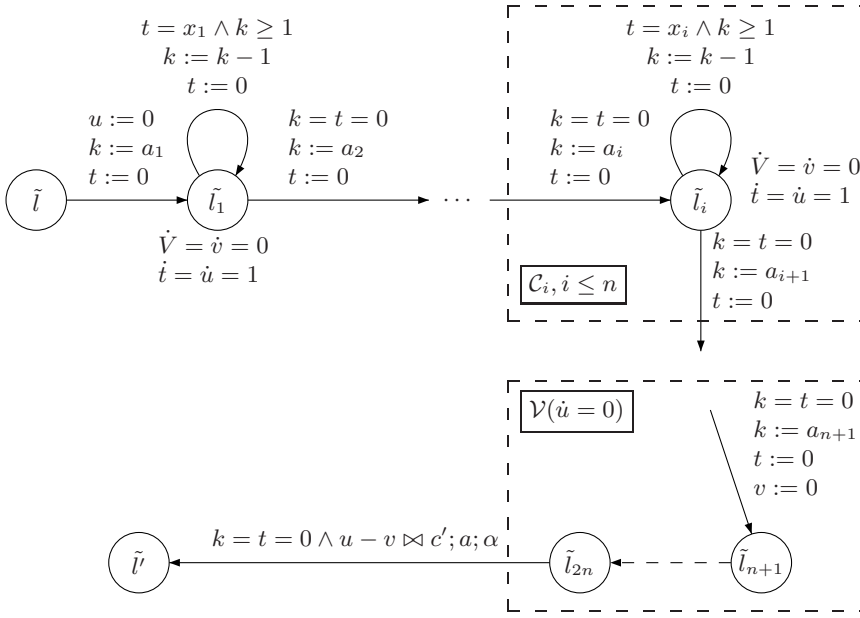


Fig. 4. Computation of u and v before evaluating the guard $u - v \bowtie c'$

Linear assignments. For translating linear assignments we apply the exact same technique as above when dealing with linear guards. Let $A = (a_{ij})$ and $\overline{b} = (b_i)$ with $i, j \in [1..n]$. We introduce n new fresh variables u_i to compute $\sum_{j \leq n} a_{ij} x_j$. At the end we perform the assignments $x_i := u_i + b_i$ (this is possible according to remark 2). Notice we need n new fresh variables as for two simultaneous assignments $x := y$ and $z := x$ we have to keep track of the old value of x needed in $z := x$. For a general transition with a guard and an assignment we perform the computation of the guard described previously and after the last unobservable state we update the variables according to the assignment using

new unobservable states. Time elapsing is unobservable in all the intermediate locations. Thus we have the following theorem:

Theorem 2. *The classes LSWA and SWA are equally expressive in the sense that* $\mathbf{TL}_{LSWA} = \mathbf{TL}_{SWA}$. □

Complexity. We do not consider the addition of integer variables. For a LSWA L with n states, m transitions and k stopwatches, the resulting SWA has at most $3k+3$ stopwatches at most $n + 4m(3k+3)$ states.

3.2 From LHA to LSWA

Let $\mathcal{H} = (N, l_0, V, \Sigma, E, Act, Inv)$ be a LHA (with true invariants). In the following we show how to transform $\mathcal{H}$ into a LSWA accepting the same timed language.

Integer slopes. We first deal with integer *positive* slopes: $l \in N$ is a location of $\mathcal{H}$ and x a variable of V s.t. $Act(l)(x) = u_1 \in \mathbb{N}$. We translate location l as follows:

1. we introduce a fresh variable t which is reset when entering $\tilde{l}_1$ and will measure the time the automaton stays in $\tilde{l}_1$,
2. an auxiliary variable v used to update x,
3. 2 locations $\tilde{l}_i$ where time elapsing is unobservable.

The translation is given in Figure 5 within the dashed rectangle (automaton $\mathcal{S}$).

Note that if we enter $\tilde{l}_2$ with the value x^0 for x and t^0 for t we reach $\tilde{l}$ with $x = x^0 + u_1.t^0$. So we can easily prove the following: if $\nu, \nu' \in \mathbb{R}^V$ and $\tilde{\nu}, \tilde{\nu}' \in \mathbb{R}^{V \cup \{t,v\}}$ s.t. $proj_V(\tilde{\nu}) = \nu$ and $d \in \mathbb{R}_{\geq 0}$ then

$$\langle \tilde{l}, \tilde{\nu} \rangle \stackrel{d}{\Longrightarrow} \langle \tilde{l}', \tilde{\nu}' \rangle \text{ iff } \langle l, \nu \rangle \stackrel{d}{\Longrightarrow} \langle l', \nu' \rangle \quad (1)$$

Then it suffices to replace location l in $\mathcal{H}$ by the sequence described by the automaton $\mathcal{S}$ on Figure 5 to obtain an automaton which accepts the same timed language as $\mathcal{H}$.

To allow for negative slope, we introduce[11] two new variables x^+ and x^-: for locations where the slope of x is $u \geq 0$, we set $\dot{x}^+ = u$ and $\dot{x}^- = 0$. When the slope of x is $u < 0$ we use the slopes $\dot{x}^+ = 0$ and $\dot{x}^- = u$. Then the actual value of x is $x^+ - x^-$. It remains to replace any occurrence of x in $\mathcal{H}$ by $x^+ - x^-$ to obtain a LHA with only positive slopes.

Interval slopes. We now upgrade the previous construction to deal with slopes belonging to closed integer intervals. We assume $Act(l)(x) \in [u_1, u_2], u_1, u_2 \in \mathbb{Z}_{\geq 0}$ (for the case one of the value is negative we split the interval into a positive part and negative part and apply the translation of x in $x^+ - x^-$.) The construction is given on Figure 5 (ignore the dashed transition).

Statement (1) also holds in this case. Finally we obtain the following theorem:

[11] very similar to the handling of negative values.

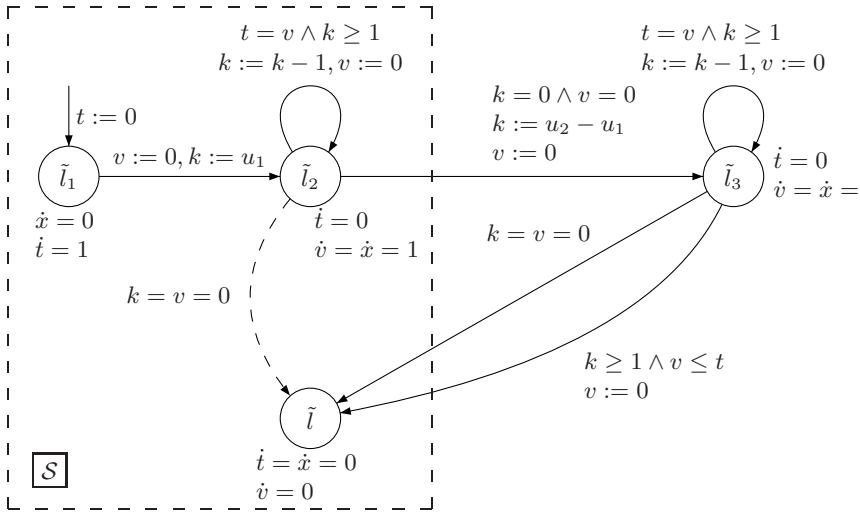


Fig. 5. Translating positive integer slopes

Theorem 3. *The classes LHA and LSWA are equally expressive in the sense that* $\mathbf{TL}_{LHA} = \mathbf{TL}_{LSWA}$. □

This ends the proof of theorem 1.

Complexity. The translation from LHA to LSWA does not require any new variables as we may use those introduced in the translation from LSWA to SWA. We only need to add two states per variable in each location. Then for a LHA with n states, m transitions and k variables the resulting SWA has at most $3k + 3$ variables and at most $2(3k + 3).(n + 4m(3k + 3) + 1)$ states. The translation is then in $\mathcal{O}(k)$ for the number of variables and in $\mathcal{O}(n.m.k^2)$ for the number of states.

4 Application to Symbolic Analysis of Hybrid Automata

Theorem 1 suggests a potential practical application as reachability analysis for a LHA L may effectively be transformed into a reachability analysis on the corresponding SWA $\tilde{L}$. Indeed, it follows from our translation[12] that for any $w \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$

$$\langle l_0, \nu_0 \rangle \overset{w}{\Longrightarrow}^* \langle l, \nu \rangle \text{ iff } \langle \tilde{l}_0, \tilde{\nu}_0 \rangle \overset{w}{\Longrightarrow}^* \langle \tilde{l}, \tilde{\nu} \rangle \tag{2}$$

whenever $proj_V(\tilde{\nu}_0) = \nu_0$ and $proj_V(\tilde{\nu}) = \nu[V/(V^+ - V^-)]$ where $\nu[V/(V^+ - V^-)]$ stands for ν with all the variables $x \in V$ replaced by $x^+ - x^-$. Thus reachability properties of L and $\tilde{L}$ are in a one-to-one correspondence.

[12] $\Longrightarrow^*$ stands for the transitive closure of $\Longrightarrow$.

4.1 Termination of Symbolic Analysis

Despite the close correspondence in (2), the usual forward reachability algorithm [AHH96][13] may behave differently on $\tilde{L}$ and L with respect to termination. Ideally, we want the algorithm to terminate on $\tilde{L}$ whenever it terminates on L. This is not always the case with the rough translation we have given.

A solution[14] is to ensure that every variable has a single representative (x^+, x^-) in each location.

4.2 Approximate Analysis – Preliminary Experiments

We have successfully analyzed a number of LHA using a stopwatch-extended version of the tool UPPAAL. The extension offers an approximate reachability algorithm for SWA obtained as a rather immediate extension of the existing reachability algorithm for TA. In particular, the existing efficient data-structures of UPPAAL have been reused.

When analyzing SWA using difference bounded matrices (DBMs [LLPY97]), the only operation applied during (symbolic) state-space exploration requiring redefinition is that of computing the *future* of a DBM.

The DBM-based algorithm for SWA yields an over-approximation of the reachable state-space as DBMs only allow to encode the difference between 2 variables. In general, an exact characterization of the reachable state-space of a SWA may require constraints involving several variables. Nevertheless, we have implemented a DBM-based, stopwatch-extension of UPPAAL. Combined with our translation from LHA to SWA we have successfully analyzed a number of examples demonstrating that the prize of over-approximation is not too high.

The examples include the water-level monitor [ACH+95] of Figure 1 for which the SWA is depicted in Figure 6. Another example is the *parameterized* version of Fischer's protocol [ACH+95]. Other successfully investigated examples include the gas burner [ACH+95] and the scheduler of [AHH96]. Also, we are currently investigating the tools applicability to case studies such as the ABR protocol [BF99] and the water-tank case study [KLPW] of the VHS project.

5 Conclusion and Future Work

In this paper we have extended hybrid automata with the notion of unobservable delays. We have proved that stopwatch automata (SWA) with unobservable delays (timed automata extended with stopwatches) are as expressive as linear hybrid automata (LHA) in the sense that every language accepted by a LHA is also accepted by a SWA. A practical application of this result is that reachability analysis for LHA may be reduced to reachability analysis for SWA. We have extended the real-time verification tool UPPAAL to SWA reusing its efficient data-structures, allowing for rather large SWA to be analyzed. Combined with

[13] as used in HyTech

[14] we cannot develop the proof in this paper.

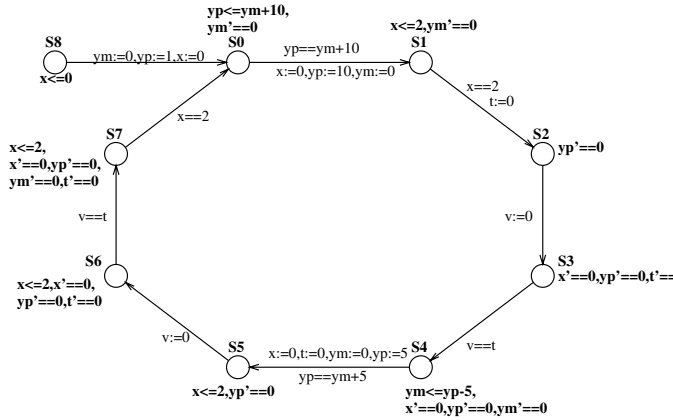


Fig. 6. The SWA water level monitor with UPPAAL

our translation result this offers a completely new method for analyzing LHA. However, the analysis is based on an over-approximation which for some cases may be too coarse to settle a given reachability property.

The translation given in this paper demonstrates that SWA equals LHA in expressive power. However, several alternative translations might have been given and from a practical point of view there are good reasons for looking at such alternative translations as they may be superior in various ways: (1) the translation may well have an impact on the *accuracy* of our tool's approximate analysis on the resulting SWA; (2) from a performance point of view it is important to limit the *complexity* of the translated stopwatch version of a LHA; in particular, translations introducing few additional variables are to be preferred; (3) we want our translation of LHA to SWA to preserve termination when applying non-approximating verification tools such as HyTech.

Our future investigations include: (1) rather than performing an approximate reachability analysis on the SWA (resulting from a translation), it might be possible (and performance-wise superior) to extend the DBM's data-structures to encode exactly the region of a SWA automaton and to perform an exact reachability analysis; (2) as a theoretical aspects we are currently studying the expressive power of unobservable delays for LHA and trying to extend our result to more general classes of hybrid automata (e.g. linear derivatives).

Acknowledgement

The authors would like to thank Gerd Behrmann, Johan Bengtsson and Paul Pettersson for numerous discussions and assistance in making the stopwatch-extension of UPPAAL.

References

ABK+97. E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-Structures for the Verification of Timed Automata. In *Proc. of HART'97, LNCS 1201*, 1997. 140

ACH94. Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. The observational power of clocks. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94: Concurrency Theory, 5th International Conference*, volume 836 of *Lecture Notes in Computer Science*, pages 162–177, Uppsala, Sweden, 22–25August 1994. Springer-Verlag. 139

ACH+95. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 137, January 1995. 138, 140, 141, 149

AD94. Rajeev Alur and David Dill. A theory of timed automata. *Theoretical Computer Science B*, 126:183–235, 1994. 138, 139, 144

AHH96. Rajeev Alur, Thomas A. Henziger, and Hei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions On Software Engineering*, 22(3):181–201, March 1996. 138, 140, 149

AP94. A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 95–104, Standford, California, USA, June 1994. Springer-Verlag. 138

AR95. A. Bouajjani and R. Robbana. Verifying omeg-regular properties for a subclass of linear hybrid systems. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 437–450, Liege, Belgium, July 1995. Springer Verlag. 138

BDFP00. Patricia Bouyer, Christine Dufourd, Eric Fleury, and Antoine Petit. Decidable updatable timed automata are bisimilar to timed automata. Research report, LSV, ENS Cachan, 2000. Forthcoming. 144

BDGP98. B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticæ*, 36:145–182, 1998. 139

BF99. B. Bérard and L. Fribourg. Automated verification of a parametric real-time program: the ABR conformance protocol. In *Proc. 11th Int. Conf. Computer Aided Verification (CAV'99), Trento, Italy, July 1999*, volume 1633 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 1999. 149

BGP96. Béatrice Berard, Paul Gastin, and Antoine Petit. On the power of non-observable actions in timed automata. In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *lncs*, pages 257–268, Grenoble, France, 22–24 February 1996. Springer. 139

BLP+99. Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *11th Computer-Aided Verification*, Trento, Italy, July 1999. 140

Cer92. K. Cerans. *Algorithmic problems in analysis in analysis of real-time specifications*. PhD thesis, University of Latvia, 1992. 138

DGP97. Volker Diekert, Paul Gastin, and Antoine Petit. Removing ε-transitions in timed automata. In R. Reischuk, editor, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science 1997*, number 1200 in Lecture Notes in Computer Science, pages 583–594, Berlin-Heidelberg-New York, 1997. Springer. 139

Hen96. Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press. 138, 140

HK96. Thomas A. Henzinger and Peter W. Kopke. State equivalences for rectangular hybrid automata. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 530–545, Pisa, Italy, 26–29 August 1996. Springer-Verlag. 139

HK97. Thomas A. Henzinger and Peter W. Kopke. Discrete-time control for rectangular hybrid automata. In Pierpaolo Degano, Robert Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 582–593, Bologna, Italy, 7–11 July 1997. Springer-Verlag. 139

HKPV98. Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, August 1998. 138, 139

HKWT95. Thomas A. Henzinger, Peter W. Kopke, and Howard Wong-Toi. The expressive power of clocks. In Zoltán Fülöp and Ferenc Gécseg, editors, *Automata, Languages and Programming, 22nd International Colloquium*, volume 944 of *Lecture Notes in Computer Science*, pages 417–428, Szeged, Hungary, 10–14 July 1995. Springer-Verlag. 139

JP94. J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 105–117, Standford, California, USA, June 1994. Springer-Verlag. 138

KLPW. K. J. Kristoffersen, K. G. Larsen, P. Pettersson, and C. Weise. Verification of an experimental batch plant. VHS internal document. 149

KPSY93. Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: A class of decidable hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 179–208. Springer-Verlag, 1993. 138

LLPY97. K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium, RTSS'97*. IEEE Computer Society Press, December 1997. 140, 149

LPY97. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal of Software Tools for Technology Transfer*, 1(1/2):134–152, October 1997. 140

Yov97. S. Yovine. Kronos: A Verification Tool for real-Time Systems. *Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997. 140

Optimizing Büchi Automata

Kousha Etessami and Gerard J. Holzmann

Bell Labs, Murray Hill, NJ
{kousha,gerard}@research.bell-labs.com

Abstract. We describe a family of optimizations implemented in a translation from a linear temporal logic to Büchi automata. Such optimized automata can enhance the efficiency of model checking, as practiced in tools such as Spin. Some of our optimizations are applied during preprocessing of temporal formulas, while other key optimizations are applied directly to the resulting Büchi automata independent of how they arose. Among these latter optimizations we apply a variant of fair simulation reduction based on color refinement. We have implemented our optimizations in a translation of an extension to LTL described in [Ete99]. Inspired by this work, a subset of the optimizations outlined here has been added to a recent version of Spin. Both implementations begin with an underlying algorithm of [GPVW95]. We describe the results of tests we have conducted, both to see how the optimizations improve the sizes of resulting automata, as well as to see how the smaller sizes for the automata affect the running time of Spin's explicit state model checking algorithm. Our translation is available via a web-server which includes a GUI that depicts the resulting automata:
http://cm.bell-labs.com/cm/cs/what/spin/eqltl.html

1 Introduction

This paper describes a collection of optimizations implemented in a translation from an extension of linear temporal logic to Büchi automata. Such ω-automata find wide spread use as modeling and specification mechanisms for reactive systems, and form the backbone of a family of model checking tools, such as Spin [Hol97], which are based on explicit state enumeration.

In Spin, a linear temporal logic formula φ, used to specify an undesired property of the system[1], is first converted to a Büchi automaton, A_φ. The accepting runs of A_φ represent executions in which the undesirable property holds. This automaton is "producted" with the system model M, in order to determine whether the system has executions exhibiting this property, as first suggested in [VW86]. The worst case running time of the producting algorithm is $O(|M| \times |A_\varphi|)$. Typically, M is far larger than A_φ, particularly because M itself arises as the product $\Pi_i M_i$ of many state machines M_i describing the concurrent components of the

[1] The fragment of LTL used in Spin normally does not allow the "next" operator. This assures that the property specified is stutter-invariant, and hence enables *partial order reduction*, [HP94].

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 153–168, 2000.

system. Since the size of A_φ is a multiplicative factor in the running time, and yet A_φ is relatively small, it makes sense to put substantial computing effort into minimizing A_φ. Unfortunately, A_φ is nondeterministic and finding a minimal equivalent nondeterministic automaton is a PSPACE-hard problem[2]. Thus, we can not hope in general to obtain an exact optimal automaton without prohibitive running time. Even a log factor approximation to the optimal size can easily be shown to be PSPACE-hard. We must be content with applying efficient algorithms and heuristics which, in practice, tend to yield small automata.

This paper describes such a collection of optimizations. We have implemented these optimizations atop a translation from an extension of LTL called (SI-)EQLTL studied in [Ete99] which allows the expression of precisely all (*stutter-invariant*) ω-regular languages. Our optimizations do not depend on the details of the temporal logic, beyond ordinary aspects of LTL, so readers only interested in LTL can confine their attention to the LTL fragment. There is a translation from this logic to Büchi automata ([Ete99]), based on the translation of LTL due to Gerth, Peled, Vardi, and Wolper [GPVW95]. That algorithm in practice does not incur the worst-case exponential blow-up necessarily incurred by a naive tableaux construction. However, the algorithm alone still produces automata that are sometimes vastly suboptimal. Thus the need for optimization.

Spin's LTL translation, also based on [GPVW95], includes some mild forms of optimization. Inspired by this work it was further modified to incorporate some of the rewrite rules discussed here and others listed in [SB00]. We will compare the results of the EQLTL translation with those obtained with Spin in section 4, both before and after the rewrite rules were added to SPIN.

We partition optimizations into three classes. Figure 1 gives a schematic for the different classes of optimizations. First, before the translation algorithm is applied, the temporal formulas themselves can be simplified and optimized, by applying, e.g., rewrite rules. Secondly, various aspects of the translation algorithm can be "tightened". In this paper we do not discuss any optimizations in this second class. See, e.g., [DGV99]. Third, the resulting automata can be reduced by applying automata theoretic optimizations, including those related to bisimulation and particularly fair simulation reduction.

We assume the reader is familiar with the basic notions of Linear Temporal Logic, and ω-automata. For a general reference on temporal logic the reader is referred to, e.g., [Eme90]. For ω-automata, the reader is referred to [Tho90]. We assume our LTL formulas are defined over a set $P = \{p_1, \ldots, p_n\}$ of propositions, with our word alphabet given by $\Sigma = 2^P$. We use $\varphi \mathrm{V} \psi$ to denote the dual of the until operator, i.e., $\varphi \mathrm{V} \psi = \neg(\neg\varphi \mathrm{U} \neg\psi)$.

In tools such as Spin, Büchi automata use a more concise labeling notation. Rather than having a distinct transition labeled with each character a of the alphabet $\Sigma = 2^P$, the labels consist of boolean formulas over the atomic propositions of P. Formally, denote by $\mathcal{B}(P)$, the set of boolean formulas over the atomic propositions P. We assume a Büchi automaton A is given by $\langle Q, \delta, q_0, F \rangle$. Here Q is a set of states, and $\delta \subseteq (Q \times \mathcal{B}(P) \times Q)$ is the transition relation with

[2] See, e.g., [SM73], from which this result follows, or see also [Koz77].

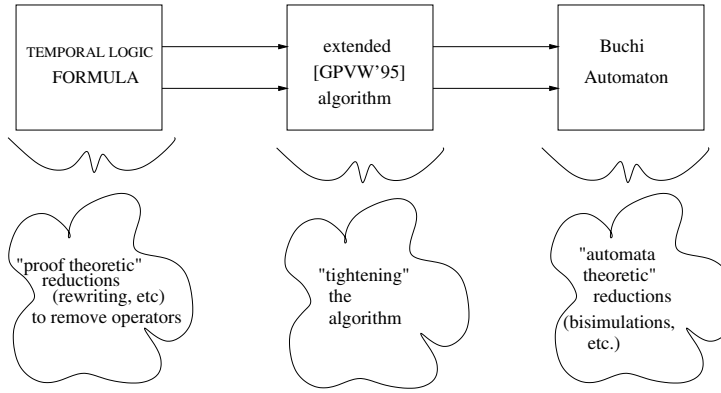


Fig. 1. classes of optimizations

labels given by the more concise formulas rather than individual characters from $\Sigma = 2^P$. Sometimes, the only formulas we will allow on transitions are *terms*, which are conjunctions of literals. $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The language $L(A)$ is defined as the set of those ω-words w which have an accepting *run* in A, where a run on w is the sequence of adjacent states that one can visit while traversing the states according to w, and the run is accepting if it infinitely often goes through a state in F.

In translating from LTL to a Büchi automaton (BA), on the way we pass through a *generalized Büchi automaton* (GBA), which rather than one accepting set F, has a family $\mathcal{F}$ of accepting sets, and a run is then said to be accepting if for each $F \in \mathcal{F}$ the run goes infinitely often through a state of F.

In section 2, we describe our proof-theoretic reductions. Section 3 covers our automata-theoretic reductions. In section 4 we discuss our experimental results. We conclude with discussion is section 5.

Note: A preliminary description of this work was presented in July of 1999 at the SPIN workshop on model checking [SPI99]. Independently, F. Somenzi and R. Bloem developed a similar set of optimizations, in a work [SB00] to appear at CAV'2000. Their translation produces generalized Büchi automata rather than ordinary Büchi automata. Their optimizations include a large number of ordinary rewrite rules, but do not include the general rewrite rules we outline using notions of left-append and suffix closure. In addition to simulation reduction, they outline a clever "reverse" simulation reduction, which we do not have. They kindly provided us with their manuscript prior to this submission. Our EQLTL translation was not modified after receipt of their manuscript, however, a version of SPIN's translation (version 3.3.10), tested here against EQLTL, does include some of the rewrite rules from their paper. We have indicated below in which experiments the additional rules were used, and it which they were disabled. In addition to the mentioned web site for the EQLTL translation, as always, the

source to the SPIN system is available from the Bell Labs web server for further experimentation.

2 Proof Theoretic Reductions and Rewriting

We begin by describing some simple proof theoretic reductions. These consist of a family of rewrite rules that are applied to formulas recursively, reducing the number of operators and/or connectives. Many such rewrite rules with a similar flavor can be found, e.g., in the text by Manna and Pnueli [MP92]. Since the output size in the translation of [GPVW95] is closely correlated with the number of operators, these reductions can pay off well. From now on, we assume all LTL formulas are in *negation normal form*, meaning negations have been "pushed in" so that they are only applied to atomic propositions.

Definition 1. *A language L of ω-words is said to be left-append closed if for all ω-words $w \in \Sigma^\omega$, and $v \in \Sigma^*$: if $w \in L$, then $vw \in L$.*

The property of left-append closed languages we will exploit in order to reduce the size of our automata is the following:

Proposition 1. *Given an formula ψ_1 such that $L(\psi_1)$ is left-append closed, and any formula γ, the following equivalences hold: (1) $\gamma \mathrm{U} \psi_1 \equiv \psi_1$, (2) $\Diamond \psi_1 \equiv \psi_1$.*

The proof of the first equivalence is straightforward. The second follows from the fact that $\Diamond\psi \equiv \mathtt{true}\mathrm{U}\psi$. Unfortunately, there is no efficient procedure to determine if a property defined by a formula is left-append closed (it is in fact PSPACE-complete). However, there is an easy to check *sufficient* condition for being left-append closed:

Definition 2. *The class of pure eventuality formulas are defined as the smallest set of LTL formulas (in negation normal form) satisfying:*

1. *Any formula of the form $\Diamond\varphi$ is a pure eventuality formula.*
2. *Given pure eventuality formulas ψ_1 and ψ_2, and γ an arbitrary formula, each of $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 \mathrm{U} \gamma$, $\Box\psi_1$, $\psi_1 \mathrm{V} \psi_2$, and $\bigcirc\psi_1$, is also a pure eventuality formula.*

Lemma 1. *Every pure eventuality formula φ defines a left-append closed property $L(\varphi)$.*

Note: Every LTL definable property that is left-append closed is definable by a pure eventuality formula: If $L(\varphi)$ is left-append closed, then $L(\varphi) = L(\Diamond\varphi)$, and $\Diamond\varphi$ is a pure eventuality formula. However, there might be formulas that are not pure eventuality formulas and yet still define left-append closed properties. Proofs must be omitted.

Just as we defined left-append closed properties and pure eventuality formulas, we can also consider suffix closed properties and pure universality formulas (formulas are always in negation normal form):

Definition 3. *A language L is* suffix closed *if whenever $w \in L$ and w' is a suffix of w, then $w' \in L$.*

Proposition 2. *For a formula ψ with a suffix closed language $L(\psi)$, and an arbitrary formula γ, the following equivalences hold: (1) $\gamma \mathrm{V} \psi \equiv \psi$, (2) $\Box\psi \equiv \psi$.*

Definition 4. *The class of* purely universal *formulas is defined inductively as the smallest set of formulas satisfying:*

1. *Any formula of the form $\Box\varphi$ is purely universal.*
2. *Given purely universal formulas ψ_1 and ψ_2, and an arbitrary formula γ, any formula of the form: $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 \mathrm{V} \psi_2$, $\Diamond\psi_1$, $\psi_1 \mathrm{U} \psi_2$, and $\bigcirc\psi_1$ is also purely universal.*

Lemma 2. *Every pure universality formula defines a suffix closed property.*

From lemmas 1 and 2, and propositions 1 and 2, the following lemma follows, describing our basic rewrite rules:

Lemma 3. (Basic Operator Reduction Lemma) *For all LTL formulas φ, ψ, and γ, the following equivalences hold:*

1. $(\varphi \mathrm{U} \psi) \wedge (\gamma \mathrm{U} \psi) \equiv (\varphi \wedge \gamma) \mathrm{U} \psi$
2. $(\varphi \mathrm{U} \psi) \vee (\varphi \mathrm{U} \delta) \equiv \varphi \mathrm{U} (\psi \vee \delta)$
3. $\Diamond(\varphi \mathrm{U} \psi) \equiv \Diamond\psi$
4. *Whenever ψ is a pure eventuality formula $(\varphi \mathrm{U} \psi) \equiv \psi$, and $\Diamond\psi \equiv \psi$.*
5. *Whenever ψ is a pure universality formula $(\varphi \mathrm{V} \psi) \equiv \psi$, and $\Box\psi \equiv \psi$.*

Note that in each of the equivalences above the right hand side has at least one fewer temporal operator than the left hand sides. A formula (or subformula) that fits the pattern on the left hand side is replaced by the one on the right. The first three equivalences each have corresponding duals which are also applied.

There are, as you can imagine, many other rules one could use (see, e.g., [MP92]). Our aim has not been to list exhaustively all such rules, but to list a few that have direct impact without excessive cost.

3 Automata Theoretic Reductions

In this section we describe the reduction techniques which are applied directly to the Büchi automata produced by the algorithm of the previous section. The main algorithm in this section is a variant of "fair" simulation reduction, which itself is a natural generalization of bisimulation reduction. Before that algorithm, however, we first describe some other reductions. The reductions covered in this entire section are *complementary*, in the sense that applying one reduction can subsequently enable further gain from another reduction, until a "fixed-point" is reached where we can gain no more.

Simplifying Edge Terms The concise notation for automata, namely term (or formula) labeled transitions, gives us an opportunity to perform some optimizations to reduce the number of edges further, by combining terms, or more generally, reducing formulas. For example, whenever we encounter two transitions: $(q_1, (p_1 \wedge p_2), q_2) \in \delta$ and $(q_1, (p_1 \wedge \neg p_2), q_2) \in \delta$, we can combine the two into one simplified transition: $(q_1, p_1, q_2) \in \delta$, using the obvious propositional rule: $(p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2) \leftrightarrow p_1$. Again, we can perform a vast family of such reductions, this time based on propositional logic proof rules rather than temporal proof rules. We will not elaborate further on these optimizations since they are a standard part of elementary propositional logic. See, e.g., [End72].

Removing "Never Accepting" Nodes This is a trivial optimization, but nevertheless quite important in the context of the other optimizations. In a Büchi automaton A, a node q that does not have any accepting run leading from it can safely be removed without changing $L(A)$. We compute the set of states with a run leading infinitely often to an accepting state by computing the strongly connected components of A, and retaining those states that lead to non-trivial SCCs containing an accept state.

Removing Fixed-Formula Balls This interesting reduction is not correct for ordinary *finite automata*, but is correct for Büchi automata. We assume we have removed "never-accepting" nodes. The idea of the reduction is that, if in a Büchi automaton we are ever stuck in a component from which we can not get out, and the only transition labels in this component are α, and there is some accepting state in the component, then we can treat the entire component as a single accepting state with a self-transition labeled by α. Formally:

Definition 5. *For $\alpha \in \mathcal{B}(P)$, a* **fixed-formula α-ball** *inside a Büchi automaton A is a set $Q' \subseteq Q$ of nodes such that*

1. *α is the unique formula which labels the transitions inside Q', i.e., if $q'_1, q'_2 \in Q'$ and $\delta(q'_1, \xi, q'_2) \in \delta$, then $\xi = \alpha$.*
2. *The nodes Q' form a strongly connected component of the underlying graph G of A, where $V_G = Q$ and $E_G(q_1, q_2) \iff \exists \sigma\ (q_1, \sigma, q_2) \in \delta$.*
3. *There is no transition leaving Q', i.e., no $(q', \xi, q) \in \delta$, where $q' \in Q'$ and $q \notin Q'$.*
4. *Q' contains an accepting state, i.e., $Q' \cap F \neq \emptyset$*

Proposition 3. *Given a Büchi automaton $A = \langle Q, \delta, q_0, F \rangle$, suppose $Q' \subseteq Q$ is a fixed-formula α-ball of A. Let $A' = \langle (Q \setminus Q') \cup \{q_{new}\}, \delta', q'_0, (F \setminus Q') \cup \{q_{new}\} \rangle$, where the transitions involving q_{new} are $(q_1, \xi, q_{new}) \in \delta'$ whenever there was some $q' \in Q'$ such that $(q_1, \xi, q') \in \delta$, and $(q_{new}, \alpha, q_{new}) \in \delta'$, and all transitions inside $Q \setminus Q'$ remain the same. Moreover, $q'_0 = q_{new}$ if $q_0 \in Q'$, and otherwise $q'_0 = q_0$. Then $L(A) = L(A')$.*

Although this looks like a rather specialized reduction, fixed-formula balls are frequently introduced into Büchi automata as a result of the translation from a GBA, A', to a BA, A. Consider, for example, the simple situation where there is an accepting state q with the unique transition $(q, \texttt{true}, q) \in \delta'$ leaving q in A'. Then, if there are k generalized Büchi accepting sets in A', there will be k copies of q, call them $q_0, \ldots, q_{k-1}$, in A forming a k-state loop with $(q_i, \texttt{true}, q_{(i+1 \bmod k)}) \in \delta$. This is a fixed-formula ball which can be collapsed to one state and one transition, potentially enabling further reductions.

3.1 Reductions Based on Bisimulation and Simulation

In this section we describe what, algorithmically, is our most elaborate reduction: a version of fair simulation reduction, with an algorithm based on color refinement (see, e.g., [HU79]). There are several distinct notions of fair simulation in the literature (see, e.g., [HKR97]). The "weaker" the notion, the more reduction it potentially enables. The notion we chose to implement is by no means the weakest available notion, but its advantage is that it admits an easy to implement and reasonably efficient algorithm which is a very natural modification of the ordinary color refinement algorithm. [HKR97] describe a weaker notion of fair simulation, and give a polynomial time algorithm for computing the relation. But their algorithm, which employs an algorithm of [HHK95] for computing a maximal simulation relation and then resorts to tree automata and their emptiness problem to deal with fair simulation, is apparently impractical, requiring a worst case running time of $O(n^6)$.[3]

Review: Basic Partition Refinement We first review the standard bisimulation reduction algorithm (see, e.g.,[KS90]), based on color-refinement partitioning of the states, accounting now for the fact that edges are labeled by terms rather than individual characters of the alphabet. The fact that labels are terms rather than arbitrary formulas helps us when we switch to simulation reduction.

The basic algorithm is given in Figure 2. On input $A = \langle Q, \delta, q_0, F\rangle$, the algorithm proceeds as follows to create an output A', such that $L(A) = L(A')$. It creates a coloring function $C^i : Q \mapsto \{1, \ldots, |Q|\}$ which initially incorporates the acceptance condition by assigning one color to accept states and a different color to reject states. It is refined after each iteration i until a fixed point is reached, namely, no new colors are created. We use $C^i(Q)$ to denote the set of colors after round i. Although during the algorithm our notation assigns *tuples* as colors, these tuples can easily converted to numbers again after each iteration, by the usual lexicographic sorting and renaming.

This algorithm is correct for NFAs as well as Büchi automata. For NFAs this is because the following inductive invariant is maintained by the algorithm. Let $S_i^q = \{s \in \cup_{j \le i} \Sigma^j \mid q \overset{s}{\rightsquigarrow} q'' \, , q'' \in F\}$ be the set of strings of length at most i with which one can reach an accept state from state q.

[3] There is a typographical error in the conference version of [HKR97] which indicates a running time $O(n^4)$, but this typo has been corrected in more recent versions.

proc *BasicBisimReduction*(A) $\equiv$
/* Initialize: $\forall q \in Q\ C^{-1}(q) := 1$, and $\forall q \in F\ C^0(q) := 1$, $\forall q \in Q \setminus F\ C^0(q) := 2$. */
$i := 0$;
while $|C^i(Q)| \neq |C^{i-1}(Q)|$ **do**
$i := i + 1$;
foreach $q \in Q$ **do**
$C^i(q) := \langle C^{i-1}(q), \cup_{(q,\tau,q') \in \delta} \{(C^{i-1}(q'), \tau)\} \rangle$
od
Rename color set $C^i(Q)$, with $\{1, \ldots, |C^i(Q)|\}$, using lexicographic ordering.
od
$C := C^i$; **return** $A' := \langle Q' := C(Q), \delta', q'_0 := C(q_0), F' := C(F) \rangle$;
/* δ' defined so that $(C(q_1), \tau, C(q_2)) \in \delta'$ */
/* if and only if $(q_1, \tau, q_2) \in \delta$ for $q_1, q_2 \in Q$ */

Fig. 2. Basic Bisimulation Reduction Algorithm

Proposition 4. *For all* $q, q' \in Q$, *if* $C^i(q) = C^i(q')$ *then* $S_i^q = S_i^{q'}$.

After at most $|Q|$ iterations, the color refinement reaches a fixed point, and thus if $C(q) = C(q')$ at this point, then $\forall l \geq 0$, $S_l^q = S_l^{q'}$. For ω-automata, in particular for Büchi automata, the proof is only slightly more subtle.

Theorem 1. *(see, e.g., [KS90]) Given a Büchi automaton A, the* BasicBisimReduction *algorithm produces a Büchi automaton* A' *such that* $L(A) = L(A')$.

Modifying the Refinement Algorithm, Simulation: We first illustrate the weakness of the ordinary color refinement. Consider the automaton in Figure 3. From the initial state q_0, reading p_1 in the first character of the input, w_0, we can go directly to the accepting state q_1, in which we can stay regardless of the rest of the ω-word, $w_1 w_2 \ldots$. Therefore, the transition from q_0 to q_2 labeled by the term $p_1 \wedge p_2$ is completely redundant. Thus, in fact we can eliminate both the state q_2, and all the subsequent states reachable from q_2 without affecting the language of the automaton. The trouble with ordinary bisimulation reduction is that it doesn't recognize this situation, nor the more complicated situations like it where one transition from a state "subsumes" another.

Using simulation reduction instead of bisimulation immediately remedies this weakness. Algorithmically, this corresponds to not just partitioning the states into equivalence classes, but maintaining a quasi-order on the states (a reflexive, transitive relation), which essentially defines which classes of states "subsume" others.

Our algorithm amounts to computing a strong version of a fair simulation relation, one that works for both finite and ω-automata.(This kind of fair simulation is termed *direct* simulation and used in the independent work of [SB00], and as they point out, had been previously used in [DHWT91].) As pointed out before, algorithms with better complexity ([HHK95]), as well as algorithms that

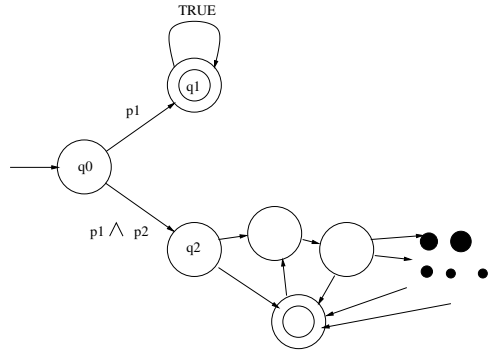


Fig. 3. Why simulation based reduction is better than bisimulation based reduction

yield greater reduction but have worst complexity ([HKR97]) exist in the recent literature. We chose our implementation based on its simplicity and relatively good performance in practice.

The algorithm we implement is a natural revision of the basic partition algorithm in Figure 2, but rather than refining a partition of the vertices, inductively refines a quasi-order on the vertices. Instead of maintaining the entire quasi-order, we can keep a partial order, $po^i_{\leq}(x, y)$, on the equivalence (color) classes that exist in the quasi-order after round i. We associate with the neighbors q' of each node q, such that for some σ, $(q, \sigma, q') \in \delta$, a **neighbor i-type** $(C^i(q'), \sigma)$. Consider the i-types $(C^i(q''), \tau)$ and $(C^i(q'), \sigma)$ where τ and σ are terms labeling transitions. We say that $(C^i(q'), \sigma)$ *i-dominates* $(C^i(q''), \tau)$ if $po^i_{\leq}(C^i(q''), C^i(q'))$ and σ is a subterm of τ (i.e., as a boolean formula τ implies σ). For a node q, and an edge $(q, \tau, q') \in \delta$, we say that the pair $(C^i(q'), \tau)$ is *i-maximal* for q, if there is no q'' with $(q, \sigma, q'') \in \delta$, such that $(C^i(q''), \sigma)$ i-dominates $(C^i(q'), \tau)$. Given $q \in Q$, let the set of i-maximal neighbor i-types of q be given by $\mathbf{N^i(q)} = \{(C^i(q'), \tau) \mid (q, \tau, q') \in \delta$ and $(C^i(q'), \tau)$ is *i-maximal*$\}$. We will say that $N^i(q')$ *i-dominates* $N^i(q)$, if for every $(c, \tau) \in N^i(q)$ there is a pair $(c', \sigma) \in N^i(q')$ such that (c', σ) i-dominates (c, τ).

Now, consider the algorithm in Figure 4. The algorithm first initializes the coloring and the partial order, so that accept nodes get a "greater" color than reject nodes. It then iteratively refines this partition, using the fact that, in the interim, when one neighbor is dominated by another, only the dominating neighbor needs to be considered in the next round of coloring. The partial order itself is upgraded using the fact that, if the neighbors of color class c dominate the neighbors of class c', and the "old color" of class c is greater than the "old color" of c', then in the new coloring, c dominates c'.

The algorithm halts when, neither the number of colors nor the partial order on the colors changes from one iteration to the next. There is however, a trick used to speed up this check: rather than checking that the entire partial order

proc *StrongFairSimulationReduction*(A) $\equiv$
/* Initialize: $\forall q \in Q\ C^{-1}(q) := 1$, and $\forall q \in F\ C^0(q) := 1$, $\forall q \in Q \setminus F\ C^0(q) := 2$. */
/* Initialize the partial order on colors: */
$po^0_{\leq}(2,1) := true;\ po^0_{\leq}(1,1) := true;\ po^0_{\leq}(2,2) := true;\ po^0_{\leq}(1,2) := false;$
$i := 0;$
while $|C^i(Q)| \neq |C^{i-1}(Q)|$ or $|po^i| \neq |po^{i-1}|$ **do**
$i := i + 1;$
foreach $q \in Q$ **do** $C^i(q) := \langle C^{i-1}(q), N^{i-1}(q)\rangle$ **od**
/* now we update the partial order, creating $po^i_{\leq}$ */
foreach $(c^i_1 = \langle c^{i-1}_1, N^{i-1}_1\rangle) \in C^i(Q)$ **do**
foreach $(c^i_2 = \langle c^{i-1}_2, N^{i-1}_2\rangle) \in C^i(Q)$ **do**
if $po^{i-1}_{\leq}(c^{i-1}_2, c^{i-1}_1)$ and N^{i-1}_1 $(i-1)$-*dominates* N^{i-1}_2
then $po^i_{\leq}(c^i_2, c^i_1) :=$ `true`;
else $po^i_{\leq}(c^i_2, c^i_1) :=$ `false`;
fi
od
od
Rename color set $C^i(Q)$, with $\{1, \ldots, |C^i(Q)|\}$, using lexicographic ordering.
Adapt $po^i_{\leq}$ to these renamed colors.
od
$C := C^i$; **return** $A' := \langle Q' := C(Q), \delta', q'_0 := C(q_0), F' := C(F)\rangle$;
/* δ' defined so that $(C(q_1), \tau, C(q_2)) \in \delta'$ */
/* if and only if $(C(q_2), \tau) \in N^i(q_1)$ */.

Fig. 4. Reduction based on strong fair simulation

remains the same, all we need to do is check that the number of pairs, $|po^i|$, in the partial order do not change. This is so because the effect of the loop on the underlying quasi-order is monotone, meaning edges are only being removed from the quasi-order during this fixed-point computation and never added. The correctness of the algorithm for finite and ω-automata, respectively are the following two claims (we omit proofs):

Proposition 5. *If* $C^i(q) \leq C^i(q')$ *then* $S^i_q \subseteq S^i_{q'}$

Theorem 2. *Given a Büchi or finite automaton A, the StrongFairSimulationReduction algorithm constructs A′, such that* $L(A) = L(A')$.

Complexity: An upper bound for running time of the algorithm can be obtained as follows: the main while loop dominates the running time. It can iterate at most m times, where m is the number of transitions in the original automaton. The body of the loop requires $O(n+k^2c)$ time, where n is the number of states in the input automaton, k is the number of states in the resulting output automaton, and c is the time required to compute the i-dominates relation on neighbor sets. Thus, the total worst case running time is bounded by $O(mk^2c + mn)$.

This is by no means the best possible: [HHK95] show that a maximal simulation relation can be computed in time $O(mn)$, and the same algorithm can

be modified to accommodate acceptance conditions as in our setting. The main benefit of our algorithm is that it is easy to implement. Also, it rarely requires the worst case time: often the while loop will iterate far fewer than m times. In practice, we find this algorithm runs fairly quickly.

4 Experimental Results

We now describe a number of experiments we have conducted on three different implemented translations to Büchi automata:

1. **GPWV95:** The first translation is based solely on the algorithm of [GPVW95], implemented in ML by Doron Peled. To be compatible, the resulting GBA from this translation is converted to a BA using a straightforward construction, and the labels are moved from states to transitions.
2. SPIN **version 3.3.10:** SPIN's LTL translation is based on [GPVW95], with a number of relatively simple additional optimizations in the third phase of the algorithm. They consist primarily of merging identical states, and removing simple cases of two-node *balls*. This version of SPIN uses some additional rewrite rules from the current paper and some taken from [SB00]. For comparison, we also include in Table 1 results, marked SPIN-, for the same version of the tool with all new rewrite rules and optimizations *disabled*.
3. **EQLTL:** This is the algorithm which includes all the optimizations we have mentioned in this paper, implemented in ML on top of the [GPVW95] algorithm. Although the logic used in this algorithm, (SI-)EQLTL ([Ete99]), allows us to express strictly more properties than LTL, namely all (stutter-invariant) ω-regular properties, we confine our experiments to the standard LTL fragment in order to be able to compare our results to the other two translations. Table 1 includes measurements for 4 versions of EQLTL with specific classes of optimization enabled or disabled. The version titled "EQLTL" performs all reduction, "EQLTL-auto" performs only the automata theoretic reductions, "EQLTL-rewr" only the rewrite rules, and "EQLTL-none" performs only bare-bones trivial reductions, like removing dead states after converting from a GBA to a BA.

We performed two sets of measurements. In the first set we randomly generated 500 LTL formulas with up to 10 temporal operators and/or boolean connectives, and 3 propositions. This set is generated by randomly choosing grammar rules from LTL's context-free grammar, expanding a formula up to its maximum length. Similar random formula generation was also used in [DGV99], [Tau99], and [SB00]. For comparison, the second set includes 12 hand selected formulas, shown below Figure 5, including many that are in common use.

Table 1 shows the total number of states and transitions generated by the three main translations for the 500 random formulas, and the averages for each. The graph on the upper left in Figure 5 shows, along the x axis the 500 formulas, sorted by the number of states in the output of the EQLTL algorithm for each formula, and on the y axis the number of states for each formula for EQLTL.

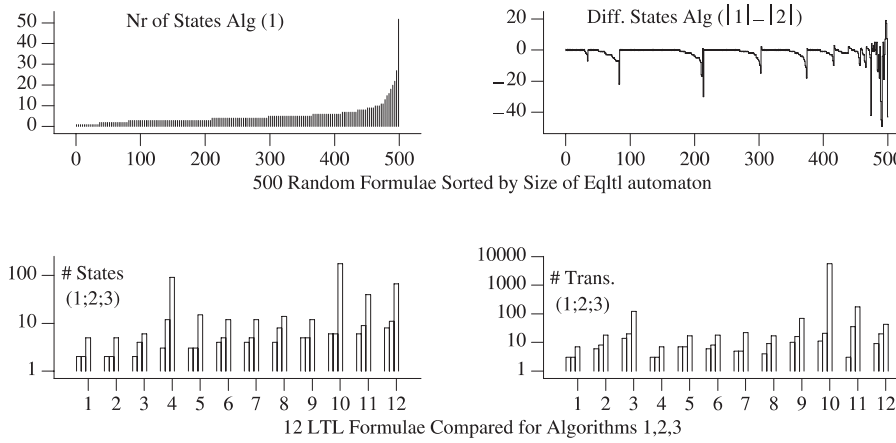


Fig. 5. Automata Sizes (1=Eqltl, 2=Spin, 3=Gpvw)

1. p U (q ∧ □ r)
2. p U (q ∧ X(r U s))
3. p U (q ∧ X(r ∧ (◊ (s ∧ X(◊ (t ∧ X(◊ (u ∧ X◊ v))))))))
4. ◊ (p ∧ X □ q)
5. ◊ (p ∧ X(q ∧ X(◊ r)))
6. ◊ (q ∧ X(p U r))
7. (◊ □ q) ∨ (◊ □ p)
8. □ (p → (q U r))
9. ◊ (p ∧ X◊ (q ∧ X◊ (r ∧ X◊ s)))
10. □◊ p ∧ □◊ q ∧ □◊ r ∧ □◊ s ∧ □◊ t
11. (p U q U r) ∨ (q U r U p) ∨ (r U p U q)
12. □ (p → (q U (□ r ∨ □ s)))

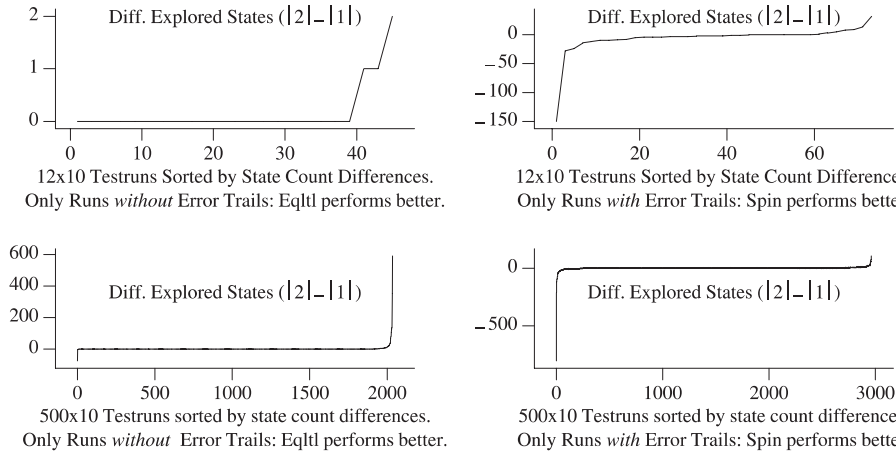


Fig. 6. Model Checking Performance (1=Eqltl, 2=Spin)

Table 1. 500 random and 12 specific formulas

	Random formulas		12 formulas		version
	states	*average*	*states*	*average*	
GPVW	13400	26.8	183	15.25	basic algorithm
Spin-	4069	8.1	77	6.4	minus rewrite rules
Spin	3419	6.8	72	6	with rewrite rules
EQLTL-none	6473	12.9	117	9.8	no non-trivial optimizations
EQLTL-rewr	5599	11.2	114	9.5	rewrite rules only
EQLTL-auto	2709	5.4	51	4.3	automata optimizations only
EQLTL	2564	5.1	49	4.1	all optimizations
	transitions	*average*	*transitions*	*average*	
GPVW	42185	84.37	2321	193.4	basic algorithm
Spin-	13982	27.9	161	13.4	minus rewrite rules
Spin	11245	22.5	156	13	with rewrite rules
EQLTL-none	18511	37.0	232	19.3	no non-trivial optimizations
EQLTL-rewr	14774	29.5	228	19.0	rewrite rules only
EQLTL-auto	4952	11.0	86	7.2	automata optimizations only
EQLTL	4952	9.9	81	6.75	all optimizations

The graph on the upper right in this figure shows the formulas in the same order, but with on the y axis the difference between the numbers of states in the automata generated by the most recent version of Spin and by EQLTL. A value of zero means that the two automata have the same number of states. Negative numbers mean that the EQLTL automaton is smaller than the Spin automaton by the amount shown. In most cases the automata are fairly close in size, but there are notable exceptions, predominantly in favor of the EQLTL algorithm.

The graph in the lower left of Figure 5 shows the number of states that is generated by the three algorithms for the 12 hand selected LTL formulas from our second test set. Three bars are shown for each formula, the first give the number of states generated by EQLTL, the second by Spin, and the third by GPVW. The graph in the lower right of the figure shows the same data for the number of transitions in each automaton. In both cases the y axis is plotted on a logarithmic scale.

We have also measured how the sizes of the automata affect the running time of Spin's model checking algorithm, to test the hypothesis that reducing automata size should help reduce expected running time of the model checker. Each formula was tested against randomly generated systems with 50 states each. Propositional values were changed randomly along the edges in each of these systems, similar to [Tau99]. A standard model checking run was performed for each formula from our test set against 10 such randomly generated systems. The model checking runs halted when the formula was either shown to be satisfied or not satisfied for the system. Figure 6 compares the number of combined states that were explored in the product of the automaton with the system for each

model checking run. The graphs show the *difference* between the number of states explored for the EQLTL automaton and the number of states explored by Spin's automaton on the y axis. A negative number (on the left side of the graphs) represents the cases where more states were explored for the EQLTL automaton than for Spin's automaton. Positive numbers (on the right side of the graphs) represent cases where the EQLTL automaton allowed the model checker to explore fewer states.

The graphs illustrate that for system models without violating runs the number of states explored, as expected, is larger when the size of the property automaton is larger (this must be so because without violating paths all states will be explored). However, somewhat surprisingly, when there are violating runs, the benefit of smaller property automata is not clear. In particular, although the EQLTL translation produces smaller automata, in most cases it explores as many states as Spin, and in a few it explores more. We have no adequate explanation for this phenomenon. Since the Spin automata on average have about twice the number of transitions of the EQLTL automata, it is possible that for a given number of states an increase in the number of transitions can contribute positively to model checking performance, but this needs to be explored.

The running times for all the EQLTL optimizations are in most cases faster than Spin's and GPVW's.

5 Conclusions

The realm of possible heuristic optimizations for Büchi automata is vast: the PSPACE-hardness of finding the optimal (or even approximately optimal) sized automaton leaves an opening for any number of optimizations. We have outlined those optimizations we have found most useful. There is, however, plenty of room for improvement, and we anticipate adding incremental improvements to our translation in the future.

The optimized (SI-)EQLTL translation is available, with a GUI that draws the resulting automaton and provides the corresponding Spin code at `http://cm.bell-labs.com/cm/cs/what/spin/eqltl.html`. The implementation is in the language ML, based on an implementation of the algorithm of [GPVW95] by Doron Peled. Spin's LTL translation is written in C.

Acknowledgement

Thanks to Doron Peled for providing us his original ML code for the [GPVW95] algorithm, to Fabio Somenzi and Roderic Bloem for providing us their unpublished manuscript [SB00], and to the anonymous reviewers for helpful comments.

References

DGV99. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proc. of 11th Int. Conf. on Computer*

Aided Verification (CAV99), number 1633 in LNCS, pages 249–260, 1999. 154, 163

DHWT91. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relations. In *Proceedings of CAV'91*, pages 329–341, 1991. 160

Eme90. E. A. Emerson. Temporal and modal logics. In J. van Leeuwen, editor, *Handbook of Theo. Comp. Sci.*, volume B, pages 995–1072. Elsevier, 1990. 154

End72. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972. 158

Ete99. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 236–248, 1999. 153, 154, 163

GPVW95. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, 1995. 153, 154, 156, 163, 166

HHK95. M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of 36th IEEE Symp. on Foundations of Comp. Sci. (FOCS'95)*, pages 453–462, 1995. 159, 160, 162

HKR97. T. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *Proc. of 9th Int. Conf. on Concurrency Theory (CONCUR'97)*, number 1243 in LNCS, pages 273–287, 1997. 159, 161

Hol97. G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 153

HP94. G. J. Holzmann and D. Peled. An improvement in formal verification. In *7th Int. Conf. on Formal Description Techniques*, pages 177–194, 1994. 153

HU79. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Formal Languages and Computation*. Addison-Wesley, 1979. 159

Koz77. D. Kozen. Lower bounds for natural proof systems. In *18th IEEE Symposium on Foundations of Computer Science*, pages 254–266, 1977. 154

KS90. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990. 159, 160

MP92. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer-Verlag, Berlin/New York, 1992. 156, 157

SB00. F. Somenzi and R. Bloem. Efficient büchi automata from ltl formulae. In *Proceedings of 12th Int. Conf. on Computer Aided Verification*, 2000. 154, 155, 160, 163, 166

SM73. L. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: preliminary report. In *Proceedings of 5th ACM Symposium on Theory of Computing*, pages 1–9, 1973. 154

SPI99. *SPIN workshop on theo. aspects of model checking*, July 1999. Trento, Italy. 155

Tau99. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into büchi automata. In *Workshop Concurrency, Specifications, and Programming*, pages 251–262, Warsaw, Sept. 1999. 163, 165

Tho90. W. Thomas. *Handbook of Theoretical Computer Science, volume B*, chapter Automata on Infinite Objects, pages 133–191. MIT Press, 1990. 154

VW86. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Dexter Kozen, editor, *First Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 322–331, 1986. 153

Generalized Model Checking: Reasoning about Partial State Spaces*

Glenn Bruns and Patrice Godefroid

Bell Laboratories, Lucent Technologies
263 Shuman Boulevard, Naperville, IL 60566, U.S.A.
{grb,god}@bell-labs.com

Abstract. We discuss the problem of model checking temporal properties on partial Kripke structures, which were used in [BG99] to represent incomplete state spaces. We first extend the results of [BG99] by showing that the model-checking problem for any 3-valued temporal logic can be reduced to two model-checking problems for the corresponding 2-valued temporal logic. We then introduce a new semantics for 3-valued temporal logics that can give more definite answers than the previous one. With this semantics, the evaluation of a formula ϕ on a partial Kripke structure M returns the third truth value $\perp$ (read "unknown") only if there exist Kripke structures M_1 and M_2 that both complete M and such that M_1 satisfies ϕ while M_2 violates ϕ, hence making the value of ϕ on M truly unknown. The partial Kripke structure M can thus be viewed as a partial solution to the satisfiability problem which reduces the solution space to complete Kripke structures that are more complete than M with respect to a completeness preorder. This *generalized model-checking problem* is thus a generalization of both satisfiability (all Kripke structures are potential solutions) and model checking (a single Kripke structure needs to be checked). We present algorithms and complexity bounds for the generalized model-checking problem for various temporal logics.

1 Introduction

Many approaches have been proposed to deal with the problem of model checking large state spaces, among them partial-order methods, symbolic verification, and abstraction techniques. But often these approaches do not suffice. Lacking a means to model check the entire state space of a system, one may settle for considering only part of the state space, hoping errors can be found there. The selection of a part of the state space can be done in various ways: randomly, breadth-first, according to general heuristics (i.e. prefer states in which process queues are filling), or according to what could be called "ad-hoc abstraction", in which one ignores certain states, or certain details of states, believed irrelevant to the property of interest.

* This is an extended abstract, with proofs omitted. For the full version of the paper see www.bell-labs.com/~{grb,god}

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 168–182, 2000.

The result of applying traditional model checking to such a "partial state space" will show that a property does or does not hold. However, one would like the partiality of the state space to be taken into account. Thus, a search should return *true* if the property definitely holds because of information in the partial state space, *false* if the property definitely fails to hold because of information in the partial state space, and *unknown* (denoted $\perp$) if the truth or falsity of the property depends on information not contained in the partial state space.

In [BG99] *partial Kripke structures* were used to represent partial state spaces. A *completeness preorder* was defined for partial Kripke structures, a 3-valued temporal logic for reasoning about such structures was defined, and this logic was shown to characterize the completeness preorder. A corollary of the characterization result is that interpreting a formula on a more complete Kripke structure will give a more definite result.

In this paper, we first extend the results of [BG99] by showing that the model-checking problem for 3-valued temporal logic can be reduced to *two* model-checking problems for the corresponding 2-valued temporal logic. Specifically, we show that any partial Kripke structure can be completed into two "extreme" complete Kripke structures, called the *optimistic* and *pessimistic* completions, and that model-checking a partial Kripke structure can be reduced to model-checking its optimistic and pessimistic completions. This implies that the problem of model-checking partial state spaces can be solved using existing tools.

The 3-valued semantics of [BG99] does not behave exactly as one might expect. Informally, one would like the model checking algorithm to return value $\perp$ for a partial Kripke structure and a formula only if there exist a more complete structure for which the formula is *true* and another more complete structure for which the formula is *false*. However, by the 3-valued semantics of [BG99], it is possible to get result $\perp$ even though only results $\perp$ and *true* can be obtained when making the given structure more complete.

We introduce in this paper a new semantics that gives value $\perp$ only when there exist a more complete structure for which the formula holds *and* a more complete structure for which the formula does not hold. The main question we consider is whether model checking under this semantics is more expensive than model checking under the semantics given in [BG99]. We give algorithms and complexity bounds for several temporal logics showing that it is indeed more expensive. The problem of model checking under the new semantics is interesting on its own because it generalizes the problems of model checking and of satisfiability checking. Solving the problem means determining if some structure more complete than a given structure satisfies a formula. If the given structure is fully incomplete, the problem reduces to satisfiability checking. If the given structure is fully complete, the problem reduces to model checking.

2 Two-Valued Modal Logics

In this section we briefly review Kripke structures and propositional modal logic (PML). Let P be a nonempty finite set of *atomic propositions*.

Definition 1. *A* Kripke structure M *is a tuple* $(S, L, \mathcal{R})$*, where* S *is a set of* states*,* $L : S \times P \rightarrow \{true, false\}$ *is an* interpretation *that associates a truth value in* $\{true, false\}$ *with each atomic proposition in* P *for each state in* S*, and* $\mathcal{R} \subseteq S \times S$ *is a transition relation on* S*.*

For technical convenience we require of every Kripke structure that its transition relation be *total*, i.e., that every state has an outgoing transition. We also assume that the number of outgoing transitions from a state is finite. We write (M, s) to refer to state s of Kripke structure M, or just s if the structure to which s belongs is clear. Also, we write $s \rightarrow s'$ as shorthand for $(s, s') \in \mathcal{R}$.

PML (e.g, see [Var97b]) is propositional logic extended with the modal operator $\Box$. Intuitively, formula $\Box\, \phi$ holds at a state s if ϕ holds at all states that can be reached from s in a single transition. Formulas of PML have the following abstract syntax, where p ranges over P:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \Box\, \phi$$

Definition 2. *The* satisfaction *of a formula* ϕ *of PML in a state* s *of a Kripke structure* $M = (S, L, \mathcal{R})$*, written* $(M, s) \models \phi$*, is defined inductively as follows:*

$$\begin{array}{ll}
(M, s) \models p & \text{if } L(s, p) = true \\
(M, s) \models \neg\phi & \text{if } (M, s) \not\models \phi \\
(M, s) \models \phi_1 \wedge \phi_2 & \text{if } (M, s) \models \phi_1 \text{ and } (M, s) \models \phi_2 \\
(M, s) \models \Box\, \phi & \text{if } (M, s') \models \phi \text{ for all } s' \text{ such that } s \rightarrow s'
\end{array}$$

We define $\phi_1 \vee \phi_2$ as $\neg(\neg\phi_1 \wedge \neg\phi_2)$ and $\Diamond\, \phi$ as $\neg(\Box\, \neg\phi)$.

PML can be used to define an equivalence relation on states of Kripke structures: two states are equivalent if they satisfy the same set of formulas of the logic. It is well known [HM85] that the equivalence relation induced in this way by PML coincides with the notion of bisimulation relation [Mil89,Par81].

Definition 3. *Let* $M_1 = (S_1, L_1, \mathcal{R}_1)$ *and* $M_2 = (S_2, L_2, \mathcal{R}_2)$ *be Kripke structures. The* bisimilarity *relation* $\sim$ *is the greatest relation* $\mathcal{B} \subseteq S_1 \times S_2$ *such that* $(s_1, s_2) \in \mathcal{B}$ *implies the following:*

- $\forall p \in P : L(s_1, p) = L(s_2, p)$,
- *if* $s_1 \rightarrow s_1'$ *then there is some* $s_2' \in S_2$ *such that* $s_2 \rightarrow s_2'$ *and* $(s_1', s_2') \in \mathcal{B}$*, and*
- *if* $s_2 \rightarrow s_2'$ *then there is some* $s_1' \in S_1$ *such that* $s_1 \rightarrow s_1'$ *and* $(s_1', s_2') \in \mathcal{B}$*.*

The following result (from [HM85]) shows that PML *logical characterizes* the bisimulation preorder: two states are bisimilar just if they satisfy the same set of PML formulas.

Theorem 1. *[HM85] Let* $M_1 = (S_1, L_1, \mathcal{R}_1)$ *and* $M_2 = (S_2, L_2, \mathcal{R}_2)$ *be Kripke structures such that* $s_1 \in S_1$ *and* $s_2 \in S_2$*, and let* Φ *denote the set of all PML formulas. Then*

$$s_1 \sim s_2 \text{ iff } (\forall \phi \in \Phi : [(M_1, s_1) \models \phi] = [(M_2, s_2) \models \phi]).$$

3 Partial Kripke Structures and Three-Valued Modal Logic

3.1 Background

In this section we present background information on partial Kripke structures and a 3-valued modal logic interpreted over them. The definitions and results of this section come from [BG99].

We model partial state spaces as partial Kripke structures, in which propositions can take a third truth value $\perp$. We then define a 3-valued modal logic whose semantics is defined with respect to partial Kripke structures. We proceed by presenting an equivalence relation and preorder implicitly defined by this logic. As before, let P be a nonempty finite set of atomic propositions.

Definition 4. *A* partial Kripke structure M *is a tuple* $(S, L, \mathcal{R})$*, where* S *is a set of* states*,* $L : S \times P \to \{true, \perp, false\}$ *is an* interpretation *that associates a truth value in* $\{true, \perp, false\}$ *with each atomic proposition in* P *for each state in* S*, and* $\mathcal{R} \subseteq S \times S$ *is a transition relation on* S*.*

A standard Kripke structure is a special case of partial Kripke structure. We sometimes refer to standard Kripke structures as *complete* Kripke structures to emphasize that no propositions within them take value $\perp$.

In interpreting propositional operators on partial Kripke structures we use Kleene's strong 3-valued propositional logic [Kle87]. In this logic $\perp$ is understood as "unknown whether true or false". A simple way to define conjunction (resp. disjunction) in this logic is as the minimum (resp. maximum) of its arguments under the *truth ordering* on truth values, in which *false* is less than $\perp$ and $\perp$ is less than *true*. We write 'min' and 'max' for these functions, and extend them to sets in the obvious way, with $\min(\emptyset) = true$ and $\max(\emptyset) = false$. We define negation using the function 'comp' that maps *true* to *false*, *false* to *true*, and $\perp$ to $\perp$. These functions give the usual meaning of the propositional operators when applied to values *true* and *false*.

Definition 5. *The value of a formula* ϕ *of 3-valued PML in a state* s *of a partial Kripke structure* $M = (S, L, \mathcal{R})$*, written* $[(M, s) \models \phi]$*, is defined inductively as follows:*

$$
\begin{aligned}
[(M, s) \models p] &= L(s, p) \\
[(M, s) \models \neg\phi] &= comp([(M, s) \models \phi]) \\
[(M, s) \models \phi_1 \wedge \phi_2] &= min([(M, s) \models \phi_1], [(M, s) \models \phi_2]) \\
[(M, s) \models \Box\, \phi] &= min(\{[(M, s') \models \phi] \mid s \to s'\})
\end{aligned}
$$

This semantics generalizes the 2-valued semantics for PML.

This 3-valued logic can be used to define a preorder on partial Kripke structures that reflects their degree of completeness. Let $\leq$ be the *information ordering* on truth values, in which $\perp \leq true$, $\perp \leq false$, $x \leq x$ (for all $x \in \{true, \perp, false\}$), and $x \not\leq y$ otherwise. The operators comp, min and max

preserve information: if $x \leq x'$ and $y \leq y'$, we have $\text{comp}(x) \leq \text{comp}(x')$, $\min(x, y) \leq \min(x', y')$, and $\max(x, y) \leq \max(x', y')$.

Definition 6. *Let $M_1 = (S_1, L_1, \mathcal{R}_1)$ and $M_2 = (S_2, L_2, \mathcal{R}_2)$ be partial Kripke structures. The* completeness preorder $\preceq$ *is the greatest relation $\mathcal{B} \subseteq S_1 \times S_2$ such that $(s_1, s_2) \in \mathcal{B}$ implies the following:*

- $\forall p \in P : L(s_1, p) \leq L(s_2, p)$,
- *if $s_1 \rightarrow s_1'$ then there is some $s_2' \in S_2$ such that $s_2 \rightarrow s_2'$ and $(s_1', s_2') \in \mathcal{B}$, and*
- *if $s_2 \rightarrow s_2'$ then there is some $s_1' \in S_1$ such that $s_1 \rightarrow s_1'$ and $(s_1', s_2') \in \mathcal{B}$.*

Intuitively, $s_1 \preceq s_2$ means that s_1 and s_2 are "nearly bisimilar" except that the atomic propositions in state s_1 may be less defined than in state s_2. Obviously, $s_1 \sim s_2$ implies $s_1 \preceq s_2$. Also, any partial Kripke structure can be completed to obtain a complete Kripke structure.

The following result shows that 3-valued PML logically characterizes the completeness preorder.

Theorem 2. *[BG99] Let $M_1 = (S_1, L_1, \mathcal{R}_1)$ and $M_2 = (S_2, L_2, \mathcal{R}_2)$ be partial Kripke structures such that $s_1 \in S_1$ and $s_2 \in S_2$, and let Φ be the set of all formulas of 3-valued PML. Then*

$$s_1 \preceq s_2 \text{ iff } (\forall \phi \in \Phi : [(M_1, s_1) \models \phi] \leq [(M_2, s_2) \models \phi]).$$

In other words, partial Kripke structures that are "more complete" with respect to $\preceq$ have more definite properties with respect to $\leq$, i.e., have more properties that are either *true* or *false*. Moreover, any formula ϕ of 3-valued PML that evaluates to *true* or *false* on a partial Kripke structure has the same truth value when evaluated on any more complete structure.

Results similar to those of this section were presented also for extended transition systems in [BG99]. Extended transition systems are labelled transition systems with a divergence predicate on states (e.g., see [Wal88,Sti87]). A connection made in [BG99] between 3-valued and 2-valued modal logics on extended transition systems partly inspired the following new results.

3.2 Positive PML

In the following sections we shall use a positive form of PML, which we refer to as PML$^+$. Here we define 2 and 3-valued semantics for PML$^+$ and observe that every formula of PML can be expressed in PML$^+$. The abstract syntax of PML$^+$ is as follows, where p ranges over P:

$$\phi ::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Box\, \phi \mid \Diamond\, \phi$$

Definition 7. *The value of a formula ϕ of 3-valued PML$^+$ in a state s of a partial Kripke structure $M = (S, L, \mathcal{R})$, written $[(M, s) \models^+ \phi]$, is defined inductively*

as follows:

$$\begin{aligned}
[(M,s) \models^+ p] &= L(s,p) \\
[(M,s) \models^+ \phi_1 \wedge \phi_2] &= min([(M,s) \models^+ \phi_1], [(M,s) \models^+ \phi_2]) \\
[(M,s) \models^+ \phi_1 \vee \phi_2] &= max([(M,s) \models^+ \phi_1], [(M,s) \models^+ \phi_2]) \\
[(M,s) \models^+ \Box\,\phi] &= min(\{[(M,s') \models^+ \phi] \mid s \rightarrow s'\}) \\
[(M,s) \models^+ \Diamond\,\phi] &= max(\{[(M,s') \models^+ \phi] \mid s \rightarrow s'\})
\end{aligned}$$

We define 2-valued PML$^+$ using this 3-valued interpretation as follows: a PML$^+$ formula ϕ holds at a state s of a complete Kripke structure M, written $(M,s) \models^+ \phi$, just if $[(M,s) \models^+ \phi] = \mathit{true}$.

We can translate every formula of PML to an equivalent formula of PML$^+$ if we consider only the partial Kripke structures $M = (S, L, \mathcal{R})$ in which, for every $p \in P$ there exists a $q \in P$ such that $L(s,p) = \mathrm{comp}(L(s,q))$ for all s in S. In such structures we refer to a proposition that is complementary to a proposition p as $\overline{p}$. We refer to such a partial Kripke structure as a *complement-closed* structure. Our translation T from PML to PML$^+$ is then as follows: $T(p) = p$, $T(\neg p) = \overline{p}$, $T(\neg(\phi_1 \wedge \phi_2)) = T(\neg\phi_1) \vee T(\neg\phi_2)$, $T(\neg(\Box\,\phi)) = \Diamond(T(\neg\phi))$, $T(\neg\neg\phi) = T(\phi)$, $T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2)$, and $T(\Box\,\phi) = \Box(T(\phi))$.

Proposition 1. *Let M be a partial Kripke structure that is complement-closed and ϕ be a PML formula. Then*

$$[(M,s) \models \phi] = [(M,s) \models^+ T(\phi)].$$

3.3 Model Checking 3-Valued Modal Logics

In this section we show that model checking 3-valued modal logic is no more expensive than model checking standard modal logic, and can be performed using existing model checkers.

From a 3-valued labelling L of a partial Kripke structure we can derive a pair of 2-valued labellings, one of which treats $\perp$ as *true*, while the other treats $\perp$ as *false*.

Definition 8. *Given a 3-valued labelling function L, we define the derived* optimistic *labelling function L_o and* pessimistic *labelling function L_p as follows:*

$$L_o(s,p) \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \mathit{if}\ L(s,p) = \perp \\ L(s,p) & \mathit{otherwise} \end{cases}$$

$$L_p(s,p) \stackrel{\text{def}}{=} \begin{cases} \mathit{false} & \mathit{if}\ L(s,p) = \perp \\ L(s,p) & \mathit{otherwise} \end{cases}$$

Given a partial Kripke structure $M = (S, L, \mathcal{R})$ we write $M_p = (S, L_p, \mathcal{R})$ for the derived pessimistic structure and $M_o = (S, L_o, \mathcal{R})$ for the derived optimistic structure.

The 3-valued interpretation of a PML$^+$ formula at a state s in a partial Kripke structure can be computed from the classical 2-valued interpretations of

the formula using the optimistic and pessimistic structures. The formula is *true* at s if it is *true* under the pessimistic interpretation, is *false* at s if it is *false* under the optimistic interpretation, and is $\perp$ otherwise.

Theorem 3. *Let $M = (S, L, \mathcal{R})$ be a partial Kripke structure with s in S, let M_p and M_o be the derived pessimistic and optimistic structures, and let ϕ be a formula of PML$^+$. Then*

$$[(M,s) \models^+ \phi] \stackrel{\text{def}}{=} \begin{cases} \textit{true} & \textit{if } (M_p, s) \models^+ \phi \\ \textit{false} & \textit{if } (M_o, s) \not\models^+ \phi \\ \perp & \textit{otherwise} \end{cases}$$

Thus, one can do 3-valued model checking by running a standard 2-valued model checker at most twice, once with the partial Kripke structure transformed to a complete, optimistic Kripke structure, and once with the partial Kripke structure transformed to a complete, pessimistic Kripke structure. These transformations are linear with respect to the size of the structure, so 3-valued model checking for PML has the same time and space complexity as the 2-valued case.

3.4 Adding Fixed-Point Operators

PML can be extended with a fixed-point operator to form a modal fixpoint logic, also referred to as the propositional μ-calculus [Koz83]. This very expressive logic includes as fragments linear-time temporal logic (LTL) [MP92] and computation-tree logic (CTL) [CE81]. In this section we extend PML$^+$ with fixed-point operators and show that model checking for this extended logic can also be reduced to standard 2-valued model checking.

PML$^+$ extended with fixed-point operators has the following abstract syntax, where p ranges over the set P of atomic propositions and X ranges over a set Var of fixed-point variables:

$$\phi ::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \Box\, \phi \mid \Diamond\, \phi \mid X \mid \nu X.\phi \mid \mu X.\phi$$

In fixed-point formulas $\nu X.\phi$ and $\mu X.\phi$ the operators ν and μ bind free occurrences of X in ϕ. We refer to this version of the modal mu-calculus [Koz83] as μL.

We now define a 2-valued semantics of μL. For the case of the fixed-point operator it makes sense to use a semantics that interprets a formula as the set of states for which the formula holds. We can derive such a set-valued semantics for PML$^+$ from the semantics given in Section 3.2 as follows:

$$\|M, \phi\| \stackrel{\text{def}}{=} \{s \in S \mid (M,s) \models^+ \phi\}$$

To interpret μL formulas we need not only a labelling to interpret atomic propositions but also a *valuation* to interpret fixed-point variables. A valuation $\mathcal{V}$ maps a fixed-point variable to a set $\mathcal{V}(X)$ of states. Thus μL formulas are interpreted relative to a Kripke structure M and a valuation $\mathcal{V}$. The new semantic

clauses for fixed-point variables and formulas are as follows: $\|M, X\|_{\mathcal{V}} \stackrel{\text{def}}{=} \mathcal{V}(X)$, $\|M, \nu X.\phi\|_{\mathcal{V}} \stackrel{\text{def}}{=} \nu f$, and $\|M, \mu X.\phi\|_{\mathcal{V}} \stackrel{\text{def}}{=} \mu f$, where νf (μf) denotes the greatest (least) fixed-point of function $f : S \to S$, defined as

$$f(A) \stackrel{\text{def}}{=} \|M, \phi\|_{\mathcal{V}[X:=A]}$$

Here $\mathcal{V}[X := A]$ stands for the valuation that is like $\mathcal{V}$ except that X is mapped to set A. Function f is a monotonic, set-valued function, so by the Knaster-Tarski theorem [Tar55] we know it has a greatest fixed point, namely $\bigcup\{A \subseteq S \mid A \subseteq f(A)\}$, where S is the set of states in Kripke structure M. Similarly, its least fixed point is $\bigcap\{A \subseteq S \mid f(A) \subseteq A\}$.

Now consider a 3-valued interpretation of μL. Here a formula ϕ is interpreted as a pair (S_1, S_2) of disjoint sets of states, where S_1 is the set for which ϕ is known to be true and S_2 is the set for which ϕ is known to be false. A valuation function $\mathcal{V}$ now maps a fixed-point variable to a pair of disjoint sets of states. Given a pair (S_1, S_2) of sets we write $\pi_1(S_1, S_2)$ for the first set and $\pi_2(S_1, S_2)$ for the second.

Definition 9. *The 3-valued interpretation $[M, \phi]_{\mathcal{V}}$ of a μL formula relative to a partial Kripke structure $M = (S, L, \mathcal{R})$ is defined as follows:*

$$\begin{aligned}
[M, p]_{\mathcal{V}} &\stackrel{\text{def}}{=} (\{s \in S \mid L(s, p) = \mathit{true}\}, \{s \in S \mid L(s, p) = \mathit{false}\}) \\
[M, \phi_1 \wedge \phi_2]_{\mathcal{V}} &\stackrel{\text{def}}{=} (\pi_1([M, \phi_1]_{\mathcal{V}}) \cap \pi_1([M, \phi_2]_{\mathcal{V}}), \pi_2([M, \phi_1]_{\mathcal{V}}) \cup \pi_2([M, \phi_2]_{\mathcal{V}})) \\
[M, \phi_1 \vee \phi_2]_{\mathcal{V}} &\stackrel{\text{def}}{=} (\pi_1([M, \phi_1]_{\mathcal{V}}) \cup \pi_1([M, \phi_2]_{\mathcal{V}}), \pi_2([M, \phi_1]_{\mathcal{V}}) \cap \pi_2([M, \phi_2]_{\mathcal{V}})) \\
[M, \Box\,\phi]_{\mathcal{V}} &\stackrel{\text{def}}{=} (\{s \mid \forall s'.s \to s' \Rightarrow s' \in \pi_1([M, \phi]_{\mathcal{V}})\}, \\
&\qquad \{s \mid \exists s'.s \to s' \wedge s' \in \pi_2([M, \phi]_{\mathcal{V}})\}) \\
[M, \Diamond\,\phi]_{\mathcal{V}} &\stackrel{\text{def}}{=} (\{s \mid \exists s'.s \to s' \wedge s' \in \pi_1([M, \phi]_{\mathcal{V}})\}, \\
&\qquad \{s \mid \forall s'.s \to s' \Rightarrow s' \in \pi_2([M, \phi]_{\mathcal{V}})\}) \\
[M, X]_{\mathcal{V}} &\stackrel{\text{def}}{=} \mathcal{V}(X) \\
[M, \nu X.\phi]_{\mathcal{V}} &\stackrel{\text{def}}{=} \nu f \\
[M, \mu X.\phi]_{\mathcal{V}} &\stackrel{\text{def}}{=} \mu f
\end{aligned}$$

In the fixed-point clauses $f(S_1, S_2) \stackrel{\text{def}}{=} [M, \phi]_{\mathcal{V}[X:=(S_1,S_2)]}$. We know f has greatest and least fixed-points by the Knaster-Tarski Theorem [Tar55] because pairs of sets ordered by

$$(S_1, S_2) \sqsubseteq (S_1', S_2') \stackrel{\text{def}}{=} S_1 \subseteq S_1' \text{ and } S_2 \supseteq S_2'$$

form a complete lattice with meet and join operators defined as $\bigvee\{(S_i, T_i) \mid i \in I\} \stackrel{\text{def}}{=} (\bigcup S_i, \bigcap T_i)$ and $\bigwedge\{(S_i, T_i) \mid i \in I\} \stackrel{\text{def}}{=} (\bigcap S_i, \bigcup T_i)$. The function $[M, \phi]_{\mathcal{V}}$ is order-preserving according to this order on pairs of sets.

For the PML$^+$ fragment of μL, this semantics is equivalent to the semantics given earlier for 3-valued PML$^+$, in the sense that $[(M, s) \models^+ \phi] = \mathit{true}$ just

if s is in $\pi_1([M,\phi]_{\mathcal{V}})$, $[(M,s) \models^+ \phi] = \mathit{false}$ just if s is in $\pi_2([M,\phi]_{\mathcal{V}})$, and $[(M,s) \models^+ \phi] = \bot$ just if s is in neither $\pi_1([M,\phi]_{\mathcal{V}})$ nor $\pi_2([M,\phi]_{\mathcal{V}})$.

Now we show a result for μL analogous to that of the previous section for PML$^+$. Given a valuation $\mathcal{V}$ over a partial Kripke structure with state set S, we write $\mathcal{V}_p$ for the valuation that maps X to $\pi_1(\mathcal{V}(X))$, and $\mathcal{V}_o$ for the valuation that maps X to $S - \pi_2(\mathcal{V}(X))$.

Theorem 4. *Let $M = (S, L, \mathcal{R})$ be a partial Kripke structure, $\mathcal{V}$ be a valuation, M_p and M_o be the derived pessimistic and optimistic structures of M, and ϕ be a formula of μL. Then*

$$[M,\phi]_{\mathcal{V}} = (\|M_p,\phi\|_{\mathcal{V}_p}, S - \|M_o,\phi\|_{\mathcal{V}_o}).$$

4 The Generalized Model-Checking Problem

4.1 Problem Statement

We have said that our three-valued semantics gives a value of $\bot$ for some formula and some state in a partial Kripke structure just if the partial Kripke structure does not contain enough information to give answer *true* or *false*. However, it could be argued that our semantics returns $\bot$ more often than it should. Consider a partial Kripke structure M consisting of a single state s such that $s \rightarrow s$ and the value of proposition p at s is $\bot$. If we interpret formula $p \vee \neg p$ at s we get $\bot$, although in all complete Kripke structures more complete than M the formula is interpreted as *true*.

This problem is not confined to formulas that are tautological or unsatisfiable. Consider the partial Kripke structure like M above but for which the value of q at s is *true*. The formula $q \wedge (p \vee \neg p)$, which is neither a tautology nor unsatisfiable, is $\bot$ at s, yet again in all complete structures the formula is *true*.

Thus, our three-valued semantics does not have the desirable property that the value of a formula ϕ at a state is $\bot$ just if there exists an s' such that $s \preceq s'$ and the value of ϕ at s' is *true* and there also exists an s'' such that $s \preceq s''$ and the value of ϕ at s'' is *false*. However, we can use this property to define an alternative three-valued semantics for modal logics. We call this the *thorough* semantics because it does does more than our other semantics to discover whether enough information is present in a partial Kripke structure to give a definite answer. Let the *completions* $\mathcal{C}(M,s)$ of a state s of a partial Kripke structure M be the set of all states s' of complete Kripke structures M' such that $s \preceq s'$.

Definition 10 (Thorough three-valued semantics). *Let ϕ be a formula of any two-valued logic for which a satisfaction relation $\models$ is defined on complete Kripke structures. The truth value of ϕ in a state s of a partial Kripke structure M under the* thorough *interpretation, written $[(M,s) \models \phi]_t$, is defined as follows:*

$$[(M,s) \models \phi]_t = \begin{cases} \mathit{true} \text{ if } (M',s') \models \phi \text{ for all } (M',s') \text{ in } \mathcal{C}(M,s) \\ \mathit{false} \text{ if } (M',s') \not\models \phi \text{ for all } (M',s') \text{ in } \mathcal{C}(M,s) \\ \bot \quad \mathit{otherwise} \end{cases}$$

In this section we focus on the following related problem.

Definition 11 (Generalized Model-Checking Problem). *Given a state s of a partial Kripke structure M and a formula ϕ of a (two-valued) temporal logic L, does there exist a state s' of a complete Kripke structure M' such that $s \preceq s'$ and $(M', s') \models \phi$?*

We call this problem the *generalized model-checking problem*. It should be clear that interpreting a formula according to the thorough three-valued semantics is equivalent to solving two instances of the generalized model checking problem. We have phrased the problem this way to emphasize its tie to the satisfiability problem.

The generalized model-checking problem generalizes both model checking and satisfiability checking. At one extreme, where M is $(\{s_0\}, L, \{(s_0, s_0)\})$ with $L(s_0, p) = \perp$ for all $p \in P$, all Kripke structures are more complete than M and the problem reduces to the satisfiability problem for the corresponding logic. At the other extreme, where M is complete, only a single structure needs to be checked and the problem reduces to model checking. Therefore, the worst-case complexity for the generalized model-checking problem will never be better than the worst-case complexities for the model-checking and satisfiability problems for the corresponding logic. The following theorem formally states that the generalized model-checking problem is at least as hard as the satisfiability problem.

Theorem 5. *Let L denote the modal mu-calculus or any of its fragments (propositional logic, propositional modal logic, LTL, CTL, CTL*, etc.). Then the satisfiability problem for L is reducible (in linear-time and logarithmic space) to the generalized model-checking problem for L.*

In the following sections, we present algorithms and complexity bounds for the generalized model-checking problem for various temporal logics. Our algorithms are based on automata-theoretic techniques (e.g., see [BVW94]). For basic notions of automata theory (including definitions of nondeterministic and alternating Büchi automata on words and trees), please refer to [Var97a].

4.2 Branching-Time Temporal Logics

We consider first the case of computation tree logic (CTL) [CES86]. The next theorem presents a decision procedure for the generalized model-checking problem for CTL.

Theorem 6. *Given a state s_0 of partial Kripke structure $M = (S, L, \mathcal{R})$ and a CTL formula ϕ, one can construct an alternating Büchi word automaton $A_{(M,s_0),\phi}$ over a 1-letter alphabet with at most $O(|S| \cdot 2^{O(|\phi|)})$ states such that*

$$(\exists (M', s'_0) : s_0 \preceq s'_0 \text{ and } (M', s'_0) \models \phi) \text{ iff } \mathcal{L}(A_{(M,s_0),\phi}) \neq \emptyset.$$

Proof. (Sketch) $A_{(M,s_0),\phi}$ is constructed from the partial Kripke structure M and a nondeterministic Büchi tree automaton A_ϕ that accepts exactly the infinite trees satisfying the formula ϕ, in such a way that $A_{(M,s_0),\phi}$ accepts exactly the computation trees of complete Kripke structures that satisfy the property ϕ and that are more complete than (M, s_0).

A corollary of the above construction is that, if a state s'_0 of a complete Kripke structure as defined in Theorem 6 exists, there also exists a state of a complete Kripke structure M' satisfying the property ϕ such that M' contains at most $|S| \cdot 2^{O(|\phi|)}$ states.

Since the emptiness problem for alternating Büchi word automata over a 1-letter alphabet can be reduced in linear time and logarithmic space to the emptiness problem for nondeterministic Büchi tree automata [BVW94], which is itself decidable in quadratic time [VW86], we obtain the following.

Theorem 7. *The generalized model-checking problem for a state s_0 of a partial Kripke structure $M = (S, L, \mathcal{R})$ and a CTL formula ϕ can be decided in time $O(|S|^2 \cdot 2^{O(|\phi|)})$.*

Note that, in the extreme case where M is complete, the upper bound given by the previous algorithm is not optimal since a traditional CTL model-checking algorithm [CES86] can decide whether $(M, s_0) \models \phi$ in time $O(|S| \cdot |\phi|)$.

In the general case, however, we can prove that the time complexity of the previous algorithm in the size of the formula is essentially optimal.

Theorem 8. *The generalized model-checking problem for CTL is EXPTIME-complete.*

Let us now discuss briefly the case of PML. Since PML is included in CTL, the above algorithm can also be used to solve the generalized model-checking problem for PML. However, the problem can now be solved using polynomial space.

Theorem 9. *The generalized model-checking problem for PML is PSPACE-complete.*

If we restrict the logic one step further and reduce it to propositional logic, it is easy to prove that the generalized model-checking problem has again the same complexity as the satisfiability problem.

Theorem 10. *The generalized model-checking problem for propositional logic is NP-complete.*

Let us now consider the case of branching-time logics more expressive than CTL, such as CTL* and the modal mu-calculus. Since formulas in these logics also have translations to nondeterministic Büchi tree automata (e.g., see [BVW94]), the general algorithm presented in the proof of Theorem 6 also provides a decision procedure for the generalized model-checking problem for these logics, with a quadratic time complexity in the size of the partial Kripke structure. As far as the complexity in the formula is concerned, we can prove the following.

Theorem 11. *For any branching-time temporal logic L containing CTL, the generalized model-checking problem for L is polynomial-time reducible to the satisfiability problem for L.*

Putting it all together, we obtain the following theorem which summarizes all the results presented in this section concerning the complexity of the generalized model-checking problem for branching-time temporal logics and a fixed partial Kripke structure.

Theorem 12. *Let L denote propositional logic, propositional modal logic, CTL, or any branching-time logic including CTL (such as CTL* or the modal mu-calculus). The generalized model-checking problem for the logic L has the same complexity as the satisfiability problem for L.*

4.3 Linear-Time Temporal Logics

Let us now turn to linear-time temporal logic (LTL) [MP92]. In this case, we can again reduce the generalized model-checking problem to checking emptiness of an alternating Büchi word automaton over a 1-letter alphabet.

Theorem 13. *Given a state s_0 of partial Kripke structure $M = (S, L, \mathcal{R})$ and an LTL formula ϕ, one can construct an alternating Büchi word automaton $A_{(M,s_0),\phi}$ over a 1-letter alphabet with at most $O(|S| \cdot 2^{|\phi|})$ states such that*

$$(\exists (M', s_0') : s_0 \preceq s_0' \text{ and } (M', s_0') \models \phi) \text{ iff } \mathcal{L}(A_{(M,s_0),\phi}) \neq \emptyset.$$

Since the proof of the above theorem is based only on the fact the property ϕ of interest can be represented by a nondeterministic Büchi word automaton, it also holds for properties directly represented by such automata (i.e., ω-regular languages) or formulas of logics which can be translated into such automata, like extended temporal logic [Wol83] and the linear-time fragment of the mu-calculus [SW90] for instance. We then obtain the following result.

Theorem 14. *The generalized model-checking problem for a state s_0 of a partial Kripke structure $M = (S, L, R)$ and an LTL formula ϕ can be decided in time $O(|S|^2 \cdot 2^{2|\phi|})$.*

So far, the above results for LTL are very similar to those of the previous section on the branching-time case. However, a major difference is that the generalized model-checking problem for LTL is harder than the satisfiability problem and the model-checking problem for LTL, which are both known to be PSPACE-complete [Eme90].

Theorem 15. *The generalized model-checking problem for linear-time temporal logic is EXPTIME-complete.*

In summary, in contrast with the results obtained for branching-time in the previous section, the generalized model checking problem is harder than the satisfiability problem in the LTL case. This is due to the need for alternating/tree

automata to solve the problem. Other problems of that flavor include the *realizability* [ALW89] and *synthesis* [PR89a,PR89b] problems for linear-time temporal logic specifications.

At first sight, one could think that the need for tree automata could come from the mismatch between LTL and the completeness preorder $\preceq$. Indeed, the completeness preorder is not logically characterized by the 3-valued extension of LTL that can be obtained following the work of [BG99]. To see this, notice that the completeness preorder reduces to a bisimulation relation in the case of complete Kripke structures [BG99]. It is well-known that, while Kripke structures that are bisimilar satisfy the same LTL formulas, Kripke structures that satisfy the same LTL formulas are not necessarily bisimilar. The completeness preorder is thus stronger than necessary for reasoning only about the linear behaviors of partial Kripke structures. However, replacing the completeness preorder $\preceq$ by the weaker "linear" preorder induced by 3-valued LTL in the definition of the generalized model-checking problem does not make this problem easier: a constructive solution of the modified problem still requires the construction of an alternating automaton $A_{(M,s_0),\phi}$ as in the proof of Theorem 13.

5 Related Work

Most of the existing work on 3-valued modal logic focuses on its proof theory. For example, see [Seg67], [Mor89], and [Fit92a]. Our work in Section 3 is closest to [Fit92b]. Here Fitting presents two interpretations of modal logic: one a many-valued version and the other based on obtaining 2-valued interpretations from each of a set of experts. Fitting shows that such a multi-expert interpretation corresponds in a precise way to a multi-valued interpretation, similarly to how we show that a 3-valued interpretation can be obtained by separate optimistic and pessimistic interpretations. However, in Fitting's case the multi-expert interpretation is not obtained by separate, 2-valued interpretations of each expert. Also, Fitting does not define a completeness preorder over his models, or characterization results.

In [SRW99] a 3-valued logic is used for program analysis. The state of program store is represented as a 3-valued structure of first-order logic. The possible values of program store are conservatively computed by an abstract interpretation of the program on such a structure. The main technical result is an embedding theorem showing that, for a certain class of abstraction functions on the domain of such structures, the interpretation of a first-order formula on the abstract structure is less definite than its interpretation on the structure itself.

A semantics like our thorough semantics could be defined for other preorders on processes, such as refinement preorder of modal process logic [LT88] or the divergence preorder [Wal88]. In [BG99] we defined a 3-valued modal logic that characterizes the divergence preorder, but did not define a thorough semantics based on it. In [ALW89] an implementation preorder is defined on process specifications consisting of a finite labelled transition system and a nondeterministic Buchi word automaton. A process specification P is said to be *realizable*

in [ALW89] if a P' lower in the implementation preorder exists for which the infinite behavior of the transition system of P' is contained in the language of the Buchi automaton of P'. The realizability problem and the generalized model checking problem for LTL clearly differ, but their relationship deserves further study.

References

ALW89. Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, July 1989. 180, 181

BG99. Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In N. Halbwachs and D. Peled, editors, *Proceedings of CAV '99, LNCS 1633*, pages 274–287, 1999. 168, 169, 171, 172, 180

BVW94. Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, California, June 1994. Springer-Verlag. 177, 178

CE81. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981. 174

CES86. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986. 177, 178

Eme90. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier/MIT Press, Amsterdam/Cambridge, 1990. 179

Fit92a. Melvin Fitting. Many-valued modal logics I. *Fundamenta Informaticae*, 15:235–254, 1992. 180

Fit92b. Melvin Fitting. Many-valued modal logics II. *Fundamenta Informaticae*, 17:55–73, 1992. 180

HM85. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985. 170

Kle87. Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1987. 171

Koz83. D. Kozen. Results on the Propositional Mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983. 174

LT88. Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988. 180

Mil89. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. 170

Mor89. Osamu Morikawa. Some modal logics based on a three-valued logic. *Notre Dame Journal of Formal Logic*, 30(1):130–137, 1989. 180

MP92. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992. 174, 179

Par81. D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981. 170

PR89a. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of the Sixteenth Symposim on Principles of Programming Languages*, Austin, January 1989. 180

PR89b. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of ICALP'89*, Stresa, July 1989. 180

Seg67. Krister Segerberg. Some modal logics based on a three-valued logic. *Theoria*, 33:53–71, 1967. 180

SRW99. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 1999. 180

Sti87. Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:331–347, 1987. 172

SW90. C. Stirling and D. Walker. CCS, liveness and local model checking in the linear-time mu-calculus. In *Proc. First International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 166–178. Springer-Verlag, 1990. 179

Tar55. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. of Maths*, 5:285–309, 1955. 175

Var97a. M.Y. Vardi. Alternating automata: Checking truth and validity for temporal logics. In *Proceedings of CADE'97*, 1997. 177

Var97b. M.Y. Vardi. Why is modal logic so robustly decidable? In *Proceedings of DIMACS Workshop on Descriptive Complexity and Finite Models*. AMS, 1997. 170

VW86. M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, April 1986. 178

Wal88. D. J. Walker. Bisimulations and divergence. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1988. 172, 180

Wol83. Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983. 179

Reachability Analysis for Some Models of Infinite-State Transition Systems

Oscar H. Ibarra*, Tevfik Bultan**, and Jianwen Su*

Department of Computer Science, University of California
Santa Barbara, CA 93106, USA

Abstract. We introduce some new models of infinite-state transition systems. The basic model, called a (reversal-bounded) counter machine (CM), is a nondeterministic finite automaton augmented with finitely many reversal-bounded counters (i.e. each counter can be incremented or decremented by 1 and tested for zero, but the number of times it can change mode from nondecreasing to nonincreasing and vice-versa is bounded by a constant, independent of the computation). We extend a CM by augmenting it with some familiar data structures: (*i*) A pushdown counter machine (PCM) is a CM augmented with an unrestricted pushdown stack. (*ii*) A tape counter machine (TCM) is a CM augmented with a two-way read/write worktape that is restricted in that the number of times the head crosses the boundary between any two adjacent cells of the worktape is bounded by a constant, independent of the computation (thus, the worktape is finite-crossing). There is no bound on how long the head can remain on a cell. (*iii*) A queue counter machine (QCM) is a CM augmented with a queue that is restricted in that the number of alternations between non-deletion phase and non-insertion phase is bounded by a constant. A non-deletion (non-insertion) phase is a period consisting of insertions (deletions) and no-changes, i.e., the queue is idle. We show that emptiness, (binary, forward, and backward) reachability, nonsafety, and invariance for these machines are solvable. We also look at extensions of the models that allow the use of linear-relation tests among the counters and parameterized constants as "primitive" predicates. We investigate the conditions under which these problems are still solvable.

1 Introduction

Since the introduction of efficient automated verification techniques such as symbolic model-checking [BCM92], finite-state machines have been widely used for modeling reactive systems. However, due to their limited expressiveness, finite-state models are not suitable for specifying most infinite-state systems. To overcome this limitation researchers have used 1) abstraction techniques to generate finite state abstractions of infinite-state systems [DF95,DGG97,BLO98], 2) semi-decision procedures which prove or disprove a property if they converge,

* Supported in part by NSF grants IRI-9700370 and IIS-9817432.
** Supported in part by NSF grant CCR-9970976.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 183–198, 2000.

but are not guaranteed to converge [BG96,WB98], or 3) conservative approximation techniques which are guaranteed to converge but may not always return a definite answer [HRP94,BGP99]. Another promising approach for verification of infinite-state systems is finding/identifying models which can represent such systems and have decidable verification queries such as reachability, invariance, etc. It is well-known that, in general, verification problems for infinite-state systems are undecidable [Esp97]. In fact, even for systems with only two variables (or counters) that can be incremented or decremented by 1 and tested for 0, we already know that the halting problem is undecidable (hence, the emptiness, reachability, and other problems are also undecidable) [Min61]. However, certain restrictions can be placed on the workings of these systems that make them amenable to analysis. Some models that have been shown to have decidable properties are: various approximations on multicounter machines [CJ98,BW94,FS00], timed automata [AD94] (and real-time logics [AH94,ACD93,HNSY94]), pushdown automata [BEM97,FWW97,Wal96]. [BER95] studies models of hybrid systems of finite automata supplied with (unbounded) discrete data structures and continuous variables and obtains decidability results for several classes of systems with control variables and observation variables. [DIBKS00] investigates discrete timed automata (i.e., timed automata with integer-valued clocks) augmented with a pushdown stack.

In this paper, we introduce some new models of infinite-state systems. The basic model, called a (reversal-bounded) counter machine (CM), is a nondeterministic finite automaton augmented with finitely many reversal-bounded counters (i.e., each counter can be incremented or decremented by 1 and tested for zero, but the number of times it can change mode from nondecreasing to nonincreasing and vice-versa is bounded by a constant, independent of the computation). This model was first introduced and studied in [Iba78]. We extend a CM by augmenting it with one of the following data structures: i) an unrestricted pushdown stack; ii) a finite-crossing read/write worktape; ii) a restricted queue. We show that emptiness, (binary, forward, and backward) reachability, nonsafety, and invariance for these machines are solvable. We also look at extensions of the models that allow the use of linear-relation tests among the counters and parameterized constants as "primitive" predicates. We investigate the conditions under which these problems are still solvable.

Our results can be used in automated verification of infinite-state systems in following ways: (1) By reducing a given infinite-state system to one of the models we present in this paper, one can prove that certain verification queries for the given system are decidable and can be verified without any abstractions or approximations (for example, this approach was used in [DIBKS00] to show the decidability of binary reachability problem for discrete pushdown timed automata by reducing it to the emptiness problem for pushdown counter machines); (2) By restricting the behaviors of a given infinite-state system using properties such as reversal-boundedness one can obtain a conservative approximation of the given system (in the sense that when an error is found in the restricted system this implies that the error exists in the original system); and (3) The proofs of

decidability of properties such as emptiness and reachability can be used as a basis for algorithms for verification of such properties.

The paper is organized as follows. Section 2 defines the new models and summarizes the main results. Section 3 gives proof sketches. Section 4 looks at some generalizations of the models. Section 5 is a brief conclusion.

2 The Models and Main Results

Many verification problems for systems that can be modeled by automata (finite- or infinite-state) can often be reduced to the emptiness problem: Given a machine M, does it accept at least one input? Decidability (existence of an algorithm) of emptiness can lead to decidability of questions such as reachability, nonsafety, invariance, containment, and equivalence, which are at the heart of verification procedures. While these problems are decidable for finite-state systems, they are, in general, undecidable for infinite-state systems. However, with appropriate restrictions, some infinite-state models have been shown to have decidable properties.

In our discussion of computational models below, we consider two types of machines: one with a one-way read-only input tape and the other without an input tape. When we are interested in "language recognition/acceptance", we consider machines with input. In verification problems (reachability, safety, etc.), we use the machines mostly as systems specifications rather than language recognizers, since the interest is more on the "behaviors" they generate; so these machines have no input. However, machines with input tape are also of interest for "parametric" systems, where the parameters can be specified on the input tape (we discuss this in Section 4). Also, many of the proofs of the results concerning verification problems on machines without an input tape are reductions to the decidability of the emptiness for machines with an input tape. It will be clear from the context, which type of machine we are dealing with.

The basic model is a nondeterministic finite-state machine augmented with k "reversal-bounded" counters (for some k). Thus, each counter can be incremented or decremented by 1 and tested for zero and is reversal-bounded in that the number of alternations between nondecreasing mode and nonincreasing mode is bounded by a constant, independent of the computation. Without loss of generality, we assume that the counters can only store nonnegative integers, since the finite-state control can remember the signs of the numbers. We call this basic model a CM. Note that, by convention, a CM has no input tape. If we attach a one-way read-only input tape, we call it a CM acceptor. We can generalize the CM by augmenting it with some well-known data structures:

1. A pushdown counter machine (PCM) is a CM augmented with an unrestricted pushdown stack.
2. A tape counter machine (TCM) is a CM augmented with a two-way read/write worktape that is restricted in that the number of times the head crosses the boundary between any two adjacent cells of the worktape is bounded by a constant, independent of the computation (i.e. the worktape

is finite-crossing). There is no bound on how long the head can remain ("sit") on a cell.

3. A queue counter machine (QCM) is a CM augmented with a queue that is restricted in that the number of alternations between non-deletion phase and non-insertion phase is bounded by a constant. A non-deletion (non-insertion) phase is a period consisting of insertions (deletions) and no-changes, i.e., the queue is idle.

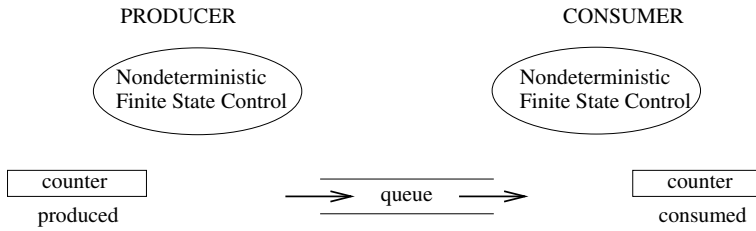


Fig. 1. A producer-consumer system with an unbounded buffer

Example 1: We give an example of a system that can be modeled by a QCM. Consider the producer-consumer system given in Fig. 1. The finite state control of the producer has a *produce* state, and the finite state control of the consumer has a *consume* state. When the producer is in *produce* state it can take the *write* transition which increments the counter *produced* by one and writes an item from a queue alphabet $\{a, b, c, d\}$ to the FIFO *queue*. When the consumer is in *consume* state and the queue is not empty, it can take the *read* transition which increments the counter *consumed* by one, removes an item from the FIFO *queue*, and stores the symbol read from the FIFO *queue* in its finite state control. Note that counters *produced* and *consumed* are reversal bounded since they are nondecreasing. If we limit the behavior of the *queue*, so that the alternations between non-write and non-read phases are bounded by a constant, this system can be represented by a QCM. Note that this limitation does not bound the size of the *queue*, or counters. We can effectively verify properties such as invariance for this restricted system (see Example 2). If we find out that the restricted system does not satisfy an invariant, this implies that the unrestricted system does not satisfy it either.

Clearly, QCMs can effectively be simulated by TCMs. Thus, decidability results for TCMs apply to QCMs. PCMs and TCMs are more powerful than CMs. A PCM is incomparable with a TCM (note that the pushdown in a PCM is unrestricted, i.e., there is no restriction on the number of alternations between non-popping and non-pushing). The restriction that the worktape of a TCM is finite-crossing is necessary; otherwise, the tape would be equivalent to a Turing machine (TM). Similarly, the restriction on the QCM is necessary; otherwise, the queue would be equivalent to a TM. It is also easy to see that a QCM with two restricted queues can simulate a TM.

Convention: By convention, throughout, CM, PCM, ... will refer to a machine *without* an input tape. When they have an input tape, they will be called CM acceptor, PCM acceptor, ... In this case the computation starts with all counters zero and the pushdown stack (worktape/queue) empty (blank). Unless otherwise specified, all machines are nondeterministic.

Define a configuration of a PCM to be a string of the form:

$$0^q\#0^{i_1}\#0^{i_2}\#...\#0^{i_k}\#w$$

where q in $\{1, 2, ..., n\}$ represents the state, 0^{i_j} represents the value of counter j (in unary), and w represents the contents of the stack, with the rightmost symbol of w the top of the stack. For for certain problems, we can have w given in "reversed".

For a TCM, since the worktape is two-way read/write, the tape contents w in the configuration has to indicate the position of the read/write head within the tape. For a QCM, the left (right) end w corresponds the *rear* (*front*) of the queue. For a CM, the configuration has no tape and is of the form $0^q\#0^{i_1}\#0^{i_2}\#...\#0^{i_k}$.

TCM (PCM, QCM) acceptors are quite powerful. For example, a TCM acceptor (even a much restricted version) M can accept the language over the alphabet $\{a, b, c, d\}$ consisting of all strings $x\#x$ such that x has the property that the sum of the lengths of all runs of c's occurring between pairs of symbols a and b (in this order) equals the number of d's. For example, $x = dacbacaccbdd$ satisfies the property, but $x = ddacbacaccbdd$ does not. M has one counter and operates in the following manner. Given input $x\#y$, M copies x on the worktape and checks that $x = y$ and resets the worktape head to the left end of x. It computes the *sum* in its counter by looking at the worktape and whenever it sees an a, it first checks that there is a matching b to the right and that all symbols in-between are c's. It then moves left (to a), adding the length of the run of c's to the counter. The process is repeated until the whole string has been examined. (So far, M crosses any boundary between two adjacent cells on the worktape at most 7 times.) M then resets the worktape head to the left end of the tape and checks that the number of d's is equal to the *sum* in the counter. Thus, M is 9-crossing, although worktape head makes an unbounded number of (left-to-right and right-to-left) turns, i.e., it is not finite-turn. Note also that M does not re-write the worktape and is deterministic.

Even a CM is quite powerful. For example, let L be the language consisting of all strings $x\#y\#z$, such that x, y, z are pair-wise distinct binary strings. A CM acceptor M with 3-counters, each making exactly one reversal, can accept L. M uses one counter to check that x is different from y, a second counter to compare x and z, and a third counter to check that y is different from z. To verify that x is different from y, M "guesses" a position of discrepancy (within the string x). It does this by incrementing the first counter by 1 for every symbol it encounters while moving right on x, and nondeterministically terminating the counting at some point, guessing that a position of discrepancy has been reached. M records in it's finite-control the symbol in that position. M uses the value in the counter to arrive at the same location within y where a discrepancy was

guessed to occur. The second and third counters are used in a similar way to compare x with z and y with z.

Decidability/complexity results concerning CMs have been obtained in [Iba78,GI81]. These results were used recently to prove the decidability (and derive the complexity) of some decision problems (containment, equivalence, disjointness, etc.) for database queries with linear constraints [IS99,ISB00].

CMs, PCMs, QCMs, and TCMs (these have no inputs!) can generate rather complex behaviors. For example, one can show that a PCM with one counter can start in its initial state with empty pushdown stack and reach a configuration where the pushdown contains a string x with the property described in the first example above.

The main results of the paper are:

1. (*Emptiness*) The emptiness problem for a class of machine acceptors is the problem of deciding, given a machine M in the class, whether the language $L(M)$ accepted by M is empty. We can show that the emptiness problems for PCM and TCM acceptors are decidable.
2. (*Binary-Reachability*) If M is a PCM (TCM), define its binary reachability set, $R(M)$, to be set of all pairs (α, β) of configurations such that α can reach β in 0 or more transitions. (Note that a pair of configurations can be represented as a string. For a PCM, the stack word component of β is written in reversed.) We can effectively construct, given a PCM (TCM) M, a PCM(TCM) acceptor accepting $R(M)$.
3. (*Forward-Reachability*) Let M be a PCM (TCM) and S be a set of configurations accepted by a CM acceptor. We can effectively construct a PCM (TCM) acceptor accepting $F_M(S)$ = the set of all configurations of M that are reachable from configurations in S in 0 or more transitions.
4. (*Backward-Reachability*) Let M be a PCM (TCM) and S be a set of configurations accepted by a CM acceptor. We can effectively construct a PCM (TCM) acceptor accepting $B_M(S)$ = the set of all configurations of M that can reach configurations in S in 0 or more transitions.
5. (*Nonsafety*) We can effectively construct, given a PCM (TCM) M and two sets of configurations I (*initial set*) and B (*bad set*) accepted by CM acceptors, a PCM (TMC) M' that accepts a configuration α if and only if (i) α is in I, and (ii) M when started in α can reach a configuration in B. Thus nonsafety is decidable.
6. (*Invariance*) We can effectively construct, given a PCM (TCM) M and two sets of configurations I (*initial set*) and G (*good set*) accepted by CM acceptor and deterministic CM acceptor respectively, a PCM (TMC) M' that accepts a configuration α if and only if (i) α is in I, and (ii) M when started in α can reach a configuration not in G. Thus invariance is decidable.

Obviously, the above results hold for CMs and, as previously observed, QCMs can effectively be simulated by TCMs; so the results hold for these machines as well. In contrast, emptiness is undecidable for PCMs and TCMs with multiple pushdown stacks (finite-crossing worktapes). In fact, it follows from the undecidability of the Post Correspondence Problem [Pos46] that emptiness is

undecidable for machines with only two pushdown stacks, even if the stacks are restricted to making only one alternation from non-popping to non-pushing.

Example 2: Consider again the producer-consumer system given in Fig. 1 and the QCM M that is constructed from it by restricting the behavior of the *queue*. An invariance property for this system can be defined as follows: I is defined as the set of configurations where both *produced* and *consumed* are 0 and the *queue* is empty, and G is defined as the set of configurations where *produced* − *consumed* is equal to the number of items in the *queue*, and the number of a's in the queue is $\leqslant$ the number of b's and the number of c's is $\geqslant$ the number of d's. We want to verify that for all the configurations in I (which represent the initial configurations of the system) the set of reachable configurations is contained in G, i.e., G is an invariant. We can construct deterministic CM acceptors for both these sets; hence, we can construct a QCM M' which will recognize all the configurations in I which can reach a configuration that is not in G. If the language accepted by M' is not empty, this means that M (and the corresponding unrestricted system) has an error. If the language accepted by M' is empty, however, this only proves that the restricted system is correct. It does not prove the correctness of the unrestricted system. Hence, the restricted system is a conservative approximation, in the sense that there are no false negatives but there could be false positives. If a given input system can be reduced to a QCM both negative and positive results would be exact.

Example 3: Consider a CM M, and suppose we are interested in the set T of pairs of configurations (α, β) of M such that there is a computation path (i.e., sequence of configurations) from α to β that satisfies a property that can be verified by a PCM (QCM, TCM) acceptor. Then T is computable and can be accepted by a PCM (QCM, TCM) acceptor. For example, suppose that the property is for the path to contain two non-overlapping subpaths (i.e., segments of computation) which go through the same sequence of states, and the length of the subpath is no less than a third of the length of the entire path. Clearly, T can accepted by a QCM acceptor or by a TCM acceptor.

We also study extensions of PCM and TCM acceptors in Section 4. In particular, we look at acceptors that allow as "primitive" predicates the use of linear-relation tests among the counters and parameterized constants, e.g., tests like "Is $3x - 5y + 11z - 2D_1 + 9D_2 < 12$?", where x, y, z are counters and D_1 and D_2 represent parameterized constants whose domain is the set of all integers $(+, -, 0)$. Note that directly implementing such tests using the standard "testing for zero" would result in a machine that is not reversal-bounded. We investigate the conditions under which the emptiness problem and other problems are decidable for these extended models.

3 Proof Sketches

We now give proof sketches of the results. We start with the emptiness problem.

Theorem 1. *The emptiness problem for PCM acceptors is decidable.*

Proof. This result was already shown in [Iba78]. For completeness, we describe the idea of the proof. Let A be an alphabet consisting of k symbols $a_1, ..., a_k$, $\mathbb{N}$ the set of nonnegative integers. For each string (word) w in A^*, we define

$$f(w) = (i_1, ..., i_k), \quad \text{where } i_j \text{ is the number of occurrences of } a_j \text{ in } w.$$

If L is a subset of A^*, we define $f(L) = \{f(w) \mid w \in L\}$.

In [Iba78], it was shown that if M is a PCM acceptor with input alphabet A, then $f(L(M))$ is an effectively computable semilinear set (or, equivalently, definable by a Presburger formula), where $L(M)$ is the language accepted by M. Hence, $L(M)$ is empty iff $f(L(M))$ is empty, which is decidable since it is Presburger. □

Obviously, the above theorem holds for machines with no pushdown stack:

Corollary 1. *The emptiness problem for CM acceptors is decidable.*

It has been shown in [GI81] that the emptiness problem for CM acceptors is decidable in n^{ckr} time for some constant c, where n is the size of the machine, k is the number of counters, and r is the reversal-bound on each counter. We believe that a similar bound could be obtained for the case of PCM acceptors.

To prove the decidability of the emptiness problem for TCM acceptors, we need some lemmas.

Lemma 1. *Let M be a TCM acceptor. We can effectively construct a TCM acceptor M' such that $L(M) = L(M')$ and M' is non-sitting in that in any computation, its read/write head does not sit on any tape cell (i.e., it always moves left or right of a cell in every step).*

Proof. Note that M' cannot just simulate a sitting step by a left (or right) move followed by a right (or left) move. This is because the read/write head can sit on a cell an unbounded number of steps, and this would make M' not finite-crossing.

What M' can do is to use a new "dummy" symbol, say #. M' begins the simulation of M by writing a finite-length sequence of #'s on the worktape, the length being chosen nondeterministically. M' simulates M, but whenever M writes a symbol on a *new* tape cell, M' also writes to the right of this cell a finite-length sequence of #'s (again the length is chosen nondeterministically). Thus, at any time, the worktape contains a string where every pair of non-dummy symbols (i.e. symbols in the worktape alphabet of M) is separated by a string of #'s. During the simulation, M' uses/moves on the #'s to simulate the sitting moves of M (this is possible if there are enough #'s between any pair of non-dummy symbols). To simulate a nonsitting move of M, M' may need to "skip over" the #'s to get to the correct non-dummy symbol. Clearly, M' is non-sitting and accepts $L(M)$. □

Lemma 2. *Let M be a TCM acceptor. We can effectively construct a TCM M' (with no input tape!) such that $L(M)$ is nonempty if and only if M' when started with a blank worktape and zero counters has a halting sequence of moves. (Note that since the machine is nondeterministic, not all sequences of moves may halt.)*

Proof. The construction of M' is straightforward. In the simulation of M, M' nondeterministically guesses the symbols comprising the input tape. □

Theorem 2. *The emptiness problem for TCM acceptors is decidable.*

Proof. From Lemma 2, we need only show that the halting problem for TCMs is decidable. Let M be a TCM. By Lemma 1, we assume that M is non-sitting. We may also assume, without loss of generality, that each counter of M makes exactly one reversal (since a counter making k reversals can be simulated by $(k+1)/2$ counters, each making exactly one reversal), halts with all the counters zero and the worktape head at the right end of the tape, and that in any computation, every counter becomes positive.

Consider a halting sequence of moves of M and look at position (cell) p of the worktape, $p = 1, 2, ...n$, for some n. In the computation, position p will be visited many times. Let $t_1, ..., t_m$ be the times M visits p.

Corresponding to the time sequence $(t_1, ..., t_m)$ associated with position p, we define a *crossing vector* $R = (I_1, ..., I_m)$, where for each i, $I_i = (d_1, q_1, r_1, r_2, d_2)$,

1. d_1 is the direction from which the head entered p at time t_i;
2. q_1 is the state when it entered p;
3. r_1 is the instruction that was used in the move above;
4. r_2 is the instruction that was used at time $t_i + 1$ when it left p;
5. d_2 is the direction from which it left p at time $t_i + 1$.

We construct a TCM M' which simulates a halting computation of M by nondeterministically guessing the sequence of crossing vectors $R_1, ..., R_n$ as it processes the worktape from left to right, making sure that R_i and R_{i+1} are compatible for $1 \leqslant i \leqslant n$. Corresponding to each counter C of M, machine M' uses two counters C_1 and C_2. C_1 is used to record the increases in C, while C_2 is used to record the decreases in C. When M' completes the simulation of M, C_1 and C_2 must contain the same value, and this can easily be checked by M'.

The theorem follows from Corollary 1 since deciding whether a TCM (which has no input tape) has a halting sequence of moves is easily reducible to deciding the emptiness problem for a CM. □

The restriction that the worktape in a TCM is finite-crossing is necessary; otherwise (i.e., if it is unrestricted), the machine becomes a Turing machine. In fact, even for a special case, emptiness is undecidable. Restrict the worktape to be a pushdown stack which can only push (i.e., write) but *cannot* pop (i.e., erase), but can enter the stack in a read-only mode. Moreover, once it enters the stack in a read-only mode, it can no longer push. There is no restriction on the number of times the stack head can cross the boundary between any two stack cells. This restricted worktape is called a "checking" tape. Call this machine CCM. (CCM acceptors without counters have been studied in [Gre68].)

Theorem 3. *The emptiness problem for CCM acceptors is undecidable.*

Proof. The proof uses the undecidability of Hilbert's Tenth Problem (HTP) [Mat70], which is to decide for a given polynomial $p(x_1, ..., x_n)$ with integer coefficients whether it has a nonnegative integral root. We omit the construction, but the idea is to show that we can effectively construct, given a polynomial $p(x_1, ..., x_n)$, a CCM acceptor M_p such that p has no integral solution if and only if $L(M_p)$ is empty. □

We consider next binary reachability.

Theorem 4. *We can effectively construct, for a given PCM M, a PCM acceptor M' accepting $R(M)$ = the set of all pairs $(\alpha\,\beta)$ of configurations such that α can reach β in 0 or more transitions.*

Proof. Given a PCM M, we construct a PCM acceptor M' that accepts $R(M)$. M' when given $(\alpha\,\beta)$, reads configuration α and sets its counters and pushdown stack to α. Then M' simulates the computation of M starting in this configuration. At some point M' guesses that it has reached the configuration β, which it can verify by reading the input (note that since the pushdown is last-in-first-out, β must have the stack word given in "reversed"). □

The same construction works for a TCM acceptor, but the stack word in configuration β does not have to be given in reversed on the input, since the read/write head is two-way.

Theorem 5. *We can effectively construct, for a given TCM M, a TCM acceptor M' accepting $R(M)$.*

We now look at nonsafety.

Theorem 6. *We can effectively construct, given a PCM (TCM) M and two sets of configurations I (*initial set*) and B (*bad set*) accepted by CM acceptors, a PCM (TMC) M' that accepts a configuration α iff (i) α is in I, and (ii) M when started in α can reach a configuration in B. Thus nonsafety is decidable.*

Proof. We only prove the PCM case; the proof for TCM is similar. Let M_I and M_B be CM acceptors accepting I and B, resp. We construct a PCM acceptor M' which, when given an input α, sets its counters and pushdown stack to this configuration while also checking that the configuration is accepted by M_I . (Note that this can be done with additional counters *without* popping the stack). Then M' simulates the computation of M starting in this configuration. At some point M' guesses that it has reached a configuration β in B, which it can verify by simulating M_B on β. In the simulation of M_B, M' uses the pushdown stack which contains w to simulate the action of M_B on this input. M' reads w from the right by "popping". There is a slight problem in that M' will be working on the reverse of w, not w. However, it can be shown that if a language is accepted by a CM acceptor, then its reverse can also be accepted by a CM acceptor. Hence, we can use the CM acceptor accepting the reverse of the language accepted by M_B in the computation of M' to decide if β is in B. □

Corollary 2. *We can effectively construct, given a PCM (TCM) M and two sets of configurations I (*initial set*) and G (*good set*) accepted by CM acceptor and deterministic CM acceptor resp., a PCM (TMC) M' that accepts a configuration α iff (i) α is in I, and (ii) M when started in α can reach a configuration not in G. Thus invariance is decidable.*

Proof. It can be shown that if G is accepted by a deterministic CM acceptor M_G, then we can effectively construct a deterministic CM acceptor accepting the set of bad configurations B = the complement of G. (Note that this is not true if M_G is not deterministic.) The result follows from the above theorem. □

Next, we show that forward reachability is computable.

Theorem 7. *We can effectively construct, given a PCM (TCM) M and a set of configurations S accepted by a CM acceptor, a PCM (TCM) acceptor accepting $F_M(S)$ = the set of all configurations that can be reached from configurations in S in 0 or more transitions.*

Proof. Let M be a PCM and S be a set of configurations accepted by a CM acceptor M_S. We construct a PCM acceptor M', which when given a configuration β (with the stack word given in reverse), nondeterministically guesses a configuration α by simultaneously setting its counters and pushdown stack to this configuration and checking that α is in S. Then M' simulates the computation of M starting in this configuration. At some point, M' guesses that it has reached β, which it can check by reading the input. The proof for the case of TCM M is similar. □

Theorem 8. *We can effectively construct, given a PCM (TCM) M and a set of configurations S accepted by a CM acceptor, a PCM (TCM) acceptor accepting $B_M(S)$ = the set of all configurations that can reach configurations in S in 0 or more transitions.*

Proof. Let M be a PCM and S be a set of configurations accepted by a CM acceptor M_S. We construct a PCM acceptor M', which when given a configuration α, sets its counters and pushdown stack to this configuration. Then M' simulates the computation of M starting in this configuration. At some point M' guesses that it has reached a configuration in S (which is accepted by M_S) and verifies this as in the proof of Theorem 6. The proof for the case of TCM M is similar. □

Clearly, all the results above hold for CMs and QCMs. In fact, some of the results can be strengthen for CMs, for example:

Theorem 9. *Let M be a CM and S be a set of configurations. Then $B_M(S)$ is accepted by a CM acceptor iff S is accepted by a CM acceptor.*

Proof. The "if part" follows from the construction in Theorem 8. Conversely, given M and S, suppose $B_M(S)$ is accepted by a CM acceptor M'. We show that S can be accepted by a CM acceptor M''.

Given β on its input tape, M'' guesses and stores in $k+1$ counters a configuration α. By using additional counters and employing M', M'' checks that α is in $B_M(S)$. Then M'' simulates M to check that α can reach β. □

We say that a set of configurations S of a CM, which are strings of form $0^q\#0^{i_1}\#0^{i_2}\#...\#0^{i_k}$, is Presburger if the set of $(k+1)$-tuples $(q, i_1, ..., i_k)$ corresponding to the configurations in S is definable by a Presburger formula.

It known that a set of configurations is accepted by a CM acceptor if and only if it is Presburger [Iba78]. Hence, we have:

Corollary 3. *Let M be a CM and S be a set of configurations. Then $B_M(S)$ is Presburger if and only if S is Presburger.*

The last two results above also hold for $F_M(S)$.

4 Extensions of the Models

In this section we look at some extensions of PCM (TCM) acceptors. We will only deal with the emptiness problem since the other problems (reachability, safety, etc.) are reducible to emptiness, as we have seen in the previous section. The proofs of the theorems below are generalizations of the proofs of similar results for the corresponding extensions of CMs in [ISDBK00]. Related results can be found in [FS00], where decidability of reachability problems for some classes of two counter machines which allow resetting a counter to zero and transferring the value of one counter to another were shown.

The first extension is to allow the counters of a PCM (TCM) to store negative numbers, and allow the counters to increment/decrement by c and allow tests of the form: "Is $x\theta c$?", where x is a counter, c is any integer constant (there are many such constants in the specification of the machine), positive, negative, or zero, and θ is one of $<, >, =$.

One can easily show that any PCM (TCM) M that uses the generalized instructions above can be converted to an equivalent machine M' using standard instructions such that $L(M) = L(M')$. The construction of M' is straightforward. M' "remembers" the signs of the counters in the states, so the counters do not have to store negative values. To handle predicates like $x < c$, M' uses fixed-size "buffers" in the states to translate the origin, etc.

Next, consider a PCM (TCM) that allows tests like, "Is $5x - 3y + 2z < 7$?", where x, y are counters. To be precise, let V be a finite set of variables over integers. An *atomic linear relation* on V is defined as $\sum_{v \in V} a_v v < b$, where a_v and b are integers. A *linear relation* on V is constructed from a finite number of atomic linear relations using $\neg$ and $\wedge$. Note that standard symbols like $>, =, \rightarrow$ (implication), $\vee$ can also be expressed using the above constructions.

Suppose we allow a PCM (TCM) M to use tests of the form: " Is L ", where L is a linear relation on the counters. The emptiness problem becomes undecidable, even for deterministic CMs. In fact, the following can be shown using the undecidability of the halting problem for unrestricted two counter machines [Min61]:

Consider only deterministic CMs (no input) with 3 counters (initially zero) which can only be incremented by 0 or 1 (thus decrementing is *not* allowed), and the only tests are of the form "Is $x = y$?", where x, y are counters. The halting problem for such machines is undecidable.

Note that in the above undecidability, the 3-counter CM is 0-reversal bounded because the mode of each of the counters is always nondecreasing. Interestingly, the emptiness problem is decidable for 2-counter CMs using the standard instructions and the test "Is $x = y$?". This follows from Theorem 1 and the observation that the pushdown stack can be used to keep track of the difference $x - y$.

In defining reversal-boundedness, we only had two modes: nondecreasing and nonincreasing. Suppose we refine these to three modes: increasing, decreasing, no-change. Say that a counter is *mode-bounded* if the number of changes in its mode during any computation is bounded by a constant. Clearly, a counter that is mode-bounded is reversal-bounded, but the converse is not true. For example, a counter that displays the pattern "$122334455\cdots$" correspond to 0-reversal, but is not mode-bounded (since although it is nondecreasing, the number of changes from no-change to increasing and vice-versa is unbounded). Call a PCM (TCM) mode-bounded if the counters are mode-bounded. We can show the following:

Theorem 10. *The emptiness problem is decidable for mode-bounded PCM (TCM) acceptors that can use linear-relation tests on the counters.*

We can further generalize the machines by allowing parameterized constants in the linear relations. So for example, we can allow tests like "Is $3x-5y-2D_1+9D_2 < 12$?", where D_1 and D_2 represent parameterized constants whose domain is the set of all integers $(+, -, 0)$. We can specify the values of these parameters by including them in the input tape. Thus, the input to the machine with t parameterized constants will have the form: "#d_1%$\cdots$%d_k%w#", where $d_1, ..., d_k$ are integers $(+, -, 0)$ that the parameterized constants $D_1, ..., D_k$ assume for this run, and % is a separator. We assume that the d_i's are represented in unary along with their signs. We can prove the following:

Theorem 11. *The emptiness problem is decidable for mode-bounded PCM (TCM) acceptors that can use linear-relation tests on the counters and parameterized constants.*

As we have seen, Theorem 10 is not true for machines whose counters are reversal-bounded but not mode-bounded. However, suppose we require that in every linear- relation L, every atomic linear-relation in L involves only the parameterized constants and at most one counter so, e.g., $4D_1 + 9D_2 < 7$ and $5x - 4D_1 + 9D_2 < 7$ are allowable, but $5x + 2y - 4D_1 + 9D_2 < 7$ is not (where x and y are counters, and D_1 and D_2 are parameterized constants). Call such a relation L a *restricted linear-relation.* Then we can prove:

Theorem 12. *The emptiness problem is decidable for PCM (TCM) acceptors that can use restricted linear-relation tests on the counters and parameterized constants.*

5 Conclusions

We introduced some new models of infinite-state transition systems by augmenting the finite-state machine with reversal-bounded counters and suitably restricted data structures (such as pushdown stack, queue, read/write tape) and showed that emptiness, (binary, forward, backward) reachability, nonsafety, and invariance for these models are solvable. We also studied generalizations of the models.

As we have seen in Examples 2 and 3, the results can be used to verify properties that are not verifiable using previous techniques. We give some more examples below.

For a configuration α, α_{x_i} and α_w denote the value of counter x_i and the stack word (or worktape contents) of α, respectively. $\#_a(w)$ denotes the number of occurences of symbol a in a stack word (or worktape contents) w.

Consider the following property concerning a PCM M:

> *For any pair of configurations* (α, β) *if* α *reaches* β, *then* $(\beta_{x_2} = \alpha_{x_1} + 2\alpha_{x_2} \wedge 2\#_a(\beta_w) = 3\#_b(\alpha_w))$.

This property can be verified, by showing that its negation can be verified. From Theorem 4, $R(M)$ can be accepted by a PCM acceptor M'. We construct a PCM acceptor M'' which simulates M' and at the same time (using additional counters) checks that $(\beta_{x_2} = \alpha_{x_1} + 2\alpha_{x_2} \wedge 2\#_a(\beta_w) = 3\#_b(\alpha_w))$ is false. Then $L(M'')$ is empty if and only if the property is false. Hence, the property can be verified. Note that:

- Even without counters, $2\#_a(\beta_w) = 3\#_b(\alpha_w)$ defines a nonregular set of stack word pairs. Hence, this property cannot be verified by the model checking procedures for pushdown systems [BEM97,FWW97,Wal96].
- Even without the pushdown stack, $\beta_{x_2} = \alpha_{x_1} + 2\alpha_{x_2}$ is not a "clock region" [AD94]. Hence, the classical region technique cannot verify this property. This is also pointed out in [CJ99].

As another example, since invariance is decidable from Corollary 2, we can verify the following property concerning a TCM M:

> *Starting from a configuration* α *satisfying:* $\#_a(\alpha_w) = \#_b(\alpha_w)$, M *can only reach a configuration* β *satisfying:* $(\beta_{x_1} = 2\beta_{x_1} + \beta_{x_2} \wedge \#_a(\beta_w) = 3\#_b(\beta_w))$.

The reason is that set of α's and β's satisying the the stated properties can be accepted by deterministic CM acceptors.

In the future we would like to investigate the decidability of liveness properties for the computational models we presented in this paper. More generally we would like to consider decidability of various temporal logic properties such as CTL, LTL, and μ-calculus. We would also like to investigate the complexity of verification procedures for these infinite-state models.

References

ACD93. R. Alur, C. Courcoibetis, and D. Dill. Model-checking in dense real time. *Information & Computation*, 104(1):2-34, 1993. 184

AD94. R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science*, 126(2):183-236, 1994. 184, 196

AH94. R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181-204, 1994. 184

BCM92. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. H. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information & Computation*, 98(2):142–170, June 1992. 183

BEM97. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. *CONCUR 1997*, pp. 135-150. 184, 196

BER95. A. Bouajjani, R. Echahed and R. Robbana. "On the Automatic Verification of Systems with Continuous Variables and Unbounded Discrete Data Structures." In *Hybrid Systems II*, LNCS 999, 1995. 184

BG96. B. Boigelot and P. Godefroid. "Symbolic verification of communication protocols with infinite state spaces using QDDs." In *Proc. Int. Conf. on Computer Aided Verification*, 1996. 184

BGP99. T. Bultan, R. Gerber, and W. Pugh. Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and Experimental Results. *ACM Trans. on Programming Languages and Systems*, 21(4):747-789, July 1999. 184

BLO98. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. 10th Int. Conf. on Computer Aided Verification*, 1998. 183

BW94. B. Boigelot and P. Wolper, Symbolic verification with periodic sets, *Proc. 6th Int. Conf. on Computer Aided Verification*, 1994 184

CJ98. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. *Proc. 10th Int. Conf. on Computer Aided Verification*, 1998. 184

CJ99. H. Comon and Y. Jurski. Timed Automata and the Theory of Real Numbers. *Proc. CONCUR*, 1999. 196

DIBKS00. Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. To appear in *Int. Conf. on Computer Aided Verification, 2000.* 184

DF95. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction. *Proc. 7th Int. Conf. on Computer Aided Verification*, 1995. 183

DGG97. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. on Programming Languages and Systems*, 19(2):253-291, March 1997. 183

Esp97. J. Esparza. Decidability of Model Checking for Infinite-State Concurrent Systems. *Acta Informatica*, 34(2):85-107, 1997. 184

FS00. A. Finkel and G. Sutre. Decidability of reachability problems for classes of two counters automata. *STACS'2000*, 346-357, Springer, 2000. 184, 194

FWW97. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *INFINITY, 1997.* 184, 196

GI81. E. M. Gurari and O. H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. *JCSS*, 22:220-229, 1981. 188, 190

Gre68. S. A. Greibach. Checking automata and one-way stack languages. *SDC Document TM 738/045/00*, 1968. 191

HNSY94. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information & Computation*, 111(2):193-244, 1994. 184

HRP94. N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In *Proc. Int. Symposium on Static Analysis*, B. LeCharlier ed., vol. 864, September 1994. 184

Iba78. O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, Vol. 25, pp. 116-133, 1978. 184, 188, 190, 194

IS99. O. H. Ibarra and J. Su, A Technique for the Containment and Equivalence of Linear Constraint Queries. *JCSS*, 59(1):1-28, 1999. 188

ISB00. O. H. Ibarra, J. Su, and C. Bartzis, Counter Machines and the Safety and Disjointness Problems for Database Queries with Linear Constraints, to appear in *Words, Sequences, Languages: Where Computer Science, Biology and Linguistics Meet*, Kluwer, 2000. 188

ISDBK00. O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. Kemmerer, Counter Machines: Decidable Properties and Applications to Verification Problems, To appear in *Proc. MFCS'2000*. 194

Mat70. Y. Matijasevic. Enumerable sets are Diophantine. *Soviet Math. Dokl*, Vol. 11, 1970, pp.354-357. 192

Min61. M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74:437-455, 1961. 184, 194

Pos46. E. Post. A variant of a recursively unsolvable problem. *Bull. Am. Math. Soc.*, 52:264-268, 1946. 188

Wal96. I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. Int. Conf. on Computer Aided Verification*, 1996 184, 196

WB98. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. Int. Conf. on Computer Aided Verification*, 1998. 184

Process Spaces

Radu Negulescu

Department of Electrical and Computer Engineering
McGill University
3480 University St., Montreal, Quebec, Canada H3A 2A7
radu@macs.ece.mcgill.ca
http://www.macs.ece.mcgill.ca/~radu

Abstract. This paper presents process spaces, a general theory for systems that are made of several interacting components and layers of abstraction. Process spaces capture in a homogeneous manner diverse correctness concerns for concurrent systems and other types of systems. The main innovation of process spaces is the idea of abstract execution, which permits to set the precision and complexity of a system model by the amount of detail included in the execution model. Notwithstanding this generality, we show that process spaces admit several algebraic properties of practical significance.

1 Introduction

Over the past few decades, concurrency theory has been emerging as a modeling paradigm for diverse types of discrete-state systems. Examples of such systems include digital circuits, distributed programs and data structures, communication protocols, and even work flows in a factory. With the spread of component-based development of distributed object-oriented software and the expected upheaval of timing issues in the new generations of integrated circuits, a fundamental understanding of concurrency issues is in high demand.

This paper presents process spaces, a theory that captures common aspects of concurrent systems and even certain dense-state systems. In process spaces, the notion of 'execution' is abstract, and is a primitive notion. An execution may be a sequence of events, a function of time, etc.; a priori, an execution has no structure. As a result, process spaces can capture a large spectrum of correctness concerns by varying the amount of detail contained in the executions. Another consequence of the abstract executions is that several meaningful properties of such systems boil down to elementary set-theoretical properties.

We generalize several operations on processes and notions of correctness that form the core of concurrency theory: relative and absolute correctness, parallel composition, etc. We discuss several algebraic properties of these conditions and operations, as well as their use for handling processes in a structured manner.

For concurrent systems, process spaces can model diverse correctness concerns, such as safety, liveness, and even absence of dangling inputs. These applications

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 199-213, 2000.

are separate instances of the same theory, for different types of executions. These applications are decoupled (e.g., the study of liveness has no safety or connectivity restrictions) and homogeneous (they have precisely the same algebraic properties).

2 The Basic Formalism

Let $\mathcal{E}$ be an arbitrary set; we refer to the elements of $\mathcal{E}$ as *executions*.

Definition 1 A *process* over $\mathcal{E}$ is a pair (X, Y) of subsets of $\mathcal{E}$ such that $X \cup Y = \mathcal{E}$. The set of all processes over $\mathcal{E}$ is called the *process space* of $\mathcal{E}$, denoted by $S_{\mathcal{E}}$.

A process (X, Y) represents a contract between a device and its environment: the device guarantees that only executions from X occur, while the environment guarantees that only executions from Y occur. A process (X, Y) partitions the underlying execution set into disjoint subsets $\overline{X}$, $X \cap Y$, and $\overline{Y}$ (see Fig. 1; $\overline{\ }$ represents complementation). The contract also qualifies 'illegal' executions, by assigning the responsibility of avoiding them either to the device or to the environment. Set $\overline{X}$ contains the executions in which the device violates the contract, while set $\overline{Y}$ contains the executions in which the environment violates the contract. The condition $X \cup Y = \mathcal{E}$, or, equivalently, $\overline{X} \cap \overline{Y} = \varnothing$, formalizes a separation of responsibilities between the device and its environment: the 'blame' for not avoiding a certain execution cannot be assigned to both device and environment.

We use the following notation for the execution sets of a process $p = (X, Y)$.

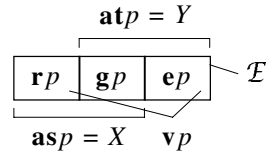


Fig. 1. Execution sets of a process.

as $p = X$ = the *accessible* set of p,
at $p = Y$ = the *acceptable* set of p,
v $p = \overline{X} \cup \overline{Y}$ = the *violation* set of p,
r $p = \overline{Y}$ = the *reject* set of p,
g $p = X \cap Y$ = the *goal* set of p,
e $p = \overline{X}$ = the *escape* set of p.

These terms were chosen to refer to the contract described above, as seen by the device (rather than the environment). For example, the acceptable executions are acceptable to the device.

Refinement is a binary relationship $\sqsubseteq$ on $S_{\mathcal{E}}$ such that

$$(X_1, Y_1) \sqsubseteq (X_2, Y_2) \;\Leftrightarrow\; X_1 \supseteq X_2 \;\wedge\; Y_1 \subseteq Y_2. \qquad \textbf{(1)}$$

Intuitively, refinement represents a *relative* notion of correctness: $p \sqsubseteq q$ means that

p may be substituted by q. A better process accepts more executions, thus its environment has weaker constraints. Also, a better process accesses fewer executions, thus its device obeys tighter constraints. Examples will be given in Section 3, including an example of detecting unfairness faults as violations of refinement.

Reflection is a unary operation $-$ on $S_{\mathcal{I}}$ such that

$$-(X, Y) = (Y, X) \,. \tag{2}$$

Informally, if a process p represents a contract seen by the device, then reflection turns the table: $-p$ represents the corresponding contract seen by the environment.

Product ($\times$) and *exclusive sum* ($\oplus$) are binary operations on $S_{\mathcal{I}}$ such that

$$(X_1, Y_1) \times (X_2, Y_2) = (X_1 \cap X_2,\ Y_1 \cap Y_2 \cup \overline{X_1 \cap X_2}) \,, \tag{3}$$

$$(X_1, Y_1) \oplus (X_2, Y_2) = (X_1 \cap X_2 \cup \overline{Y_1 \cap Y_2},\ Y_1 \cap Y_2) \,. \tag{4}$$

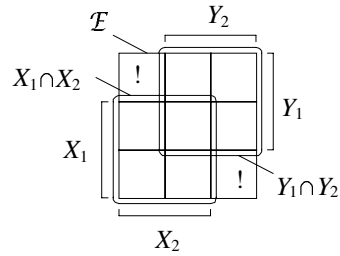


Fig. 2. Deriving the product and exclusive sum operators.

The product models a system formed by two devices operating jointly. The new system's accessible executions include all executions that are accessible to both components, and its acceptable executions include all executions that are acceptable to both components. In Fig. 2, we see that the executions marked with '!' are so far not accounted for. These executions are escapes for the resulting system, because they should be avoided by one of the component devices. Accordingly, these executions are acceptable to the resulting system. In the definition of product, note that $\overline{X_1 \cap X_2} \setminus (Y_1 \cap Y_2) = \overline{X_1} \cap \overline{Y_2} \cup \overline{Y_1} \cap \overline{X_2}$, which means that the acceptable set is augmented from $Y_1 \cap Y_2$ by precisely the executions marked '!'.

In the exclusive sum, we consider the '!' executions to be rejects, rather than escapes, for the resulting system. This amounts to blaming the environments, rather than the devices, for not avoiding these executions. (In this sense, the exclusive sum assumes that the environments, rather than the devices, operate jointly.)

Top, denoted by $\top$, is the process $(\emptyset, \mathcal{E})$, where $\emptyset$ is the empty set. *Bottom*, denoted by $\bot$, is the process $(\mathcal{E}, \emptyset)$. *Void*, denoted by Φ, is the process $(\mathcal{E}, \mathcal{E})$.

A process (X, Y) is *robust* if $Y = \mathcal{E}$. A process (X, Y) is *chaotic* if $X = \mathcal{E}$. The sets of robust and chaotic processes over $\mathcal{E}$ are denoted by $\mathcal{R}_{\mathcal{I}}$ and $\mathcal{C}_{\mathcal{I}}$, respectively. Robust and chaotic processes can be interpreted as 'pure guarantees' and 'pure assumptions', respectively. Robustness also represents an *absolute* notion of

correctness: the fact that a process has no rejects means that it needs no guarantees from the environment and it can operate any environment conditions. Chaotic processes are so called because any execution is accessible to them. Void is both robust and chaotic.

3 Examples of Modeling

The process space formalism is applied to particular correctness concerns and types of systems by choosing an execution set. The entire construction in Section 2 has the execution set as its sole parameter; choosing an execution set amounts to instantiating the construction. In examples, we show how refinement and robustness, the process space conditions for relative and absolute correctness, can be used to model specific correctness concerns and detect faults in various types of systems.

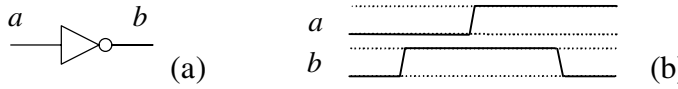


Fig. 3. Inverter: (a) signals; (b) waveform.

Some of our examples refer to concurrent systems and their applications. Concurrent systems have discrete state spaces and operate in continuous time. The state transitions are called *events*. Each event is associated with an *action*, or port, of a concurrent system. Actions are from a set $\mathcal{U}$, called the *action universe*. For example, the behavior of the inverter in Fig. 3 (a) can be modeled by associating actions to the terminals of the inverter. Each action represents a terminal: the action universe includes $\{a, b\}$. Each event represents a voltage transition on the respective terminal: the waveform in Fig. 3 (b) can be represented by the sequence of events *bab*.

For simplicity, events are often assumed to be instantaneous and not simultaneous, and we also adopt this assumption in several examples. However, process spaces are by no means bound to this assumption or even to event-based executions; a 'true concurrency' process space, whose executions allow simultaneous events, is briefly discussed in [11] and [13].

The global scope of events (all processes have the same execution set) is not a limitation; events are ignored by a state machine, for instance, if they produce self-loops at every state.

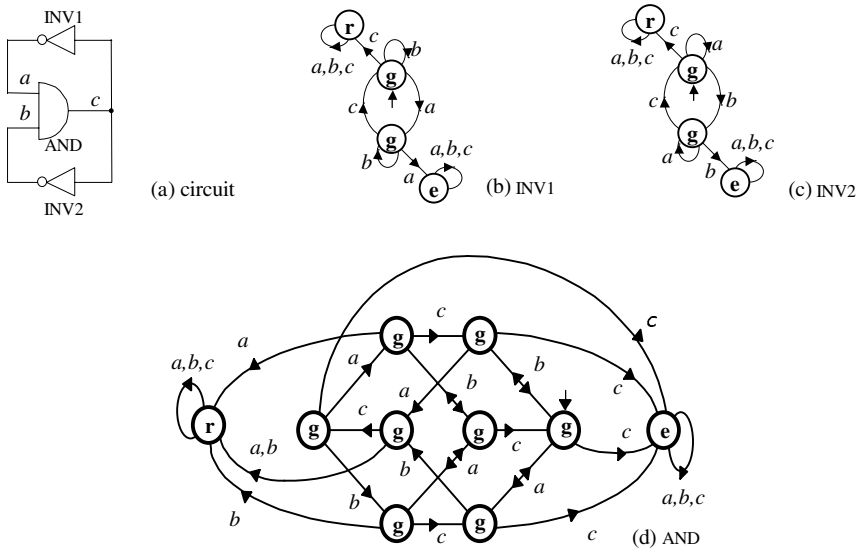


Fig. 4. Circuit and processes for Example 1.

Example 1 Informally speaking, safety properties of concurrent systems assert that 'something bad does not happen' [8]. For studying safety, we take the execution set to be $\mathcal{U}^*$ (the set of finite sequences over $\mathcal{U}$). Each concurrent system s is modeled by a process σs over $\mathcal{U}^*$, whose executions represent finite sequences of actions that can be observed up to a certain time; call it the *safety process* of s.

For instance, Fig. 4 (a) represents a circuit of two inverters and an AND gate. Let σINV1, σAND, and σINV2 be the safety processes of the three gates in Fig. 4 (a). These processes are models of the hazard-free behavior of the respective gates. Assuming $\mathcal{U} = \{a, b, c\}$, the goal, reject, and escape sets of these processes are represented by the state graphs in Fig. 4 (b), (c), and (d): for example, word *abca* is a goal of σAND, because it leads from the initial state, marked by an incoming arrow, to a state marked **g** (for 'goal'). Certain output transitions, such as a c event from the initial state in Fig. 4 (d), are ruled out by taking the corresponding words to lead to a state marked **e**. Hazards are ruled out by forbidding those input events that disable output events. For example, two consecutive transitions on c are omitted from the goal set of INV1, because the second c would disable the output event a and thus constitutes a hazard; the second c leads to a state marked **r**.

Notice that events whose actions are neither inputs nor outputs of a gate can occur arbitrarily in the state graph of that gate (such events are not 'seen' by the gate). Events from outside the alphabets of a gate produce self-loops in every state.

Next, we check the robustness of the product of the safety processes representing the parts of the circuit. For the circuit in Fig. 4 (a), the condition is not satisfied. Let $u = abcac$. We note that $u \in \mathbf{as}\sigma\text{INV1}$, $u \in \mathbf{as}\sigma\text{AND}$, and $u \in \mathbf{as}\sigma\text{INV2}$, but $u \notin \mathbf{at}\sigma\text{INV2}$: u leads to states marked **g** in Fig. 4 (b) and (d), but to a state marked **r** in Fig. 4 (c). After some set manipulations, it follows that $u \notin \mathbf{at}\,(\sigma\text{INV1} \times \sigma\text{AND} \times \sigma\text{INV2})$, and thus $\sigma\text{INV1} \times \sigma\text{AND} \times \sigma\text{INV2} \notin \mathcal{R}_{\mathcal{U}_*}$.

```
task body P0 is begin
 loop
    select
       Critical_Section_1;
    or
       Critical_Section_2;
    end select;
 end loop;
end P0;
```

Fig. 5. Specification for Example 2.

The violation of robustness can be interpreted as follows. The offending word (u) represents a hazard. After *abca*, both the input and the output signals of INV2 are high, and an output event is enabled. However, another input transition c can occur first, changing the input voltage to low and disabling the output transition.

Therefore, our robustness condition can be used to detect hazards in an asynchronous digital circuit, by using safety models that consider hazards illegal, similar to [4], p. 46, and the 'stability' condition in [16], p. 165.

Example 2 Informally speaking, liveness properties of concurrent systems assert that 'something good eventually does happen' [8]. For studying liveness, we take the execution set to be $\mathcal{U}^\infty$ (the set of finite or infinite sequences over $\mathcal{U}$), and we consider that each concurrent system s is represented by its *liveness process*, which is a process λs over $\mathcal{U}^\infty$ whose executions are finite or infinite sequences of events that can be observed until the 'end of time'.

To illustrate liveness, let us consider a classical example of starvation (an unfairness fault), adapted from [1], p. 35. The concurrent program in Fig. 6 attempts to ensure mutual exclusion between the critical sections of tasks `P1` and `P2` by using variables `C1` and `C2`. `C1` and `C2` are initially set at `1`. We state the specification as the task `P0` in Fig. 5. Let the actions be

r_{xyz} = Px reads C$y = z$,
w_{xyz} = Px sets Cy to z,
ecs_x = enter `Critical_Section_`x,
lcs_x = leave `Critical_Section_`x,
$encs_x$ = enter `Non_Critical_Section_`x,
$lncs_x$ = leave `Non_Critical_Section_`x.

Consider infinite execution $u = encs_2 lncs_2 (encs_1 lncs_1 w_{110} r_{121} w_{220} r_{210} ecs_1 lcs_1 w_{111} w_{221})^\omega$. Word u is an accessible complete execution for `P1`, because `P1` executes its main loop, in which it can stay forever. Word u is also an accessible complete execution for `P2`, because `P2` executes its inner loop and reads `C1 = 1` at every iteration, and thus can stay in its inner loop forever. On the other hand, we take u not to be an accessible complete execution for `P0`, because `P0` executes a non-

```
            C1, C2: Integer range 0..1 := 1;

task body P1 is begin          task body P2 is begin
  loop                           loop
    Non_Critical_Section_1;        Non_Critical_Section_2;
    loop                           loop
      C1 := 0;                       C2 := 0;
    exit when C2 = 1;              exit when C1 = 1;
      C1 := 1;                       C2 := 1;
    end loop;                      end loop;
    Critical_Section_1;            Critical_Section_2;
    C1 := 1;                       C2 := 1;
  end loop;                        end loop;
end P1;                        end P1;
```

Fig. 6. Program for Example 2.

deterministic choice in its loop (between ecs_1 and ecs_2), and, for any non-zero probability of ecs_2, P0 should eventually choose ecs_2. (We have used 'strong liveness with respect to outputs' [13] as a guide for determining the goals of λP1, λP2, and λP0.) Thus, $u \in \mathbf{as}(\lambda\mathtt{P1} \times \lambda\mathtt{P2})$ and $u \notin \mathbf{as}\lambda\mathtt{P0}$. Consequently, we have $\mathbf{as}\lambda\mathtt{P0} \not\supseteq \mathbf{as}(\lambda\mathtt{P1} \times \lambda\mathtt{P2})$ and thus $\lambda\mathtt{P0} \not\subseteq (\lambda\mathtt{P1} \times \lambda\mathtt{P2})$.

The violation of refinement can be interpreted as follows. Word u causes starvation because it never allows P2 to enter its critical section.

Example 3 Process spaces can also be used for modeling properties of steady-state electrical networks whose parameters (coefficients) are not precisely known.

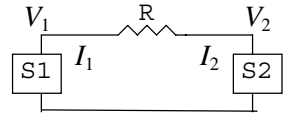


Fig. 7. Network for Example 3.

If there are n real state variables of interest in the network, we take the execution set to be $\mathbb{R}^n$. For example, Fig. 7 represents a steady-state circuit. The outputs of two voltage sources S1 and S2 are connected by a resistor R. There are four variables of interest: V_1, I_1, V_2, and I_2, the output voltages and currents of the two sources; correspondingly, we take the executions to be vectors (V_1, I_1, V_2, I_2) from $\mathbb{R}^4$. Suppose source $\mathtt{S}_x$ delivers an electromotive force between $V_{x\min}$ and $V_{x\max}$ if I_x is between $I_{x\min}$ and $I_{x\max}$. We represent source S1 by a processes ηS1:

$$\mathbf{g}\eta\mathtt{S1} = \{(V_1,I_1,V_2,I_2) \in \mathbb{R}^4 \mid V_{1\min} \leqslant V_1 \leqslant V_{1\max} \text{ and } I_{1\min} \leqslant I_1 \leqslant I_{1\max}\}, \quad \textbf{(6)}$$

$$\mathbf{at}\eta\mathtt{S1} = \{(V_1, I_1, V_2, I_2) \in \mathbb{R}^4 \mid I_{1\min} \leqslant I_1 \leqslant I_{1\max}\}. \quad \textbf{(7)}$$

and source S2 by a similar processes ηS2.

Also suppose the resistance of R is r and that R can operate under any voltages. Accordingly, we take

$$\eta\mathtt{R} = (\{(V_1,I_1,V_2,I_2) \in \mathbb{R}^4 \mid I_1 + I_2 = 0 \text{ and } r\,I_1 = V_1 - V_2\},\ \mathbb{R}^4). \quad \textbf{(8)}$$

Now, suppose that $V_{1\min} = V_{2\min} = 4.9\text{V}$, $V_{1\max} = V_{2\max} = 5.2\text{V}$, $I_{1\max} = I_{2\max} = 25\text{mA}$, $I_{1\min} = I_{2\min} = -25\text{mA}$, and $r = 10\Omega$. Robustness of product is not satisfied for parameters in these ranges. For execution $z = (5.2, 0.03, 4.9, -0.03)$, we have $z \in \mathbf{as}\eta\mathtt{S1} \cap \mathbf{as}\eta\mathtt{S2} \cap \mathbf{as}\eta\mathtt{R}$, but $z \notin \mathbf{at}\eta\mathtt{S1}$. Thus, $\eta\mathtt{S1} \times \eta\mathtt{S2} \times \eta\mathtt{R} \notin \mathcal{R}_{\mathbb{R}^4}$.

The violation of robustness can be interpreted as follows. Due to slack in the values of the electromotive forces, a short with a current larger than 25mA may occur, which may damage the sources.

Example 4 The process space correctness conditions are also suitable for studying connectivity concerns, such as input control (absence of dangling inputs). For this, we take the execution domain to be the universe of actions itself. In other words, we consider that each concurrent system s is represented by its *action control process*, a process κs over $\mathcal{U}$, where executions are labeled as reject, goal, or escape, according to whether the corresponding actions are 'controlled' by the system or not: inputs are rejects, outputs are escapes, and all other actions are goals.

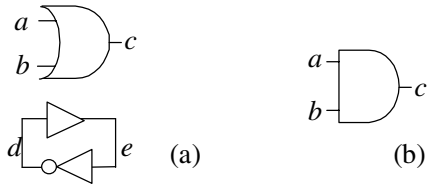


Fig. 8. Circuit and gate for Example 4.

In this process space, robustness amounts to the absence of dangling inputs. Let the components of the circuit in Fig. 8 (a) be OR, BUF, and INV, from top to bottom. Let $\mathcal{U} = \{a, b, c, d, e\}$. For each component in Fig. 8 (a), we construct the input control process as described above. For instance, $\kappa\text{OR} = (\{a, b, d, e\}, \{c, d, e\})$. After some straightforward manipulations, we obtain $\kappa\text{OR} \times \kappa\text{BUF} \times \kappa\text{INV} = (\{a, b\}, \{c, d, e\}) \notin \mathcal{R}_{\{a,\,b,\,c,\,d,\,e\}}$, thus the robustness condition is not satisfied by the circuit. Informally, notice that $\mathcal{U} \setminus \mathbf{at}(\kappa\text{OR} \times \kappa\text{BUF} \times \kappa\text{INV}) = \{a, b\}$ indicates precisely the two dangling inputs in the circuit.

Input control can also be considered in a relative sense. Let us check whether the circuit in Fig. 8 (a) is a correct implementation of the AND gate in Fig. 8 (b), with respect to input control. For that, we demand $\kappa\text{AND} \sqsubseteq \kappa\text{OR} \times \kappa\text{BUF} \times \kappa\text{INV}$. We have, as above, $\mathbf{as}(\kappa\text{OR} \times \kappa\text{BUF} \times \kappa\text{INV}) = \{a, b\}$, which is a subset

of $\{a, b, d, e\} = \mathbf{as}\kappa\mathrm{AND}$. Also, $\mathbf{at}(\kappa\mathrm{OR} \times \kappa\mathrm{BUF} \times \kappa\mathrm{INV}) = \{c, d, e\} = \mathbf{at}\kappa$ AND. Thus, the refinement condition is satisfied. This is because, informally speaking, the two dangling inputs of the circuit are controlled by the environment of the AND gate.

4 Process Space Structure

In this section, we examine some of the algebraic properties of process spaces and their significance. Proofs can be found in [13].

Proposition 1 *Product and exclusive sum are idempotent, associative, and commutative.*

Idempotency ensures that composing a device with an identical replica of itself produces an identical device. Although in many applications we cannot connect a device with its identical replica because of output interference, there are situations where we can. In [14] and [12], certain metric-free timing constraints are modeled as devices that have common outputs with gates in a circuit; we find that connecting such a constraint with an identical replica of itself should indeed produce the same constraint. Commutativity and associativity ensure that the order of composing devices in parallel does not matter.

Proposition 2 *Refinement is reflexive, transitive, and antisymmetric.*

Proposition 2 shows that refinement is a partial order.

A consequence of the antisymmetry property is that process spaces are a fully abstract model. Full abstraction requires that the equivalence relationship of the model does not make more identifications than the equality relationship of the model. The natural equivalence in process spaces is double refinement; equality of processes is equality of their execution sets. The antisymmetry property states that double refinement implies equality.

Refinement induces a complete distributive lattice on a process space. The induced meet ($\sqcap$) models non-deterministic choice between processes, while the induced join ($\sqcup$) models non-deterministic choice between environments.

Theorem 3 (*Monotonicity*) *For processes p, q and r,*

(a) $p \sqsubseteq q \;\Rightarrow\; p \times r \sqsubseteq q \times r\,,$ (*monotonicity of* $\times$ *with respect to* $\sqsubseteq$)

(a′) $p \sqsubseteq q \;\Rightarrow\; p \oplus r \sqsubseteq q \oplus r\,.$ (*monotonicity of* $\oplus$ *with respect to* $\sqsubseteq$)

Note that Theorem 3 does not have restrictions on the ports of the devices represented by p, q, and r. (There are no ports in the general process spaces.) For the safety and liveness conditions deriving from $\sqsubseteq$, this absence of restrictions may be

surprising. For example, the device of r may have common internal ports with the device of q, but not with the device of p, while Theorem 3 still holds.

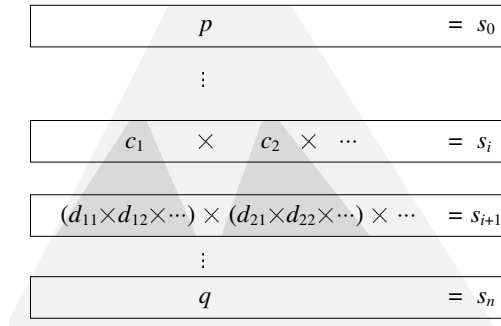


Fig. 9. Modular and hierarchical verification.

The properties of $\sqsubseteq$ and $\times$ allow for hierarchical and modular verification. The problem is to determine whether $p \sqsubseteq q$, where p represents a specification and q an implementation of a system of several components. To establish refinement between the specification process and the overall implementation, we typically devise a chain of intermediate specifications $s_0, s_1, \ldots, s_n$ such that $s_0 = p$ and $s_n = q$ (see Fig. 9). Consecutive specifications (including p and including q) may be broken into components: $s_i = c_1 \times c_2 \times \cdots$ and $s_{i+1} = (d_{11} \times d_{12} \times \cdots) \times (d_{21} \times d_{22} \times \cdots) \times \cdots$. It suffices to verify, for each j, that $c_j \sqsubseteq d_{j1} \times d_{j2} \times \cdots$; then, by commutativity and monotonicity of $\times$, and transitivity of $\sqsubseteq$, $p \sqsubseteq q$ is established.

Modular and hierarchical verification can reduce computational costs by breaking the overall verification problem into smaller problems. This approach pays off because, as we show in [13], verification of robustness of product of processes is PSPACE-hard for execution sets that are regular languages of finite words.

Proposition 4 *For process p,*

(a) $p \times \Phi = p$, **(a′)** $p \oplus \Phi = p$. (*identity element*)

Proposition 4 ensures that composition with a ‘block of wood’ process does not affect the rest of the system - a ‘block of wood’ accepts everything but does nothing useful.

Proposition 5 *For processes p and q,*

(a) $--p = p$,
(b) $p \sqsubseteq q \;\Leftrightarrow\; -q \sqsubseteq -p$,
(c) $-(p \times q) = -p \oplus -q$, **(c′)** $-(p \oplus q) = -p \times -q$.

Proposition 5 (a) and (b) show that reflection is its own inverse and turns around the refinement relationship. Proposition 5 (c) and (c') constitute De Morgan's laws for product and exclusive sum, with respect to reflection. A consequence of Proposition 5 is that process spaces have a complete duality with respect to reflection.

Remark (*Duality Principle*) Let X be a statement about process spaces. The *dual* of X is a statement X^{∂} obtained by replacing in X every occurrence of $\sqsubseteq$ by $\sqsupseteq$, of $\sqcup$ by $\sqcap$, of $\times$ by $\oplus$, of $\mathcal{R}_{\mathcal{I}}$ by $\mathcal{C}_{\mathcal{I}}$, of **as** by **at**, of **r** by **e**, and conversely. ($-$, Φ, $\mathcal{S}_{\mathcal{I}}$, **g**, and **v** are their own duals.) Notice that $X^{\partial\partial} = X$. Process spaces admit the following duality principle: if statement X holds, then X^{∂} holds, too.

Theorem 6 (*Verification*) *For processes p and q,*

$$p \sqsubseteq q \;\Leftrightarrow\; -p \times q \in \mathcal{R}_{\mathcal{I}} \;. \tag{9}$$

Theorem 6 links the absolute and relative notions of correctness in process spaces. Theorem 6 permits to verify whether an implementation satisfies (refines) a specification by placing the implementation in the environment of the specification, and then checking the robustness of their product. Such approaches to refinement verification were taken in [5] and [4], for their models.

Theorem 7 (*Testing*) *For processes p and q,*

$$p \sqsubseteq q \;\Leftrightarrow\; \forall\, r \in \mathcal{S}_{\mathcal{I}}\colon\; (\, r \times p \in \mathcal{R}_{\mathcal{I}} \;\Rightarrow\; r \times q \in \mathcal{R}_{\mathcal{I}}\,) \,. \tag{10}$$

Theorem 7 is another link between the absolute and relative notions of correctness. One can define refinement from a 'testing' point of view: p is refined by q if q passes any test that p passes. Passing a test r can be viewed as the absence of rejects when the device is composed with r. Theorem 7 shows that this testing definition of refinement is equivalent to the direct definition given in Section 2.

The testing paradigm is commonplace in concurrency theory (see e.g. [3]).

Theorem 8 (*Design*) *For processes p, q and r,*

$$p \sqsubseteq q \times r \;\Leftrightarrow\; p \oplus -q \sqsubseteq r \,. \tag{11}$$

The *design equation* is $p \sqsubseteq q \times r$, where process p represents a known specification, process q represents a known part of the implementation and process r represents the unknown remaining part of the implementation. Theorem 8 solves the design equation by showing that the weakest solution in the sense of refinement is $p \oplus -q$.

The design equation is a simplified version of the supervisory control theory that was introduced in [15]. Results on the design equation can be found in [20].

Theorem 9 (*RC Decomposition*) *For process p,*

(a) $p \sqcup \Phi$ is robust;
(b) $p \sqcap \Phi$ is chaotic;
(c) $p = (p \sqcap \Phi) \times (p \sqcup \Phi)$.

Recall that robust processes can be regarded as pure guarantees, and chaotic processes as pure requirements. Theorem 9 shows that every process is the product of a pure guarantee and a pure requirement, providing a way to compute the factors.

Proposition 10 $\mathcal{R}_\mathcal{I}$ *is closed under* $\times$, $\oplus$, $\sqcup$, *and* $\sqcap$.

Since the composition of two robust devices or environments is also robust, one can ensure the robustness of a system simply by using robust components.

Proposition 11 *In the lattice* $\langle S_\mathcal{I}, \sqcup, \sqcap \rangle$, $\mathcal{R}_\mathcal{I}$ *is the principal filter generated by* Φ.

Proposition 11 provides a characterization of robust processes as those processes that refine the void process, or processes that satisfy a 'block of wood' specification.

5 Relationships to Previous Work

Process spaces are related to several theories of concurrency that are based on traces, executions, or failures, including [3], [4], [5], [6], [7], [19], and [20] among others. All these formalisms have parallel composition operators that are comparable to our product, and notions of relative correctness that are comparable to our refinement. While we leave the proofs for further work, we roughly outline here the likely relationships to two widely used theories.

Process spaces	**CSP [6]**	**Dill [4]**
process	process	trace structure
accessible	trace or failure	possible
reject	divergence	failure
goal		success
product	parallel composition	parallel composition
refinement	refinement	inverse of conformation
robustness		receptivity
reflection		mirror

Although there is an agreement in essence between process spaces and the formalisms above, as well as among these previous formalisms, there are differences too.

For example, our product operation is idempotent, just like Dill's parallel composition of trace structures, whereas the parallel composition in CSP is not idempotent. We found idempotency to be suitable for our applications, as explained in Section 2.

A widely used formalism of concurrency that is not included in the list above is CCS [10]. A fundamental difference between process spaces and CCS, regardless of the execution domain, is that CCS relies on equivalence rather than a partial order (refinement) for comparing specifications to implementations. We use refinement for the sake of certain applications where specifications need to be free of implementation-bias. For instance, consider a specification for an abstract data type 'set' with operations 'insert an element' and 'extract an element'. This specification admits several implementations: stack, queue, heap, etc. Since these implementations are inequivalent, the specification cannot be equivalent to all of them. We could strengthen the specification by reducing the non-determinism of the 'extract' operation, but this would favor some of the possible implementations and rule out the others. On the other hand, it has been shown in [18] that the CCS operational semantics makes finer distinctions than certain trace and failure semantics of CCS agents.

Connections also exist between process spaces and several algebraic formalisms. In [2], we have shown that process spaces are ternary algebras. A difference, however, is that process spaces admit a ternary symmetry akin to the duality of Section 4 (see [13]), whereas no such ternary symmetry exists in general ternary algebra. A CONCUR reviewer pointed out that the process spaces product corresponds to the linear logic par, exclusive sum to tensor, and refinement to implication; under this interpretation, Theorem 8 and several other process space properties hold in linear logic. For an account of linear logic, see e.g. [17]. A difference, however, is that our product and exclusive sum have the same identity element.

6 Concluding Remarks

The process space product, refinement, and reflection, as well as some of their properties, are closely related to operators and relationships from other theories of concurrency, most notably to [6] and [4]. However, it appears that the key idea of abstracting the notion of execution has not been proposed before our technical report [11]. To eliminate the structure of executions, we have abandoned the notions of event, state, action, and port, as well as all the related restrictions.

It may be surprising that several meaningful properties of processes, such as the properties that enable modular and hierarchical verification, can be stated and proven without referring to the structure of executions. In particular, it may be surprising that full abstraction is a property of general process spaces, and does not rely on any assumptions regarding the level of detail of the model.

Some important concurrency operations that we have not discussed here are the derivative of a process with respect to a trace ('after'-operator), the hiding of a set of actions in a process, and the renaming of actions of a process. These operations and their properties are discussed in [13], Section 8.4, as instances of general Galois

connections between execution sets. However, we limit the scope of this paper to the basic operations introduced in Section 2.

Due to their generality, we found process spaces to be helpful in applications that involve unusual connectivity conditions or correctness concerns. Such applications have been demonstrated in [14] and [12], for the verification of asynchronous circuits. The fact that the process space formalism applies uniformly across several levels of abstractions permitted us to combine switch-level, gate-level, and relative timing assumptions into a single model, using the same operations and tool.

The flip side of generality is that, although our theory is the same for different correctness concerns in isolation, our product operator may not conserve properties of processes that combine several correctness concerns; arguments to that effect have been presented in [9]. In such situations, the targeted properties of processes can be restored after computing composition, by using domain-specific information about the structure of executions.

In this paper, process spaces are used to detect safety faults, liveness faults, shorts in electrical networks, and the presence of dangling inputs. Other potential applications are indicated in [13]: detection of deadlock and livelock, a treatment of progress, a new classification of liveness and progress faults, and a connection to dynamical systems. Also, process spaces can be applied where terminals can change direction (exploiting the absence of built-in distinctions between inputs and outputs in process spaces), and where simultaneous events are permitted [13]. Each of these applications inherits all the operators and algebraic properties of process spaces.

Further work should link process spaces to other models of concurrency by assigning processes as semantics to objects of the other models so that the main operations are preserved. A possible link to linear logic has been noticed by one of the CONCUR reviewers; other links exist to several trace-based or failure-based models (see Section 5). A lucrative direction for further work is to study process spaces over an execution set that includes causal and operational information, as in a Petri net. Process spaces are in a good position for building such links, due to abstract executions and the absence of restrictions. We hope such links can provide a basis for convergence of different points of view in concurrency theory.

Acknowledgements I am very indebted to J. A. Brzozowski, Robert T. Berks, David L. Dill, Jo C. Ebergen, Charles E. Molnar, Tom Verhoeff, and to the anonymous CONCUR reviewers for insightful critical reviews of this or related manuscripts, and for suggestions for further investigation.

References

1. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Prentice Hall (1990)
2. Brzozowski, J. A., Lou, J. J., Negulescu, R.: A characterization of finite ternary algebras. International Journal of Algebra and Computation 6 (1997) 713-721

3. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science (1983) 83-133
4. Dill, D.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. ACM distinguished dissertations. MIT Press (1989)
5. Ebergen, J. C.: A formal approach to designing delay-insensitive circuits. Distributed Computing 5 (1991) 107-119
6. Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall. (1985)
7. Josephs, M. B.: Receptive process theory. Acta Informatica 1 (1992) 17-31
8. Lamport, L., Lynch, N.: Distributed computing: models and methods. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics. The MIT Press - Elsevier (1990) 1159-1196
9. Mallon, W., Udding, J.T., Verhoeff, T.: Analysis and applications of the XDI model. In: Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (1999)
10. Milner, R.: Communication and Concurrency. Prentice Hall. (1989)
11. Negulescu, R.: Process spaces. Technical report CS-95-48, University of Waterloo, Canada (1995)
12. Negulescu, R.: Event-driven verification of switch-level correctness concerns. In: Int. Conf. on Application of Concurrency to System Design, Aizu-Wakamatsu, Japan (1998) 213-223
13. Negulescu, R.: Process Spaces and Formal Verification of Asynchronous Circuits. Ph.D. Thesis, University of Waterloo, Canada (1998)
`http://macs.ece.mcgill.ca/~radu/Papers/RN_thesis.ps.gz`
14. Negulescu, R., Peeters, A.: Verification of speed-dependences in single-rail handshake circuits. In: Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, IEEE Press (1998) 159-170
15. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete-event processes. SIAM J. Control and Optimization 1 (1987) 206-230
16. Staunstrup, J.: A Formal Approach to Hardware Design. Kluwer Academic Publishers (1994)
17. Troelstra, A.S.: Lectures on Linear Logic. Center for the Study of Language and Information, Leland Stanford Junior University (1992)
18. van Glaabeek, R.J.: The linear time - branching time spectrum. In: Baeten, J.C.M., Klop, J.W. (eds.): CONCUR '90. LNCS 458. Springer-Verlag (1990)
19. Valmari, A., Kokkarinen, I.: Unbounded verification results by finite-state compositional techniques: 10any states and beyond. In: Int. Conf. on Application of Concurrency to System Design, Aizu-Wakamatsu, Japan (1998) 75-85
20. Verhoeff, T.: A Theory of Delay-Insensitive Systems. Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (1994)

Failure Semantics for the Exchange of Information in Multi-Agent Systems

Frank S. de Boer, Rogier M. van Eijk,
Wiebe van der Hoek, and John-Jules Ch. Meyer

Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{frankb,rogier,wiebe,jj}@cs.uu.nl

Abstract. In this paper, we present a semantic theory for the exchange of information in multi-agent systems. We define a concurrent programming language for systems of agents that maintain their own private stores of information and that interact with each other by means of a synchronous communication mechanism that allows for the exchange of information. The semantics of the language, which is based on a generalisation of traditional failure semantics, is shown to be fully-abstract with respect to observing of each terminating computation its final global store of information.

1 Introduction

Multi-agent systems are the subject of a very active and rapidly growing research field in both artificial intelligence and computer science. Although there is no formal definition of an *agent* (in fact this also holds for the notion of an *object*, which nevertheless has proven to be a very successful concept for the design of a new generation of programming languages), generally speaking, one could say that a *multi-agent system* constitutes a system composed of several autonomous agents that operate in a (distributed) environment which they can perceive, reason about as well as can affect by performing actions [15]. In the current research on multi-agent systems, a major topic is the development of a standardised *agent communication language* for the exchange of information. Recently, several agent communication languages have been proposed in the literature, like for instance the language KQML [8]. However none of these communication languages have been given a fully formal account of their semantics [14]. The main contribution of this paper is the introduction of a formal semantic theory for the exchange of information in multi-agent systems.

1.1 Concurrent Programming

We introduce a concurrent programming language that concentrates on the information processing aspects of agents. The underlying computational model of the language has already been introduced in [5]. The basic operations of the

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 214–228, 2000.

language for the processing of information are the *ask* and *tell* operations of Concurrent Constraint Programming (CCP) [13]. This programming paradigm derives from traditional programming by replacing the *store-as-valuation* concept of von Neumann computing by the *store-as-constraint* model. This computational model is based on a global *store*, represented by a constraint, that expresses partial information on the values of the variables that are involved in computations. The different concurrently operating processes in CCP refine this partial information by adding (*telling*) new constraints to the store. Additionally, communication and synchronisation are achieved by allowing processes to test (*ask*) if the store entails a particular constraint before they proceed in their computation. These basic operations of asking and telling are defined in terms of the logical notions of conjunction and entailment, which are supported by a given underlying constraint system.

In our programming language, however, the global store of Concurrent Constraint Programming is *distributed* among the agents of the system. That is, the above described ask and tell operations of CCP are used by an agent to maintain its own *private* store of information. More precisely, these operations are performed by concurrently executing *threads* within the agent. The agent itself, however, has no direct access to the parts of the global store that are distributed among the other agents in the system. Instead, the agents can only obtain information from each other by means of a synchronous communication mechanism.

This communication mechanism is based on a generalisation of the communication scheme of (imperative) concurrent languages like Communicating Sequential Processes (CSP) [11], where the generalisation consists of the exchange of *information*, i.e. constraints, instead of the communication of simple *values*. Abstractly, communication between two agents comprises the supply of an *answer* of one agent to a posed *question* of another agent, and as such presents the basics of a *dialogue*. In particular, posing a question amounts to asking the other agent whether some information holds, while the answering agent in turn provides information from its own private constraint store that is logically strong enough to entail the question. In general, our programming language thus can be viewed upon as a particular model of the concept of *distributed knowledge* as introduced in [9]. The above described communication mechanism then provides a way in which the distributed knowledge of a multi-agent system can become shared among the agents.

1.2 Fully Abstract Semantics

The main result of this paper is a compositional semantics for the multi-agent language that is *fully abstract* with respect to observing the final (global) stores of terminating computations. This semantics is based on a generalisation of the *failure* semantics as developed for CSP, in which failure sets are employed to give a semantic account of (possible) *deadlock* behaviour [4]. However, whereas in CSP a failure set is simply given by a subset of the complement of all the *initial* actions of a process, in our framework, these failure sets are defined in terms of

the information that is *logically* irrelevant to the specific question or answer of the agent. Moreover, for CSP-like languages the failure sets can without loss of generality be assumed to be *finite* [2]. This assumption, which plays an essential role in the full abstractness proof, fails in the context of the exchange of information. However, we show that our notion of failure semantics, which includes *infinite* failure sets, satisfies a *compactness property* that roughly amounts to the following: if every *finite* subset of a given set of answers or questions is logically irrelevant (with respect to a particular question/answer of a given agent) then this entire set is irrelevant.

1.3 Related Work

To the best of our knowledge, this paper presents a first formal semantic account of the exchange of information in multi-agent systems. This semantics, we believe, provides a general basis for the semantics of agent communication languages in general as introduced in artificial intelligence, like for instance the language KQML [8].

Other approaches that relate to our programming language include the work of Réty on distributed concurrent constraint programming [12]. One of the differences with our approach is that in this framework distributed processes do not share any variables. In particular, communication between processes proceeds by means of a form of constraint abstraction: during the exchange of a constraint, the variables of the sender that occur in the constraint are replaced by the variables of the receiving processes.

Additionally, there is the research on synchronous concurrent constraint programming, which is a version of CCP that in addition to the standard ask and tell operations, covers a synchronous communication mechanism in which a constraint is told to the constraint store only if there is another process asking for it [3]. The main difference with our approach is that in this framework both the synchronous and asynchronous form of communication proceed via a global constraint store.

1.4 Overview

The remainder of this paper is organised as follows. In Section 2, we give the syntax of our multi-agent programming language, which combines the language of concurrent constraint programming with synchronous communication primitives for the exchange of information. The structural operational semantics of this language is subsequently defined in Section 3 in terms of a local and global transition system. Additionally, in Section 4, we define a failure semantics which is shown to be fully abstract with respect to observing the final information store of terminating computations. Finally, in Section 5 we round off by suggesting several directions for future research.

2 Syntax

In this section, we introduce the syntax of our agent language, which like Concurrent Constraint Programming is parameterised by a constraint system that is used to represent information.

Definition 1 A constraint system $\mathbf{C}$ is a tuple $(C, \sqsubseteq, \sqcup, \mathit{true}, \mathit{false})$, where C (the set of constraints, with typical element φ) is a set ordered with respect to $\sqsubseteq$, $\sqcup$ is the least upperbound operation, and *true*, *false* are the least and greatest elements of C, respectively.

The interpretation of $\varphi \sqsubseteq \psi$ is that φ contains less information than ψ, while $\varphi \sqcup \psi$ denotes the conjunction of φ and ψ.

In order to model *hiding* of local variables and *parameter passing* in constraint programming, in [13] the notion of constraint system is enriched with *cylindrification operators* and *diagonal elements*, which are concepts borrowed from the theory of cylindric algebras [10].

Definition 2 Given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$, we introduce a family of operators $\{\exists_x \mid x \in \mathit{Var}\}$ (cylindrification operators) and of constants $\{d_{xy} \mid x, y \in \mathit{Var}\}$ (diagonal elements).

Starting from a constraint system $\mathbf{C}$, we define a cylindric constraint system $\mathbf{C}'$ as the constraint system whose support set C' is the smallest such that

$$C' = C \cup \{\exists_x \varphi \mid x \in \mathit{Var},\ \varphi \in C'\} \cup \{d_{xy} \mid x, y \in \mathit{Var}\}$$

modulo the identities and with the additional relations derived by the following axioms ($\exists_x \varphi \sqcup \psi$ stands for $(\exists_x \varphi) \sqcup \psi$):

A1. $\exists_x \varphi \sqsubseteq \varphi$,
A2. if $\varphi \sqsubseteq \psi$ then $\exists_x \varphi \sqsubseteq \exists_x \psi$,
A3. $\exists_x(\varphi \sqcup \exists_x \psi) = \exists_x \varphi \sqcup \exists_x \psi$,
A4. $\exists_x \exists_y \varphi = \exists_y \exists_x \varphi$,
A5. $d_{xx} = \mathit{true}$,
A6. if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
A7. if $x \neq y$ then $\varphi \sqsubseteq d_{xy} \sqcup \exists_x(\varphi \sqcup d_{xy})$.

The above laws give to $\exists_x$ the flavour of a first-order *existential quantifier*, as the notation also suggests. The constraint d_{xy} can be interpreted as the equality between x and y. Cylindrification and diagonal elements allow us to model the variable renaming of a formula φ; in fact, by the above axioms, the formula $\exists_x(d_{xy} \sqcup \varphi)$ can be interpreted as the formula $\varphi[y/x]$, namely the formula obtained from φ by replacing all the free occurrences of x by y.

In the following definition, we assume a given set *Chan* of communication channels, with typical element c, and a set *Proc* of procedure identifiers, with typical element p.

Definition 3 *(Basic actions)* Given a cylindrical constraint system $\mathbf{C}$ the basic actions of the programming language are defined as follows:

$$a ::= c!\varphi \mid c?\varphi \mid \mathsf{ask}(\varphi) \mid \mathsf{tell}(\varphi).$$

The execution of the output action $c!\varphi$ consists of sending the information φ along the channel c, which has to synchronise with a corresponding input $c?\psi$, for some ψ with $\psi \sqsubseteq \varphi$. In other words, the information φ can be sent along a channel c only if some information entailed by φ is requested. The execution of an input action $c?\psi$, which consists of receiving the information φ along the channel c, also has to synchronise with a corresponding output $c!\varphi$, for some φ with $\psi \sqsubseteq \varphi$. The execution of a basic action $\mathsf{ask}(\varphi)$ by an agent consists of checking whether the private store of the agent entails φ. On the other hand, the execution of $\mathsf{tell}(\varphi)$ consist of adding φ to the private store.

Definition 4 *(Statements)* The behaviour of an agent is then described by a statement S:

$$S ::= a \cdot S \mid S_1 + S_2 \mid S_1 \;\&\; S_2 \mid \exists_x S \mid p(\bar{x}).$$

Statements are thus built up from the basic actions using the following standard programming constructs: action prefixing, which is denoted by $\cdot$; non-deterministic choice, denoted by $+$; internal parallelism, denoted by $\&$; local variables, denoted by $\exists_x S$, which indicates that x is a local variable in S; and (recursive) procedure calls of the form $p(\bar{x})$, where $\bar{x}$ denotes a sequence of variables which constitute the actual parameters of the call.

Definition 5 *(Multi-agent systems)* A multi-agent system A is defined by

$$A ::= \langle D, S, \varphi\rangle \mid A_1 \parallel A_2 \mid \delta_H(A).$$

A basic agent in a multi-agent system is represented by a tuple $\langle D, S, \varphi\rangle$ consisting first of all of a set D of procedure declarations of the form $p(\bar{x}) :\!-\; S$, where $\bar{x}$ denote the formal parameters of p and S denotes its body. The statement S in $\langle D, S, \varphi\rangle$ describes the behaviour of the agent with respect to its *private* store φ. The threads of S, i.e. the concurrently executing substatements of S, interact with each other via the private store of the basic agent by means of the actions $\mathsf{ask}(\psi)$ and $\mathsf{tell}(\psi)$. As in the operational semantics below the set D of procedure declarations will not change, we usually omit it from notation and simply write $\langle S, \varphi\rangle$ instead of $\langle D, S, \varphi\rangle$.

Additionally, a multi-agent system itself consists of a collection of concurrently operating agents that interact with each other only via a synchronous information-passing mechanism by means of the communication actions $c!\psi$ and $c?\psi$.

For technical convenience only we restrict to the parallel composition of agent systems: the semantic treatment of the sequential composition of multi-agent systems and the non-deterministic choice between agent systems is standard.

Finally, the encapsulation operator δ_H with $H \subseteq \mathit{Chan}$, which stems from the process algebra ACP, is used to define local communication channels [1]. That is, $\delta_H(A)$ denotes a multi-agent system in which the communication channels in H are local and hence, cannot be used for communication with agents outside the system.

3 Operational Semantics

The structural operational semantics of the programming language is defined by means of a *local* and a *global* transition system. Given a set of declarations D, a local transition is of the form

$$\langle S, \varphi \rangle \xrightarrow{\ l\ } \langle S', \psi \rangle$$

where either l equals τ in case of an *internal* computation step, that is, a computation step which consists of the execution of a basic action of the form $\mathsf{ask}(\varphi)$ or $\mathsf{tell}(\varphi)$, or l is of the form $c!\varphi$ or $c?\varphi$, in case of a communication step. We employ the symbol E to denote successful termination.

Definition 6 *(The local transition system)* We have the following transitions for the basic actions.

- $\langle c!\varphi, \psi \rangle \xrightarrow{c!\varphi} \langle E, \psi \rangle$ if $\varphi \sqsubseteq \psi$
- $\langle c?\varphi, \psi \rangle \xrightarrow{c?\varphi} \langle E, \psi \sqcup \varphi \rangle$
- $\langle \mathsf{ask}(\varphi), \psi \rangle \xrightarrow{\tau} \langle E, \psi \rangle$ if $\varphi \sqsubseteq \psi$
- $\langle \mathsf{tell}(\varphi), \psi \rangle \xrightarrow{\tau} \langle E, \psi \sqcup \varphi \rangle$

An output action $c!\varphi$ can only take place in case the information φ to be communicated is entailed by the private store ψ. In other words, the agents are assumed to be *truthful*. On the other hand, the information φ received by an input action $c?\varphi$ is added to the private store. It is worthwhile to remark here that alternatively we could have defined input actions semantically by $\langle c?\varphi, \psi \rangle \xrightarrow{c?\varphi} \langle E, \psi \rangle$. Whether or not the private store is correspondingly updated can be controlled by the agent itself by means of the action $\mathsf{tell}(\varphi)$. However, for technical convenience only we have adopted in this paper the first approach.

The actions $\mathsf{ask}(\varphi)$ and $\mathsf{tell}(\varphi)$ are the familiar operations from CCP which allow an agent to inspect and update its private store.

Furthermore, we have the usual rules for action prefixing, recursive procedure calls and the programming constructs for non-deterministic choice and parallel composition. Here we list only the following generalisation of the rule for local variables as defined in [5].

$$\frac{\langle S, \exists_x \psi \sqcup \varphi \rangle \xrightarrow{\ l\ } \langle S', \psi' \rangle}{\langle \exists_x^{\varphi} S, \psi \rangle \xrightarrow{\ l'\ } \langle \exists_x^{\psi'} S', \psi \sqcup \exists_x \psi' \rangle}$$

where the label l' equals τ in case $l = \tau$, and $l' = c!\exists_x\varphi$ ($l' = c?\exists_x\varphi$) in case $l = c!\varphi$ ($l = c?\varphi$), for all c and φ.

The syntax of the language is thus extended with a construct of the form $\exists_x^{\varphi} S$ denoting that in the statement S the variable x is a local variable, where the constraint φ collects the information on the local variable x. In this notation, the statement $\exists_x S$ is written as $\exists_x^{true} S$, denoting that the local constraints on x are initially empty.

Thus, in our framework we have a *global* store that is distributed over the agents, which is formally defined below. Each agent has direct access only to its *private* store. Information in the private store about the *global* variables can be communicated to the other agents. The *local* variables of an agent, which are introduced by the hiding operator $\exists_x$, however, cannot be referred to in communications. This explains why in the context of $\exists_x^{\varphi} S$ the constraints on the local variable x in a communication are existentially quantified.

A *global* transition is of the form $A \stackrel{l}{\longrightarrow} A'$, where l indicates whether the transition involves an internal computation step, that is, $l = \tau$, or a communication, that is, $l = c!\varphi$ or $l = c?\varphi$.

Definition 7 *(Transitions for multi-agent systems)*
The following rule describes parallel composition by interleaving of the basic actions:

$$\frac{A_1 \stackrel{l}{\longrightarrow} A_1'}{\begin{array}{c} A_1 \parallel A_2 \stackrel{l}{\longrightarrow} A_1' \parallel A_2 \\ A_2 \parallel A_1 \stackrel{l}{\longrightarrow} A_2 \parallel A_1' \end{array}}$$

In order to describe the synchronisation between agents we introduce a synchronisation predicate $|$, which is defined as follows. For all $c \in \mathit{Chan}$ and $\varphi, \psi \in \mathcal{L}$, if $\psi \sqsubseteq \varphi$ then

$$(c!\varphi \mid c?\psi) \text{ and } (c?\psi \mid c!\varphi).$$

In all other cases, the predicate $|$ yields the boolean false. We then have the following synchronisation rule:

$$\frac{A_1 \stackrel{l_1}{\longrightarrow} A_1' \quad A_2 \stackrel{l_2}{\longrightarrow} A_2'}{A_1 \parallel A_2 \stackrel{\tau}{\longrightarrow} A_1' \parallel A_2'} \text{ if } l_1 \mid l_2$$

This rule shows that an action of the form $c?\psi$ only matches with an action of the form $c!\varphi$ in case ψ is entailed by φ. In all other cases, the predicate $|$ yields false and therefore no communication can take place.

Finally, *encapsulation* of communications along a set of channels H is described by the rule:

$$\frac{A \stackrel{l}{\longrightarrow} A'}{\delta_H(A) \stackrel{l}{\longrightarrow} \delta_H(A')} \text{ if } \mathit{chan}(l) \cap H = \emptyset$$

where chan is defined by $\mathit{chan}(c!\varphi) = \mathit{chan}(c?\varphi) = \{c\}$ and $\mathit{chan}(\tau) = \emptyset$.

For any multi-agent system A we use the notation $\mathit{store}(A)$ to denote its constraint store.

Definition 8 *(Global store)*
We define $store(A)$ by induction on the structure of the agent system A:

$$\begin{aligned} store(\langle D, S, \varphi \rangle) &= \varphi \\ store(A_1 \parallel A_2) &= store(A_1) \sqcup store(A_2) \\ store(\delta_H(A)) &= store(A) \end{aligned}$$

For any multi-agent system A, $store(A)$ thus denotes the *global* constraint store that is distributed among its (sub-)agents. In fact, this amounts to what is known as *distributed knowledge* in the research on distributed systems, referring to the knowledge that would result if the knowledge of all agents in a distributed system is taken together [9].

We want to observe the behaviour of a multi-agent system when it runs on its own, that is, as a *closed* system without interaction with an environment. In the definition below the *observable* behaviour of a multi-agent system is therefore defined as the set of final stores of *terminating* computations that consist of internal computation steps only. Moreover, the existence of an internally diverging computation or the generation of an inconsistent global store will be considered as a fatal *error* and as such they will give rise to *chaos*, which amounts to a situation in which simply anything can be observed.

Definition 9 *(Operational Semantics)*
In defining the operational semantics we make use of several auxiliary notions:

- The notation $A \Longrightarrow B$ indicates the existence of a terminating computation that consists of internal computation steps only: $A \xrightarrow{\tau} \cdots \xrightarrow{\tau} B$. Additionally, $B \stackrel{\tau}{\not\longrightarrow}$ indicates that starting from B there is no further τ-transition possible; that is, B is either terminated or represents a *deadlock* situation.
- Furthermore, the construct $A \Longrightarrow \Omega$ indicates the existence of an infinite computation that consists of internal computation steps only: $A \xrightarrow{\tau} \cdots \xrightarrow{\tau} A_i \xrightarrow{\tau} \cdots$. Such computations are called *divergent*.

Then we define:

$$\begin{aligned} &\mathcal{O}(A) = \\ &\{store(B) \mid A \Longrightarrow B \stackrel{\tau}{\not\longrightarrow}\} \ \cup \\ &\{\varphi \mid \varphi \in \mathcal{C},\ A \Longrightarrow \Omega \text{ or } A \Longrightarrow B, \text{ with } store(B) = false\} \end{aligned}$$

Note that the treatment of (internally) diverging computations and inconsistent stores can be mathematically justified in terms of the following *recursive* definition of $\mathcal{O}$:

$$\begin{aligned} &\mathcal{O}(A) = \\ &\{store(A) \mid A \stackrel{\tau}{\not\longrightarrow}\} \ \cup \\ &\{\varphi \mid \varphi \in \mathcal{O}(B), \text{ with } A \xrightarrow{\tau} B, \text{ or } store(A) = false\} \end{aligned}$$

The above non-recursive definition then can be shown to correspond with the *greatest fixpoint* of this recursive equation.

We observe that internal divergent computations may be *unfair*: If A gives rise to an internally divergent computation so will $A \parallel B$, for *any* B. However, the results of this paper can be easily extended to a semantics of the parallel composition operator which is *weakly* fair in the following sense: every parallel agent which is enabled will eventually be executed.

4 Failure Semantics

In order to obtain a compositional and fully abstract characterisation of the above described notion of observables, we introduce a failure semantics that records for each sequence of communications of an agent system the corresponding failure set and the corresponding private store.

Definition 10 *(Failure semantics)*
First, we introduce the following notions:

- The transition relation $A \stackrel{w}{\Longrightarrow} B$, where w is a sequence of communication actions, indicates that the agent system A can evolve itself into B by executing a sequence of actions w' such that w can be obtained from w' by deleting all τ-moves.
- We let I be the function that associates with each multi-agent system A the collection of initial actions it can perform, which is defined as follows:

$$I(A) = \{l \mid A \stackrel{l}{\longrightarrow} B, \text{ for some } B\}.$$

- We denote by $\overline{X}$, with X a set of communication actions, the set of communication actions $c?\varphi$ ($c!\varphi$) such that there does not exist a complementary $c!\psi \in X$ ($c?\psi \in X$) with $\varphi \sqsubseteq \psi$ ($\psi \sqsubseteq \varphi$).

The failure semantics $\mathcal{F}$ is then defined as follows:

$$\begin{array}{l}\mathcal{F}(A) = \\ \{\langle w, (store(B), F)\rangle \mid A \stackrel{w}{\Longrightarrow} B,\ \tau \notin I(B),\ F \subseteq \overline{I(B)}\} \cup \\ \{\langle w \cdot w', \bot\rangle \mid A \stackrel{w}{\Longrightarrow} B, \text{ with } B \Longrightarrow \Omega \text{ or } store(B) = false\}.\end{array}$$

Note that w' ranges over all sequences of communication actions. Alternatively, we have the following *recursive* definition of $\mathcal{F}$:

$$\begin{array}{l}\mathcal{F}(A) = \\ \{\langle \epsilon, (store(A), F)\rangle \mid \tau \notin I(A),\ F \subseteq \overline{I(A)}\} \cup \\ \{\langle l \cdot w, t\rangle \mid A \stackrel{l}{\longrightarrow} B, \text{ with } \langle w, t\rangle \in \mathcal{F}(B) \text{ or } store(B) = false\}\end{array}$$

where l denotes a communication action. The above non-recursive definition then can be shown to correspond with the *greatest fixpoint* of this recursive equation.

The above failure semantics associates with each multi-agent system A a set of so called *failure traces*, which record the sequence of communication actions

that are generated by a computation of A, the corresponding resulting (global) store and a failure set of communication actions that are *refused*. Divergence and inconsistency are represented by the symbol $\bot$, denoting a situation of chaos in which everything is possible.

This definition of the failure sets differs from the usual one which is simply defined as a subset of the *complement* of the set of initial actions. The standard failure sets as such indicate the actions which the process itself cannot perform. However, a communication action which an agent cannot perform is not necessarily an action which it refuses for synchronisation, as shown in the following example.

Example 11 Consider the agents

$$A = \langle c!(\varphi \sqcup \psi), \varphi \sqcup \psi \rangle \text{ and } B = \langle c!(\varphi \sqcup \psi) + c!\varphi, \varphi \sqcup \psi \rangle.$$

The set of initial actions of A and B are clearly different. However, it is easy to see that $\overline{X} = \overline{Y}$, for $X = \{c!(\varphi \sqcup \psi)\}$ and $Y = \{c!(\varphi \sqcup \psi), c!\varphi\}$. In fact, in general we have that $\overline{\overline{X}}$, for any set of communication actions X, consists of exactly those communication actions $c?\varphi$ ($c!\varphi$) such that $c?\psi \in X$ ($c!\psi \in X$), with $\psi \sqsubseteq \varphi$ ($\varphi \sqsubseteq \psi$). That is, $\overline{\overline{X}}$ contains all the communication actions which derive from X by *asking more* or *telling less*.

Of particular interest is to observe here that the failure sets can be *infinite* (but denumerable) sets of communication actions (that is, when the underlying constraint system contains infinitely but denumerable many constraints).

The following theorem states the correctness of the above failure semantics.

Theorem 12 *(Correctness of $\mathcal{F}$)*

$$\varphi \in \mathcal{O}(A) \text{ iff } \langle \epsilon, (\varphi, F) \rangle \in \mathcal{F}(A), \text{ for some } F, \text{ or } \langle \epsilon, \bot \rangle \in \mathcal{F}(A)$$

Moreover, we have the following compositionality result.

Theorem 13 *(Compositionality of $\mathcal{F}$)*
In order to define the semantics of the parallel composition of agents we first introduce the parallel composition of sequences of communication actions. Given such sequences w_1 and w_2 we define $w_1 \parallel w_2$ as the following set of communication sequences [1]. First, we define $\epsilon \parallel \epsilon$ to be $\{\epsilon\}$, and for the other cases:

- $w_1 \parallel w_2 = (w_1 \mathbin{\lfloor\!\lfloor} w_2) \cup (w_2 \mathbin{\lfloor\!\lfloor} w_1) \cup (w_1 \mid w_2)$, where the leftmerge operator $\lfloor\!\lfloor$ and the synchronisation merge $\mid$ are (recursively) defined by:
- $(a \cdot w) \mathbin{\lfloor\!\lfloor} w' = a \cdot (w \parallel w')$ and
- $(c?\varphi \cdot w) \mid (c!\psi \cdot w') = (w \parallel w')$ provided that $\varphi \sqsubseteq \psi$. In all other cases we have: $w_1 \mid w_2 = \emptyset$.

Additionally, let $X \mid Y$, where X and Y are sets of communication actions, indicate that there exists $c?\varphi \in X$ and $c!\psi \in Y$ (or vice versa) such that $\varphi \sqsubseteq \psi$.

Finally, for the compositional modelling of chaos introduced by internally diverging computations and inconsistent constraint stores, we need the following notions.

- $\mathcal{F}^\omega(A)$ denotes all the *infinite* sequences w of communication actions such that for every prefix w' of w, we have $\langle w', t\rangle \in \mathcal{F}(A)$, for some t
- the predicate $v_1 \uparrow v_2$ satisfies:

$$(c!\psi \cdot v) \uparrow (c?\varphi \cdot v') = (\varphi \sqsubseteq \psi \text{ and } v \uparrow v')$$

- Finally, the composition of termination modes is defined as follows:
  - Provided that $\overline{F_1} \not| \overline{F_2}$ and $\varphi_1 \sqcup \varphi_2 \neq \mathit{false}$ we define:
    $(\varphi_1, F_1) \parallel (\varphi_2, F_2) = \{(\varphi_1 \sqcup \varphi_2, F) \mid F \subseteq F_1 \cap F_2\}$
  - Provided that $\varphi_1 \sqcup \varphi_2 = \mathit{false}$ we define:
    $(\varphi_1, F_1) \parallel (\varphi_2, F_2) = \{\bot\}$
  - $(\varphi, F) \parallel \bot = \{\bot\}$ and $\bot \parallel (\varphi, F) = \{\bot\}$

Note that the first two equations for the composition of termination modes are partially defined: in the other cases the result is the empty set. Additionally, note that in the first equation we make explicit use of *infinite* failure sets since, given an infinite underlying constraint system, for any *finite* failure sets F_1 and F_2 we have that $\overline{F_1} \mid \overline{F_2}$: choose a constraint φ such that neither $\varphi \sqsubseteq \psi$ nor $\psi \sqsubseteq \varphi$, for any constraint ψ occurring in F_1 or F_2. Then we have for any channel c that, for example, $c!\varphi \in \overline{F_1}$ and $c?\varphi \in \overline{F_2}$. We then have:

$$\begin{array}{l}\mathcal{F}(A_1 \parallel A_2) = \\ \{\langle w, t\rangle \mid w \in (w_1 \parallel w_2),\ t \in (t_1 \parallel t_2),\ \langle w_i, t_i\rangle \in \mathcal{F}(A_i),\ i = 1, 2\} \cup \\ \{\langle w \cdot w', \bot\rangle \mid w \in (w_1 \parallel w_2) \mid (w_1 \cdot u) \in \mathcal{F}^\omega(A_1),\ (w_2 \cdot v) \in \mathcal{F}^\omega(A_2) \text{ and } u \uparrow v\}\end{array}$$

and additionally:

$$\begin{array}{l}\mathcal{F}(\delta_H(A)) = \\ \{\langle w, (\varphi, F)\rangle \mid \langle w, (\varphi, F')\rangle \in \mathcal{F}(A), \\ \mathit{chan}(w) \cap H = \emptyset \text{ and } F \subseteq F' \cup \{c!\psi, c?\psi \mid c \in H\}\} \cup \\ \{\langle w, \bot\rangle \mid \langle w, \bot\rangle \in \mathcal{F}(A),\ \mathit{chan}(w) \cap H = \emptyset\}\end{array}$$

Divergence of the parallel composition $A_1 \parallel A_2$ thus stems from the fact that A_1 or A_2 *themselves* diverge, or that each of them generates an infinite sequence of communication actions such that each individual communication action in one sequence matches with the communication action in the other sequence. The failure semantics $\mathcal{F}$ however still distinguishes too many agent systems, that is, it is not fully abstract with respect to the observables $\mathcal{O}$, as shown by the following example.

Example 14 Consider again the agents

$$A = \langle c!(\varphi \sqcup \psi), \varphi \sqcup \psi\rangle \text{ and } B = \langle c!(\varphi \sqcup \psi) + c!\varphi, \varphi \sqcup \psi\rangle.$$

The failure semantics distinguishes these two agents as we have:

$$\langle c!\varphi, (\varphi \sqcup \psi, F)\rangle \in \mathcal{F}(B) - \mathcal{F}(A),$$

for all F. However, intuitively, there is no context $C[\cdot]$ such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$. (Formally, this can be shown via the correctness and compositionality of the operator $\mathcal{F}_\alpha$ that is defined below.)

From this example we conclude that in order to obtain a fully abstract semantics we should take account of the fact that the communication of a constraint includes the communication of all weaker constraints and that reception of a constraint includes the reception of all stronger constraints. We will introduce an abstraction of $\mathcal{F}$ that incorporates these properties of asking more and telling less.

Definition 15 *(Abstraction operator)*
For every set W of traces we denote by $\alpha(W)$ the smallest set V that contains W and additionally satisfies:

- $w_1 \cdot c?\varphi \cdot w_2 \in V \Rightarrow w_1 \cdot c?\psi \cdot w_2 \in V$, provided that $\varphi \sqsubseteq \psi$
- $w_1 \cdot c!\varphi \cdot w_2 \in V \Rightarrow w_1 \cdot c!\psi \cdot w_2 \in V$, provided that $\psi \sqsubseteq \varphi$.

Definition 16 *(The semantics $\mathcal{F}_\alpha$)*
We define the semantics $\mathcal{F}_\alpha$ as follows:

$$\mathcal{F}_\alpha(A) = \alpha(\mathcal{F}(A))$$

where $\alpha(\mathcal{F}(A))$ denotes the obvious extension of α to sets of failure traces.

The semantics $\mathcal{F}_\alpha$ are correct and compositional with respect to the observable $\mathcal{O}$. The compositionality is established in the following theorem.

Theorem 17 *(Compositionality of $\mathcal{F}_\alpha$)*
We have:

$$\begin{array}{l}\mathcal{F}_\alpha(A_1 \parallel A_2) = \\ \{\langle w, t\rangle \mid w \in (w_1 \parallel w_2),\ t \in (t_1 \parallel t_2),\ \langle w_i, t_i\rangle \in \mathcal{F}_\alpha(A_i),\ i = 1, 2\} \cup \\ \{\langle w \cdot w', \bot\rangle \mid w \in (w_1 \parallel w_2) \mid (w_1 \cdot u) \in \mathcal{F}^\omega_\alpha(A_1),\ (w_2 \cdot v) \in \mathcal{F}^\omega_\alpha(A_2) \text{ and } u \uparrow v\}\end{array}$$

where $\mathcal{F}^\omega_\alpha(A)$ consists of all the *infinite* sequences w of communication actions such that for every prefix w' of w, we have $\langle w', t\rangle \in \mathcal{F}_\alpha(A)$, for some t. Additionally, we have

$$\begin{array}{l}\mathcal{F}_\alpha(\delta_H(A)) = \\ \{\langle w, (\varphi, F)\rangle \mid \langle w, (\varphi, F')\rangle \in \mathcal{F}_\alpha(A), \\ \mathit{chan}(w) \cap H = \emptyset \text{ and } F \subseteq F' \cup \{c!\psi, c?\psi \mid c \in H\}\} \cup \\ \{\langle w, \bot\rangle \mid \langle w, \bot\rangle \in \mathcal{F}_\alpha(A),\ \mathit{chan}(w) \cap H = \emptyset\}\end{array}$$

Note that thus the abstraction operator α simply distributes over the semantic counterparts of the operators of parallel composition and encapsulation.

In order to prove full abstractness of the semantics $\mathcal{F}_\alpha$ we need the following *compactness* property.

Theorem 18 *(Compactness of the failure semantics $\mathcal{F}_\alpha$)*
We have that if $\langle w, (\varphi, F')\rangle \in \mathcal{F}_\alpha(A)$, for every *finite* subset F' of a given set of communication actions F, then also $\langle w, (\varphi, F)\rangle \in \mathcal{F}_\alpha(A)$.

Proof. Suppose that for all finite $F' \subseteq F$ we have $\langle w, (\varphi, F')\rangle \in \mathcal{F}_\alpha(A)$. We assume that F is infinite, as the finite case is trivial. Let $l_1, l_2, \ldots$ be an enumeration of the elements of F. Consider the following collection $\mathbf{F}$ of subsets of F:

$$\begin{aligned} F_0 &= \emptyset \\ F_{j+1} &= F_j \cup \{l_j\} \end{aligned}$$

Consider next the collection C of computations of A that generate the word w, yield the store φ and refuse a set $F_i \in \mathbf{F}$. We assume that there is a bound k on the number of successive τ-steps that can occur in each computation in C. For, if such a bound does not exist then from the fact that the language gives rise to only finitely branching computation trees, we conclude via Königs Lemma, which states that any finitely branching tree with an infinite number of nodes has an infinite path, that there must be a computation in C that after having generated a prefix of w goes into an infinite loop of τ-steps. Then since such a diverging computation gives rise to chaos, we immediately obtain $\langle w, (\varphi, F)\rangle \in \mathcal{F}_\alpha(A)$.

Hence, we assume that such a bound k exists. As the computation tree is finitely branching, there also exists a bound on the length of the computations in C, and therefor C is finite. However as there are infinitely many sets $F_i \in \mathbf{F}$, the pigeon-hole principle then tells us that there must be a computation p in C that refuses an infinite number of finite sets in $\mathbf{F}$. We claim that p also refuses F.

Consider an arbitrary element l_i of F. As $l_i \in F_{i+1}$ and p refuses an infinite number of failure sets of $\mathbf{F}$, there must be an index $j \geq i+1$ such that p refuses F_j. As this set F_j also includes the action l_i, we obtain that the computation p refuses l_i. As l_i was chosen arbitrarily from F, we conclude that p refuses all elements of F (or more). By the definition of the failure semantics which says that any subset of a failure set is also a failure set, we obtain $\langle w, (\varphi, F)\rangle \in \mathcal{F}_\alpha(A)$.

Next, we show the full abstractness for $\mathcal{F}_\alpha$.

Theorem 19 *(Full abstractness for $\mathcal{F}_\alpha$)*
For any two agents A and B:

$$\mathcal{F}_\alpha(A) = \mathcal{F}_\alpha(B) \Leftrightarrow \mathcal{O}(C[A]) = \mathcal{O}(C[B]), \quad \text{for any context } C[\cdot]$$

Proof. The implication from left to right follows from the correctness of $\mathcal{F}_\alpha$ and its compositionality. For the reverse implication we proceed as follows. Suppose $\mathcal{F}_\alpha(A) \neq \mathcal{F}_\alpha(B)$ then w.l.o.g. there must exist $\langle w, \bot\rangle$ or $\langle w, (\varphi, F)\rangle \in \mathcal{F}_\alpha(A) - \mathcal{F}_\alpha(B)$, for some *finite* set F (according to the compactness property established in Theorem 18).

The idea is then to define a context $C[\cdot]$ such that there exists a constraint ψ with $\psi \in \mathcal{O}(C[A])$ and $\psi \notin \mathcal{O}(C[B])$. In order to achieve this, we define the complement $\widetilde{l}$ of a communication action l as follows: $\widetilde{c!\varphi} = c?\varphi$ and $\widetilde{c?\varphi} = (\mathsf{tell}(\varphi) \cdot c!\varphi)$, for all c and φ. Let w be given by $l_1 l_2 \cdots l_n$.

First, we consider the case $\langle w, \bot\rangle$. Consider the context $C[\cdot]$ that is defined by:

$$\langle \widetilde{l_1} \cdots \widetilde{l_n}, \mathit{true}\rangle \parallel \cdot$$

It is easy to see that $\mathcal{O}(C[A])$ contains all constraints. We claim that this does not hold for $\mathcal{O}(C[B])$. For, suppose $\mathcal{O}(C[B])$ contains all constraints then there are two possibilities. It could be the case that $B \stackrel{u}{\Longrightarrow} B' \stackrel{\tau}{\Longrightarrow} \Omega$, for some prefix u of w (modulo asking more and telling less) or it could be the case that $B \stackrel{u}{\Longrightarrow} B'$ with $store(B') = false$ for some prefix u of w. Then by the definition of $\mathcal{F}_\alpha$ in which divergence and inconsistency gives rise to chaos, we conclude $\langle u, \bot\rangle \in \mathcal{F}_\alpha(A)$ and hence, as u is a prefix of w we have: $\langle w, \bot\rangle \in \mathcal{F}_\alpha(A)$. This yields a contradiction.

Next, we consider the case $\langle w, (\varphi, F)\rangle$ (where $\varphi \neq false$). Consider the following context $C[\cdot]$:

$$\langle \widetilde{l_1} \cdots \widetilde{l_n} \cdot \mathsf{tell}(ok_1) \cdot \Sigma_{l \in F}(l \cdot \mathsf{tell}(ok_2)), true\rangle \parallel \cdot$$

where ok_1 and ok_2 denote constraints that do not occur in the multi-agent systems A and B and $\Sigma_{l\in F}$ denotes the non-deterministic choice between the actions in F. The idea of this context is that it offers the complements of the actions of w then produces a signal ok_1, and finally, offers the actions in the failure set F. Additionally, it produces the signal ok_2 in case one of the elements in F is accepted. It is easy to see that $(\varphi \sqcup ok_1) \in \mathcal{O}(C[A])$.

We claim that $(\varphi \sqcup ok_1) \notin \mathcal{O}(C[B])$. For, otherwise it would be the case that $B \stackrel{u}{\Longrightarrow} B'$ with $u = w$ modulo asking more and telling less (as the signal ok_1 has been produced) and B' refuses the set F or more (otherwise the signal ok_2 would have been produced). Then via the definition of $\mathcal{F}_\alpha$ which says that any subset of a failure set is also a failure set, we conclude $\langle w, (\varphi, F)\rangle \in \mathcal{F}_\alpha(B)$. This yields a contradiction and hence we obtain $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$.

5 Conclusions and Future Research

In this paper, we have outlined a concurrent programming language for multi-agent systems that concentrates on the information-processing aspects of agents. The language is given a compositional semantics, based on a generalisation of traditional failure semantics, which is shown to be fully-abstract with respect to observing the global information stores of terminating computations.

Our main goal is now to extend our failure semantics to more sophisticated agent communication languages. For example, currently, we are investigating an extension of our language which incorporates agent *signatures* such that communication of information additionally involves the *translation* of information from the signature of the sender to that of the receiving agent, as outlined in [6]. Furthermore, we aim to study the incorporation of non-monotonically increasing information stores as described in [7].

References

1. J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984. 219, 223

2. F. van Breughel. Failures, finiteness and full abstraction. In S. Brookes and M. Mislove, editors, *Proceedings of the Thirteenth Conference on the Mathematical Foundations of Programming Semantics*, volume 6 of Electronic Notes in Theoretical Computer Science. Elsevier, 1997. 216
3. L. Brim, D. Gilbert, J.-M. Jacquet, and M. Křetínský. A process algebra for synchronous concurrent constraint programming. In M. Hanus and M. Rodriquez-Artalejo, editors, *Proceedings of the fifth Conference on Algebraic and Logic Programming*, volume 1139 of *LNCS*, pages 165–178. Springer-Verlag, 1996. 216
4. S. D. Brookes, C. A. R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984. 215
5. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. A language for modular information-passing agents. In K. R. Apt, editor, *CWI Quarterly, Special issue on Constraint Programming*, volume 11, pages 273–297. CWI, Amsterdam, 1998. 214, 219
6. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Systems of communicating agents. In Henri Prade, editor, *Proceedings of the 13th biennial European Conference on Artificial Intelligence (ECAI-98)*, pages 293–297, Brighton, UK, 1998. John Wiley & Sons, Ltd. 227
7. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Information-passing and belief revision in multi-agent systems. In J. P. M. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98)*, volume 1555 of *Lecture Notes in Artificial Intelligence*, pages 29–45. Springer-Verlag, Heidelberg, 1999. 227
8. T. Finin, D. McKay, R. Fritzson, and R. McEntire. KQML: An Information and Knowledge Exchange Protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994. 214, 216
9. J. Y. Halpern and Y. Moses. A guide to the completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379, 1992. 215, 221
10. L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland Publishing, Amsterdam, 1971. 217
11. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 215
12. Jean-Hugues Réty. *Langages concurrents avec contraintes, communication par messages et distribution*. PhD thesis, University of Orleans, France, 1997. 216
13. V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Massachusetts, 1993. 215, 217
14. M. Wooldridge. Verifiable semantics for agent communication languages. In *Proceedings 3rd International Conference on Multi-Agent Systems (ICMAS'98)*, pages 349–356, Los Alamitos, California, 1998. IEEE Computer Society. 214
15. M. Wooldridge and N. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995. 214

Proof-Outlines for Threads in Java

Erika Ábrahám-Mumm[1] and Frank S. de Boer[2]

[1] Christian-Albrechts-University Kiel, Germany
eab@informatik.uni-kiel.de
[2] Utrecht University, The Netherlands
frankb@cs.uu.nl

Abstract. We introduce an assertional method for specifying and proving properties of the multi-threaded flow of control in Java. The method integrates in a modular manner reasoning about the shared-variable concurrency within one object and the communication of values between threads.

1 Introduction

The main goal of this paper is a method for specifying and proving the correctness of multi-threaded Java programs. More specifically, we introduce an abstract programming language JavaMT which focuses on the main aspects of the multi-threaded flow of control in Java.

A program in JavaMT consists of a number of class definitions and a main statement. Execution of a program starts with a root-process executing this main statement. Each class definition consists of a number of method definitions and a main statement. When an object is created it starts executing the main statement of its class. The method definitions of a class are partitioned into two sets, the synchronized methods and the non-synchronized ones. Every object can execute several methods in parallel (in an interleaved manner). However, execution of a synchronized method excludes the parallel execution of any other synchronized method of the same object.

Our proof method consists of annotating each class definition of the given program with *local assertions* which express certain properties of the values of the variables of the class. Such an annotated class definition is *locally* correct if the (standard) verification conditions hold for assignments without side-effects and the usual sequential programming constructs. On the other hand, we have no *local* verification conditions for assignments with side-effect, i.e., assignments involving the creation of a new object or a method call. As such the local assertions associated with these assignments (that is, the pre- and postcondition) express *assumptions* about the identity of the newly created object, in case of assignments involving the creation of a new object, and the return value, in case of assignments involving a method call. These assumptions will be verified in the *cooperation test*. This test is modelled after the corresponding test introduced in the proof system for CSP as described in [3]. It involves reasoning about a configuration of objects in terms of a *global* assertion language.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 229–242, 2000.

On the other hand, reasoning about the multi-threaded flow of control within an object involves an *interference freedom test*, which is modelled after the corresponding test introduced in the proof system for shared-variable concurrency as described in [9].

Our proof method is *modular* in the sense that it allows for a separate interference freedom test and cooperation test. This modularity, which in practice will simplify correctness proofs considerably, is obtained by the restriction to temporary variables in assignments with side-effects. This restriction itself corresponds with a fine-grained model of the parallel interleaving of threads.

2 The Programming Language JavaMT

In this section we introduce the programming language. We assume as given a set $\mathcal{C}$ of *class names*, with typical element c. The set $\mathcal{C} \cup \{\mathsf{Int}, \mathsf{Bool}\}$ of basic *data types*, with typical element d, we denote by $\mathcal{D}$. Here Int and Bool denote the types of the integers and booleans, respectively. For each basic data type d we denote by d^* the type of all (finite) sequences of elements of type d. The set of all (data) types $\mathcal{D} \cup \{d^* \mid d \in \mathcal{D}\}$, with typical element t, we denote by $\mathcal{T}$.

For every type t we introduce a set $IVar_t$ of instance variables of type t, with typical element x. We assume $IVar_t \cap IVar_{t'} = \emptyset$ for distinct types t and t', so that the type of each variable is uniquely determined. Such a variable $x \in IVar_t$ can refer to values of type t only. The instance variables of an object exist throughout its lifetime. On the other hand, for each $t \in \mathcal{T}$ we have a set of *temporary* variables $TVar_t$, with typical elements u and v. We assume $TVar_t \cap TVar_{t'} = \emptyset$ for distinct types t and t', so that also the type of each temporary variable is uniquely determined. Temporary variables are used as formal parameters and local variables of method definitions. Thus they only exist during the execution of the method to which they belong. By $IVar$ and $TVar$ we denote the set of instance variables and temporary variables (which are assumed to be disjoint). Elements of $IVar \cup TVar$ we denote by y.

Furthermore, we assume given a set F of operators, with typical element f. To each such an operator f corresponds a type $(t_1 \times \cdots \times t_n) \to t$ (a predicate like identity on integers, for example, is represented by a function of type $(\mathsf{Int} \times \mathsf{Int}) \to \mathsf{Bool}$). We assume for the operators $\mathsf{f} \in F$ a fixed interpretation.

We introduce the syntax of the programming language 'bottom up': we introduce expressions, statements, methods, classes, and, finally, programs, in that order.

Expressions We define the set Exp_t^c of *expressions* in class c of type t, with typical element e. We omit the typing information.

$$e ::= y \mid \mathsf{self} \mid \mathsf{nil} \mid \mathsf{f}(e_1, \ldots, e_n)$$

In order to define the set $SExp_t^c$ of expressions s with possible *side-effects*, we assume a set M of method names, with typical element m. Moreover, we assume without loss of generality that each method name m is assigned a (unique)

type $t_1 \times \cdots \times t_n \to t$, where $t_1, \ldots, t_n$ are the types of its parameters and t is the type of the result value. Again, we omit the typing information.

$$s ::= e \mid \mathsf{new} \mid e_0!m(e_1, \ldots, e_n)$$

An expression of the form x denotes the value of the instance variable x of the given object which is denoted by the expression self. The type of the expression self is determined by its context, i.e., we have $\mathsf{self} \in \mathit{Exp}^c_c$. An expression of the form u denotes the value of the temporary variable u. The expression nil denotes 'uninitialized'. It can be used for every type. The first kind of side-effect expression is a normal expression, which has no actual side-effect, of course. The second kind is the creation of a new object. We have that $\mathsf{new} \in \mathit{SExp}^c_t$, for some $t \in \mathcal{C}$ (that is, we only allow the creation of objects of user-defined classes). This new object will also be the value of the side-effect expression. The third kind of side-effect expression specifies that a message is to be sent to the object that results from e_0, with method name m and with arguments (the objects resulting from) $e_1, \ldots, e_n$. The value of an expression $e_0!m(e_1, \ldots, e_n)$ is the return value of the method.

Statements A *statement* $S \in \mathit{Stat}^c$ in a class c is constructed from assignments of the form $x := e$[1] and $u := s$ (the type of the expression e and s should correspond with that of the variable x and u, respectively). We additionally require that s does not contain *instance* variables. We assume the presence of the standard sequential operations of sequential composition, (deterministic) choice and iteration.

Methods The set Meth^c of *method definitions* in class c, with typical element μ, is defined by:

$$\mu ::= \langle m(u_1, \ldots, u_n) \mid \ S; e \rangle$$

where S and e are a statement and an expression in class c, and $S; e$ is called the body of m. We require that all u_i are different and that none of them occurs at the left hand side of an assignment in S.

When an object is sent a message, the method named in the message is invoked as follows: The parameters of the method, the temporary variables $u_1, \ldots, u_n$, are given the values specified in the message, all other temporary variables (i.e., the *local* variables of the method) are initialized to nil, and then the statement S is executed. As the result of the method the value of expression e is sent back to the sender of the message.

We restrict to a class of programs that allow the assignment of (the value of) a side-effect expression s to temporary variables only. An assignment $x := s$ to an instance variable x can be modelled by the sequential composition $v := s; x := v$ for some fresh temporary variable v. In general this introduces more interleaving points.

[1] An array assignment $x[i] := e$ we model as an assignment $x := (x[i] := e)$, where the *expression* $x[i] := e$ denotes the sequence which results from assigning to the ith element of the array x the value of e.

Classes The set *Class* of *class definitions*, with typical element C, is defined by:

$$C ::= \langle c : SYN, NSYN \mid S \rangle$$

where both SYN and $NSYN$ are disjoint (finite) sets of method definitions (we assume that all the method definitions have an unique method name).

Methods in SYN are synchronized methods, whereas the methods of $NSYN$ are not. The execution of a synchronized method excludes the parallel execution of any other synchronized method within the object. Synchronized methods in JavaMT correspond to synchronized instance methods in Java and allow to protect (instance) methods as critical sections within an object. The extension of JavaMT with synchronized statements is straightforward. Since in JavaMT each object has access only to it's own variables (and there are no static variables), the representation of synchronized static methods, i.e., critical sections within a class, in JavaMT is not necessary.

The statement S in $\langle c : SYN, NSYN \mid S \rangle$ is the main statement of the class. A newly created object of class c starts executing this statement. In JavaMT, classes with a non-empty main statement correspond to Java thread classes, where the main statement of such a class in JavaMT represents the *run* method of the thread class in Java (and not it's *constructor* method). Creation of a new object with a non-empty main statement in JavaMT corresponds to the creation of a new thread object and calling it's *start* method in Java.

In the sequel we will also refer to the statement S of a synchronized method m defined by $\langle m(u_1, \ldots, u_n) \mid S; e \rangle$ as being synchronized.

Programs A *program* $\rho = \langle C_1, \ldots, C_n \mid c : S \rangle$, finally, consists of a finite number of class definitions and a main statement $S \in Stat^c$. Execution of ρ starts with the execution of S by the root-process. We assume that c is not among the class names of the classes $C_1, \ldots, C_n$ and that all the class names of the classes $C_1, \ldots, C_n$ are different .

3 Semantics

In this section we briefly sketch the main ideas underlying the semantics of the programming language.

We assume for every type $t \in \mathcal{T}$ a set $\mathbf{O}^t$ of values of type t, with typical element α. To be precise, $\mathbf{O}^{\mathsf{Int}}$ denotes the set of integers and $\mathbf{O}^{\mathsf{Bool}}$ denotes the set of boolean values true and false, whereas for every class $c \in \mathcal{C}$ we just take for $\mathbf{O}^c$ an arbitrary infinite set *disjoint* from any other set $\mathbf{O}^d$, $d \neq c$. The elements of a set $\mathbf{O}^c$ will be used as *identities* of objects in class c, i.e., each object can be uniquely identified by it's value. Note that the value of an object is not the valuation of it's variables, but the identifier of the object. By $\mathbf{O}^{d^*}$, also with typical element α, we denote the set of all finite sequences of values in $\mathbf{O}^d_\perp (= \mathbf{O}^d \cup \{\perp\})$, where the value of nil we denote by $\perp$. By Val we denote the set of all possible values, i.e., $Val = \bigcup_t \mathbf{O}^t \cup \{\perp\}$.

We introduce the set Σ, with typical element σ, of *global states* such that $\sigma(c) \subseteq \mathbf{O}^c$, for each class c of ρ, gives the finite set of *existing* objects in class c,

i.e., the objects in c which have been created so far. Moreover, for each class c and $\alpha \in \sigma(c)$ we have $\sigma(\alpha) \in \mathit{IVar} \to \mathit{Val}$ (assuming that $\sigma(\alpha)(x) \in \mathbf{O}^t$, for $x \in \mathit{IVar}_t$).

In the sequel, in the context of a given global state σ, we will use the phrase 'object α exists in σ', or simply 'the existing object α', to indicate that $\alpha \in \sigma(c)$, for some $c \in \mathcal{C}$.

In order to represent the multi-threaded control within an object we introduce for every object α, an arbitrary infinite set $\bar{\alpha}$ of its *threads*, with typical element $i, j, \ldots$. We require that $\bar{\alpha} \cap \bar{\beta} = \emptyset$, for $\alpha \neq \beta$. By $\bar{i}$ we denote the unique object α such that $i \in \bar{\alpha}$.

In order to describe the mechanism of message passing we introduce control statements $i \uparrow e$ and $j \downarrow u$. Execution of $i \uparrow e$ by a thread j is synchronized with the execution of a corresponding statement $j \downarrow u$ by thread i and consists of assigning the result of the evaluation of e by the object $\bar{j}$ to the variable u of i. Moreover, the synchronization between different threads within an object we describe by means of a boolean instance variable sem which will be used as a *binary semaphore*. This variable is assumed not to occur in the given program.

The operational semantics of a program ρ is defined in terms of a transition relation between configurations of the form $\langle T, \sigma \rangle$, where $\sigma \in \Sigma$ and T is a finite set of triples (i, τ, S), with $\tau \in \mathit{TVar} \to \mathit{Val}$ (assuming that $\tau(u) \in \mathbf{O}^t$, for $u \in \mathit{TVar}_t$) and S a statement possibly containing the above control statements. Such a triple (i, τ, S) indicates that the statement S is to be executed by thread i in its local state τ. The local state of a thread thus specifies the values of its temporary variables. On the other hand, the values of the instance variables of an object α in a global state σ are given by its *internal* state $\sigma(\alpha)$. We refer to the full paper for a detailed description this transition relation.

4 The Assertion Language

In this section we define two different assertion languages. An *assertion* describes the state of (a part of) the system at one specific point during its execution. The first assertion language describes the *internal state* of a single object. This is called the *local* assertion language. The other one, the *global* assertion language, describes a whole system of objects.

For every type t we introduce a set of *logical* variables LVar_t of type t, with typical element z. Logical variables do not appear in programs, they are used in assertions as bound variables in quantification and for expressing the constancy of certain expressions.

4.1 The Local Assertion Language

We define the set LExp_t^c of local expressions of type t in class c, with typical element l.

Local expressions We have the following grammar for *local expressions* l (we omit the typing information)

$$l ::= y \mid \mathsf{self} \mid \mathsf{nil} \mid z \mid \mathsf{f}(l_1, \ldots, l_n)$$

The main difference with expressions e of the programming language is the presence of logical variables z.

We denote by $\mathcal{L}(l)(\omega, \sigma, \tau, \alpha)$ the result of the evaluation of l in the logical environment ω, the global state σ, the local state τ, and the existing object α. The logical environment ω assigns values to the logical variables, i.e., $\mathcal{L}(z)(\omega, \sigma, \tau, \alpha) = \omega(z)$. The values of the instance variables are given by $\sigma(\alpha)$, i.e., $\mathcal{L}(x)(\omega, \sigma, \tau, \alpha) = \sigma(\alpha)(x)$. The values of the temporary variables are given by the local state τ, i.e., $\mathcal{L}(u)(\omega, \sigma, \tau, \alpha) = \tau(u)$. Moreover, the expression self denotes the object α, i.e., $\mathcal{L}(\mathsf{self})(\omega, \sigma, \tau, \alpha) = \alpha$. Next we introduce local assertions.

Local assertions The set $LAss^c$ of *local assertions* in class c, with typical element p, is defined as follows:

$$p ::= l \mid \neg p \mid p_1 \wedge p_2 \mid \exists z(p) \mid \exists z \in l(p) \mid \exists z \sqsubseteq l(p)$$

Basic local assertions are boolean local expressions.

We denote by $\omega, \sigma, \tau, \alpha \models p$ that p is true in the logical environment ω, the global state σ, the local state τ, and the existing object α (we require that ω assigns to each logical variable (sequences) of *existing* objects only). Quantification can be applied only to logical variables. Unrestricted quantification is only allowed in case of integer and boolean logical variables. Thus in an assertion $\exists z(p)$ the variable z is required to be of type Int or Bool. In a restricted quantification $\exists z \in l(p)$[2] the variable z is of some type d and l is of type d^*. The assertion $\exists z \in l(p)$ amounts to stating the existence of an element in the sequence denoted by l for which p holds. In a restricted quantification $\exists z \sqsubseteq l(p)$ both the variable z and the expression l are of a sequence type d^*. The assertion $\exists z \sqsubseteq l(p)$ amounts to stating the existence of a *subsequence* of l for which p holds. Restricted quantification of (logical) variables ranging over (sequences of) objects ensures that the evaluation of a local assertion by an object is not affected by the other objects, i.e., for any global state σ and σ' such that $\sigma(\alpha) = \sigma'(\alpha)$ we have $\omega, \sigma, \tau, \alpha \models p$ if and only if $\omega, \sigma', \tau, \alpha \models p$. Formally, $\omega, \sigma, \tau, \alpha \models p$ is defined by a straightforward induction on the complexity of p.

4.2 The Global Assertion Language

In this section we define the global assertion language. We first introduce the set $GExp_t$ of global expressions of type t, with typical element G.

Global expressions We define $GExp_t$ by (we omit the typing information)

$$G ::= \mathsf{nil} \mid z \mid G.x \mid \mathsf{f}(G_1, \ldots, G_n)$$

We denote by $\mathcal{G}(G)(\omega, \sigma)$ the result of the evaluation of G in the logical environment ω and the global state σ. For example, $\mathcal{G}(G.x)(\omega, \sigma) = \sigma(\alpha)(x)$, where $\alpha = \mathcal{G}(G)(\omega, \sigma)$. The global expression $G.x$ thus denotes the value of the instance variable x of the object referred to by the expression G. Note that $G.x$

[2] The operation $\in$ here generalizes set-theoretical membership to sequences.

and $G'.x$ denote the same variable, that is, these expressions are *aliases*, if G and G' refer to the same object.

Global assertions The set $GAss$ of *global assertions*, with typical element P, is defined as follows:

$$P \ ::= \ G \mid \neg P \mid P_1 \wedge P_2 \mid \exists z(P)$$

Here G denotes a boolean global expression.

We denote by $\omega, \sigma \models P$ that P is true in the logical environment ω and global state σ (again, we require that ω assigns to each logical variable (sequences) of *existing* objects only). Its formal definition proceeds by a straightforward induction on the complexity of P. Quantification over (sequences of) objects is interpreted as ranging only over the *existing* objects, i.e., the objects that have been created up to the current point in the execution of the program, and the value of nil. For example, $\omega, \sigma \models \exists z P$, where z is of type c, if $\omega', \sigma \models P$, where ω' results from ω by assigning to the logical variable z an element of $\sigma(c) \cup \{\perp\}$.

The verification conditions underlying our notion of proof-outlines defined in the next section involve the following substitution operations.

Let z be a logical variable denoting an object, $\boldsymbol{u}$ a sequence of (distinct) temporary variables $u_1, \ldots, u_n$ and $\boldsymbol{G}$ a corresponding sequence $G_1, \ldots, G_n$ of global expressions.

We define a substitution operation $[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}]$ which transforms a local expression l, with temporary variables $u_1, \ldots, u_n$, into a corresponding global expression $l[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}]$. We have the following three main clauses.

$$\mathsf{self}[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}] \equiv z, \ x[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}] \equiv z.x, \ u_i[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}] \equiv G_i$$

This operation is extended to local assertions in the standard way.

We have the following main property of this substitution.

Lemma 1. *For τ a local state such that $\tau(u_i) = \mathcal{G}(G_i)(\omega, \sigma)$, $i = 1, \ldots, n$, we have that*

$$\mathcal{G}(l[z, \boldsymbol{G}/\mathsf{self}, \boldsymbol{u}])(\omega, \sigma) = \mathcal{L}(l)(\omega, \sigma, \tau, \alpha),$$

where $\alpha = \omega(z)$.

Given a local expression l (or local assertion p), we will view the temporary variables occurring in the global expression $l[z/\mathsf{self}]$ (or assertion $p[z/\mathsf{self}]$) as logical variables. Formally, these temporary variables should be replaced by logical variables. However, for notational convenience only we avoid this substitution by viewing these temporary variables at the level of global assertion language as logical variables.

Let z be a logical variable denoting an object, $\boldsymbol{x}$ a sequence $x_1, \ldots, x_n$ of (distinct) instance variables and $\boldsymbol{e}$ a corresponding sequence of expressions $e_1, \ldots, e_n$.

We define a substitution operation $[z, \boldsymbol{x} := \boldsymbol{e}]$ such that $G[z, \boldsymbol{x} := \boldsymbol{e}]$ should denote the value of G *after* the object denoted by z has executed the *simultaneous*

assignment $\boldsymbol{x} := \boldsymbol{e}$. We have the following main case. Let E_i, $i = 1, \ldots, n$, denote the global expression $e_i[z/\mathsf{self}]$.

$$(G.x_i)[z, \boldsymbol{x} := \boldsymbol{e}] \equiv \mathsf{if}\ (G[z, \boldsymbol{x} := \boldsymbol{e}]) = z\ \mathsf{then}\ E_i\ \mathsf{else}\ (G[z, \boldsymbol{x} := \boldsymbol{e}]).x_i\ \mathsf{fi}$$

This operation is extended to (global) assertions in the standard way.

The substitution operation $[z, \boldsymbol{x} := \boldsymbol{e}]$ has to account for possible aliases of the variables x_i, namely, expressions of the form $G.x_i$: It is possible that, after substitution, both G and z refer to the same object, so that $G.x_i$ is the same variable as x_i and should be substituted by E_i. It is also possible that, after substitution, G and z refer to different objects, in this case no substitution should take place. Since we cannot decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides "dynamically".

We have the following main property of this substitution.

Lemma 2. *We have that*

$$\mathcal{G}(G[z, \boldsymbol{x} := \boldsymbol{e}])(\omega, \sigma) = \mathcal{G}(G)(\omega, \sigma'),$$

where σ' results from σ by assigning $\mathcal{G}(E_i)(\omega, \sigma)$ to the instance variable x_i of $\omega(z)$, i.e., $\sigma'(\alpha)(x_i) = \mathcal{G}(E_i)(\omega, \sigma)$, where $\alpha = \omega(z)$.

5 Proof Outlines

In order to reason about the communication of values between objects, i.e., the sending of actual parameters in a method call and the return of the result value of a method, we introduce the following *communication* statements. An assignment $u := e_0!m(\boldsymbol{e})$ we translate into (the sequential composition of) the communication statements $e_0!m(\boldsymbol{e})$ and $?u$. Execution of an output statement $e_0!m(\boldsymbol{e})$ consists of the communication of the values of the actual parameters $\boldsymbol{e}$ to the object denoted by e_0, whereas the corresponding input statement $?u$ models the reception of the result value.

In order to reason about, for example, the possible interference between the different threads within an object, we also need a mechanism for the identification of threads (note that the assertion language itself only provides a mechanism for object identification). We do so by introducing for each class an auxiliary instance variable t (not occurring in the given program). For every object α, the variable t is initialized to 0 upon creation. It will be incremented each time a thread of α executes a method call. The value of t before the incrementation will be saved in an auxiliary (local) variable v and will be sent together with the object identity α to the called method as additional parameters. These parameters together with the identity of the called object can be used to identify both threads that are involved in the communication. More precisely, we replace output $e_0!m(\boldsymbol{e})$ by the statement $v, \mathsf{t} := \mathsf{t}, \mathsf{t} + 1; e_0!m(\mathsf{self}, v, \boldsymbol{e})$, where v is some 'fresh' temporary variable.

We extend each method definition $\langle m(\boldsymbol{u}) \mid S; e\rangle$ with the new formal parameters v_1 and v_2, where v_1 is used to store the (object) identity of the sender of the message[3] and v_2 is an integer variable.

Finally, in order to express certain properties of the multi-threaded flow of control of a given program we need, in general, to augment the program with *auxiliary* variables. Assignments to auxiliary variables can be added only to assignments $x := e$ (or $u := e$) without side-effect or as a separate assignment statement. The result of this is a *multiple* assignment $\boldsymbol{y} := \boldsymbol{e}$, where $\boldsymbol{y}$ denotes a sequence of mutually distinct variables and $\boldsymbol{e}$ a corresponding sequence of expressions.

Given a program ρ, we denote by ρ' the result of, first, introducing the communication statements which model explicitly the communication mechanism, then adding the mechanism for the identification of threads, and, finally, the introduction of (assignments to) arbitrary auxiliary variables. In the following definition we introduce the notion of an *annotation* of this transformation ρ' of the given program ρ.

Annotated programs An *annotation* of ρ' associates with every occurrence $\tilde{S}$ of a sub-statement S occurring in the definition of class c local assertions $pre(\tilde{S})$ and $post(\tilde{S})$ in class c. These assertions $pre(\tilde{S})$ and $post(\tilde{S})$ are also called the *precondition* and *postcondition* of the particular occurrence $\tilde{S}$ of a statement S.

In the sequel we sometimes identify the occurrence $\tilde{S}$ of a sub-statement S in a program ρ with the statement S itself. We define next the *verification conditions* which an annotated program should satisfy in order to be *locally* correct.

Local correctness An annotated program is *locally correct* if the usual verification conditions [1] for the standard sequential constructs hold. In particular, for multiple assignments of the form $\boldsymbol{y} := \boldsymbol{e}$, where $\boldsymbol{y}$ denotes a sequence $y_1, \ldots, y_n$ of variables (in $IVar \cup TVar$) and $\boldsymbol{e} = e_1, \ldots, e_n$ denotes a corresponding sequence of expressions (corresponding with respect to the types of the variables of $\boldsymbol{y}$), we have that the precondition p of $\boldsymbol{y} := \boldsymbol{e}$ and its postcondition q should satisfy the verification condition $p \rightarrow q[\boldsymbol{e}/\boldsymbol{y}]$, where $[\boldsymbol{e}/\boldsymbol{y}]$ denotes the (standard) operation of replacing simultaneously every occurrence of a variable y_i by its corresponding expression e_i.

Moreover, the precondition p of an output statement $x!m(\boldsymbol{e})$ should imply its postcondition (note that its execution does not affect the state of the active object).

Finally, for every class c of the given program there exists a local assertion I in class c, its *class invariant* which may refer only to the instance variables of c, such that for every sub-statement S (in class c) this invariant I is implied by both its associated precondition and postcondition.

[3] Formally, since the class which the sender belongs to is not known, we have to introduce a supertype o such that each class c of the given program is a subtype of o and v_1 is of type o. This is harmless because v_1 is only an auxiliary variable which as such cannot be used as the destination of a message.

The class invariant will be used in the cooperation test defined in the next section for the description of the internal state (as given by the instance variables) of the object which is about to answer a message.

Note that no *local* verification conditions are imposed on the precondition and postcondition of an input statement $?u$ and a new-statement $u :=$ new. The postcondition of an input statement thus expresses an *assumption* about the return value. Similarly, the postcondition of a new-statement expresses an assumption about the identity of the new object. Moreover, the precondition of the body of a method expresses an assumption about the actual parameters received. These assumptions will be verified in the *cooperation test* in the next section.

But first we introduce the definition of the *interference freedom test*, i.e., we define the verification condition for showing that the assertions in an annotated program are invariant under the execution of assignments occuring in the program.

Interference freedom test Let p be an assertion in class c (of the given annotated program ρ') and $\boldsymbol{x} := \boldsymbol{e}$ be a multiple assignment also occurring in class c. Note that $\boldsymbol{x}$ here are instance variables. Moreover, p or $\boldsymbol{x} := \boldsymbol{e}$ occurs in a non-synchronized statement. Let q be the precondition of $\boldsymbol{x} := \boldsymbol{e}$ and p' be the assertion p with each temporary variable u replaced by a 'fresh' temporary variable u'. We have the following verification condition which captures that p' is *invariant* over the execution of $\boldsymbol{x} := \boldsymbol{e}$.

$$(p' \wedge q \wedge (v_1 \neq v_1' \vee v_2 \neq v_2')) \rightarrow (p'[\boldsymbol{e}/\boldsymbol{x}]),$$

where v_1 and v_2 (and v_1' and v_2') are the temporary variables used for identifying the active threads. (Note that the temporary variables v_1 and v_2 of the thread executing the main statement of a class have both the value nil, see the transition for new-statements.)

In the interference freedom test we can restrict to multiple assignments to instance variables because the effect of assignments to temporary variables are only local to the thread executing the assignment. Note that we have to replace the temporary variables occurring in the assertion p in order to avoid possible name clashes with those occurring in $\boldsymbol{x} := \boldsymbol{e}$ and its associated precondition q. The local assertion $v_1 \neq v_1' \vee v_2 \neq v_2'$ expresses that we indeed are dealing with two *different* threads. Possible occurrences of the expression self in this context denotes the object executing these different threads.

The interference freedom test does not involve assignments with side-effects. This is correct because of our assumption that these assignments involve only temporary variables as target (as such their effect is only local with respect to the active thread). On the other hand, the assumptions made about assignments with side-effects are verified in the cooperation test defined in the next section. Note that for each assertion p in class c, only the invariance of p under assignments in the same class has to be verified, since objects does not have direct access to the variables of other objects.

6 The Cooperation Test

The cooperation test involves proving that a given global assertion GI is invariant over certain regions of the transformed program ρ' which are referred to as *critical sections*. In order to indicate that an occurrence $\tilde{S}$ of a sub-statement S of ρ' is a critical section we enclose $\tilde{S}$ by brackets: $\langle \tilde{S} \rangle$. The global assertion GI is allowed to refer only to auxiliary instance variables which occur only in these critical sections, consequently it will be invariant over the remaining parts of the program. Therefore, the assertion GI is also called the *global invariant*. In general it will be used to describe the communication behavior of objects.

We next describe in more detail the way these critical sections are added to our transformed program ρ'.

Critical sections The body of every method contains *critical sections* consisting only of assignments to auxiliary instance variables in the following manner:

$$\langle \boldsymbol{x}_1 := \boldsymbol{f}_1 \rangle; S; e; \langle \boldsymbol{x}_2 := \boldsymbol{f}_2 \rangle$$

In case of a synchronized method these multiple assignments $\boldsymbol{x}_1 := \boldsymbol{f}_1$ and $\boldsymbol{x}_2 := \boldsymbol{f}_2$ contain the assignments $\mathsf{sem} := \mathsf{false}$ and $\mathsf{sem} := \mathsf{true}$, respectively.

Moreover, we require that the communication statements of ρ' do occur in critical sections in the following manner:

$$\langle e_0!m(\boldsymbol{e}); \boldsymbol{x}_1 := \boldsymbol{f}_1 \rangle; \langle ?v; \boldsymbol{x}_2 := \boldsymbol{f}_2 \rangle,$$

where $\boldsymbol{x}_1 := \boldsymbol{f}_1$ and $\boldsymbol{x}_2 := \boldsymbol{f}_2$ contain only assignments to auxiliary instance variables.

Finally, we require that every new-statement $u := \mathsf{new}$ occurs in a critical section $\langle u := \mathsf{new} \rangle$ (we do not need the inclusion of assignments to auxiliary variables here; however such assignments can be included: it requires only a minor modification of the corresponding verification condition).

The cooperation test involves checking certain verification conditions which are associated with the critical sections. We first consider the critical sections containing communication statements.

The cooperation test: communication Let m be a *non-synchronized method* occurring in, say, class c', with body S_1

$$\langle \boldsymbol{x}_1 := \boldsymbol{f}_1 \rangle; S; e'; \langle \boldsymbol{x}_2 := \boldsymbol{f}_2 \rangle,$$

and formal parameters $\boldsymbol{u} = u_1, \ldots, u_n$. Let

$$\langle e_0!m(e_1, \ldots, e_n); \boldsymbol{x}_3 := \boldsymbol{f}_3 \rangle; \langle ?v; \boldsymbol{x}_4 := \boldsymbol{f}_4 \rangle$$

be a corresponding statement S_2 in class c, i.e., the expression e_0 is of type c'. Let z and z' be two distinct logical variables of type c and c'. These variables will be used to represent in the global assertion language the two communicating objects.

Let p be the precondition of the statement S_2 and r be the precondition of the body S_1 of m with every local variable of m, i.e., temporary variable which is not a formal parameter, replaced by nil.

We have the following verification condition (in the global assertion language) which guarantees invariance of GI and which additionally justifies the assumption expressed by r about the actual parameters:

$$(GI \wedge p[z/\mathsf{self}] \wedge I[z'/\mathsf{self}] \wedge e_0[z/\mathsf{self}] = z') \rightarrow (GI' \wedge r[z', \boldsymbol{E}/\mathsf{self}, \boldsymbol{u}]),$$

where I is the class invariant of c', $\boldsymbol{E}$ denotes the sequence of global expressions $E_1, \ldots, E_n$, with $E_i \equiv e_i[z/\mathsf{self}]$, and GI' denotes the assertion $GI[z', \boldsymbol{x}_1 := \boldsymbol{f}_1][z, \boldsymbol{x}_3 := \boldsymbol{f}_3]$, which results from applying the substitutions $[z', \boldsymbol{x}_1 := \boldsymbol{f}_1]$ and $[z, \boldsymbol{x}_3 := \boldsymbol{f}_3]$ to GI.

The global assertion $p[z/\mathsf{self}]$ describes the state of the sender of the message, whereas the global assertion $I[z'/\mathsf{self}]$ describes the state of the receiving object. The global assertion $e_0[z/\mathsf{self}] = z'$ states that the result of the evaluation of e_0 by the object (denoted by) z is the object denoted by z'. The global assertion GI' denotes the result of evaluating GI after the objects denoted by z and z' have executed the assignments $\boldsymbol{x}_3 := \boldsymbol{f}_3$ and $\boldsymbol{x}_1 := \boldsymbol{f}_1$, respectively. Finally, the assertion $r[z', \boldsymbol{E}/\mathsf{self}, \boldsymbol{u}]$ describes the local state of the receiver of the message after reception of the actual parameters.

Next, let p denote the precondition of the critical section $\langle ?v; \boldsymbol{x}_4 := \boldsymbol{f}_4 \rangle$. Moreover, r now denotes the postcondition of the input $?v$ which expresses an assumption of the return value. By q we denote the postcondition of the above statement S occurring in the body of m. The following verification condition guarantees invariance of GI and additionally justifies the assumption expressed by r about the return value:

$$(GI \wedge p[z/\mathsf{self}] \wedge q[z', \boldsymbol{E}/\mathsf{self}, \boldsymbol{u}] \wedge e_0[z/\mathsf{self}] = z') \rightarrow (GI' \wedge r[z, E'/\mathsf{self}, v]),$$

where GI' denotes the assertion $GI[z', \boldsymbol{x}_2 := \boldsymbol{f}_2][z, \boldsymbol{x}_4 := \boldsymbol{f}_4]$. Moreover, $\boldsymbol{E}$ denotes the sequence of actual parameters expressed in the global assertion language as described above and E' denotes the global expression $e'[z/\mathsf{self}]$.

Note that $q[z', \boldsymbol{E}/\mathsf{self}, \boldsymbol{u}]$ indeed characterizes the state of the receiver of the message because of the assumption that the formal parameters $\boldsymbol{u}$ are read-only and because the values of the actual parameters $\boldsymbol{e}$ are not affected (since they do not involve instance variables). Moreover, because of the built-in mechanism for the identification of threads the substitution of the first two formal parameters in q by their corresponding actual parameters ensures that $q[z', \boldsymbol{E}/\mathsf{self}, \boldsymbol{u}]$ indeed uniquely identifies the thread activated by the calling thread. Finally, we remark that without loss of generality we may assume that the sets of temporary variables occurring in p, r, and the expressions $e_0, e_1, \ldots, e_n$, and those occurring in q are disjoint. In fact these variables are viewed as *logical variables* in the global assertion language (this assumption allows us to surpress the explicit renaming of these temporary variables into disjoint sets of logical variables).

In case of a synchronized method we only have to add to the antecedence of the first verification condition above the information that $z'.\mathsf{sem} = \mathsf{true}$ which indicates that new threads can be created. Note that by definition sem is reset by the critical sections of the method.

Next we consider critical sections containing new statements.

The cooperation test: object creation Let p and q be the precondition and postcondition of a critical section $\langle u := \mathsf{new}\rangle$ (occurring in, say, class c') Let u be a temporary variable of type c, and the logical variables z and z' be of type c' and c^*, respectively. The logical variable z' will be used to store the objects in class c which exist before the creation of the new object. The logical variable z will be used to denote the object executing the statement $u := \mathsf{new}$. Moreover, let V be the set of the instance variables occurring in class c. Given that z' stores all the old objects in class c, that p holds for the old value of u then can be expressed in the global assertion language by restricting all the quantification involving objects in class c in $\exists u(p[z/\mathsf{self}])$ to z'. Similarly, that GI holds before the creation of the new object then can also be expressed by restricting all the quantification involving objects in class c in GI to z'. Let P denote the global assertion $\exists u(p[z/\mathsf{self}])$ and $\downarrow z'$ denote the operation which when applied to a global assertion restricts all quantifications involving objects in class c to z'. The *strongest* postcondition of the assignment $u := \mathsf{new}$ with respect to the given precondition $GI \wedge P$, is expressed by the following global assertion:

$$\exists z'\Big((GI \wedge P) \downarrow z' \wedge \mathit{Init}(z')\Big),$$

where $\mathit{Init}(z')$ denotes the following (global) assertion

$$u \neq \mathsf{nil} \wedge u \notin z' \wedge \forall v(v \in z' \vee v = u) \wedge u.\mathsf{sem} \wedge \bigwedge_{x \in V} u.x = \mathsf{nil},$$

This assertion expresses that u denotes a new object and that its semaphore sem is initialized to true, whereas its other instance variables are initialized to the value of nil. Note that we treat the temporary variable u as a logical variable.

Let R be the above strongest postcondition and Q be the assertion $q[z/\mathsf{self}]$. The following verification condition guarantees invariance of GI and justifies the assumption about the identity of the new object expressed by q:

$$R \rightarrow (Q \wedge GI).$$

7 Conclusion

We refer to the full paper for proofs of soundness and completeness.

There is a lot of literature on the semantics and proof theory of Java (here we only mention [4,6,8]). To the best of our knowledge the method presented in this paper is the first sound and complete method for reasoning about the multi-threaded flow of control in Java.

The proof method itself can be considered as a further development of the work in [5] and [7]. In the language POOL considered in [5] objects only have a single-threaded flow of control. Consequently the proof method requires only a cooperation test. One of the main difference with the work reported in [7] is that our proof method integrates the interference freedom test and the cooperation

test as introduced for shared-variable concurrency and CSP in an object-oriented context.

Ultimately the goal is a definition of a concrete syntax of assertions for annotating real-life Java programs and the development of a compiler which translates these annotated Java programs into corresponding verification conditions. A theorem prover then could be used for verifying the validity of these verifications conditions. Of particular interest in this context is an integration of our method in the system developed in the LOOP project [8].

Acknowledgement

We thank Willem-Paul de Roever for his valuable comments on earlier drafts of this paper.

References

1. K. R. Apt: Ten years of Hoare logic: a survey — part I. ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, pp. 431–483. 237
2. K. R. Apt. Formal justification of a proof system for Communicating Sequential Processes. Journal of the ACM, Vol. 30, No. 1, January 1983, pp. 197–216.
3. K. R. Apt, N. Francez, and W. P. de Roever. A proof system for Communicating Sequential Processes. ACM Transactions on Programming Languages and Systems, 2:359-385, 1980. 229
4. E. Boerger and W. Schulte. Modular Design for the Java Virtual Machine Architecture. In Architecture Design and Validation Methods. Lecture Notes in Computer Science, 1999. 241
5. F. S. de Boer. A proof system for the parallel object-oriented language POOL. Proceedings of the seventeenth International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, Vol. 443, 1990. 241
6. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In Formal Syntax and Semantics of Java, Lecture Notes in Computer Science, Vol. 1523, 1999. 241
7. R. T. Gerth and W.-P. de Roever. Proving monitors revisited: A first step towards verifying object oriented systems. Fundamenta informaticae IX, North-Holland, p. 371-400, 1986. 241
8. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools. Proceedings of the European Symposium on Programming, Lecture Notes in Computer Science, Vol. 1381, 1998. 241, 242
9. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. Acta Informatika, 6:319-340, 1976. 230
10. J. V. Tucker and J. I. Zucker. Program Correctness over Abstract Data Types, with Error-State Semantics. CWI Monograph Series, Vol. 6, Centre for Mathematics and Computer Science/North-Holland, 1988.

Deriving Bisimulation Congruences for Reactive Systems

James J. Leifer and Robin Milner

University of Cambridge Computer Laboratory
New Museums Site, Pembroke St., Cambridge CB2 3QG, UK
{James.Leifer,Robin.Milner}@cl.cam.ac.uk

Abstract. The dynamics of reactive systems, e.g. CCS, has often been defined using a labelled transition system (LTS). More recently it has become natural in defining dynamics to use reaction rules – i.e. unlabelled transition rules – together with a structural congruence. But LTSs lead more naturally to behavioural equivalences. So one would like to *derive* from reaction rules a suitable LTS.
This paper shows how to derive an LTS for a wide range of reactive systems. A *label* for an agent a is defined to be any context F which intuitively is just large enough so that the agent Fa ("a in context F") is able to perform a reaction. The key contribution of this paper is a precise definition of "just large enough", in terms of the categorical notion of *relative pushout* (RPO), which ensures that bisimilarity is a congruence when sufficient RPOs exist. Two examples – a simplified form of action calculi and term-rewriting – are given, for which it is shown that sufficient RPOs indeed exist. The thrust of this paper is, therefore, towards a general method for achieving useful behavioural congruence relations.

1 Purpose

The semantics of interactive systems is in a state of flux, inevitably so because new models for such systems are constantly appearing. Frequently, a calculus is developed to model certain features (e.g. communication, mobility and security) and the behaviour of agents is described in terms of state transition rules, also called reduction rules, rewriting rules, firing rules, etc.; we shall call them *reaction rules*. The question of behavioural equivalence between two agents immediately arises.

A sledgehammer approach to behavioural equivalence is in terms of *contexts*. It is often easy to determine, for a calculus, the class of all possible contexts C in which agents may appear; then we can declare that two agents a and b are *contextually equivalent* – here written $a \sim b$ – iff for all contexts C the agents Ca and Cb have the same reaction pattern (which may be defined differently for different kinds of equivalence). This definition has the advantage of making $\sim$ a congruence ($a \sim b$ implies $Ca \sim Cb$), and the disadvantage that to check equivalence one has to consider *all* contexts.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 243–258, 2000.

A common and more practical approach has been to define (by rules) not only the *reactions* $a \longrightarrow\!\triangleright\, a'$ of each agent, but also a system of *labelled transitions* $a \xrightarrow{\lambda}\!\triangleright\, a'$, where the *label* λ is drawn from some tractable set representing all the "ways" in which an agent may interact with its environment. We may then define $a \sim b$ to mean that a and b have the same pattern (traces, bisimilarity, ...) of labelled transitions, not merely the same pattern of reactions. But we are still faced with proving a congruence property with respect to some class of contexts; this proof may be hard, and is often ad hoc for each calculus.

This paper offers a general method for deriving a labelled transition system (LTS) whose labels are a restricted class of contexts. The crux of the paper is that these labels are defined in terms of the categorical notion of *relative pushout* (RPO), and that the induced bisimulation equivalence (either strong or weak) is guaranteed to be a congruence when sufficient RPOs exist.

2 Background and Outline

Since the early days of process calculi, the question of behavioural equivalence has been central. It has usually been operationally defined, and often centred upon an LTS; this was the case with CCS [15]. There have indeed been notable exceptions to the use of LTSs as the defining method: Hoare's CSP [12] was given an elegant denotational semantics, the *failures* model; in the Process Algebra [3] which originated with Bergstra and Klop the emphasis was upon an algebraic theory rather than upon transitions. But LTSs have been prominent, and they led to an intense study of the different equivalences they induce [9], and of their congruential properties [10,24].

With the π-calculus [18] the LTS methodology became strained because the passage of names as messages required a somewhat ad hoc structure in the labels. For this reason Milner [16], inspired by the Chemical Abstract Machine of Berry and Boudol [4], devised an alternate semantics based upon structural congruence and reaction rules, with specific definitions of behavioural equivalence and specific congruence proofs, often based upon barbed bisimulation [19].

Simultaneously, action calculi [17] were proposed as a framework embracing a wide variety of process calculi. Many calculi, including the λ-calculus, the π-calculus, Petri nets and the Ambient calculus can be presented as action calculi, which employ a uniform notion of structural congruence. Thus arose the challenge to find a general way of deriving LTSs, and thence behavioural congruences, from reaction rules all expressed within action calculi.

Sewell [22] has derived an LTS for several classes of reactive system, and in each case proved the induced bisimilarity to be a congruence. He also proposed a notion of *colouring* to keep track of component occurrences, and thereby to yield satisfactory congruences. This work has given guidance on what a uniform approach might be, and on which congruences it should yield. We here offer a uniform approach applying to any reactive system which forms a category possessing *relative pushouts*; in our ongoing work we aim to demonstrate that action calculi enjoy this property.

Outline In the next section we discuss the derivation of LTSs and motivate the use of contexts as labels. In Section 4 we define the notion of *relative pushout* (RPO) and the sister notion of an *idem pushout* (IPO) – a self-relative RPO. In Section 5, we define the labelled transitions of an LTS in terms of IPOs. We then prove that the associated strong bisimilarity is a congruence; we also show that a weak bisimilarity is a congruence. In Sections 6 and 7 we study two examples: a simple class of graphs related to action calculi [17], and term-rewriting with "multi-hole" contexts, in comparison with Sewell's study [22]. Current and future work is discussed in Section 8.

3 Motivation

We wish to answer two questions about arbitrary reactive systems consisting of agents (whose syntax may be quotiented by a structural congruence) and a reaction relation $\longrightarrow\!\!\triangleright$ (generated by reaction rules):

1. Can we *derive* a labelled transition relation $\xrightarrow{\lambda}\!\!\triangleright$ where λ comes from a small set of labels that intuitively reflect how an agent interacts with its environment?
2. Under what general conditions is bisimulation over $\xrightarrow{\lambda}\!\!\triangleright$ a congruence?

We can begin to address question 1 by considering CCS [15]. Let a, b range over agents (processes), C, D, F range over agent contexts (processes with a hole), and x range over names. The usual labelled transitions $\xrightarrow{\lambda}\!\!\triangleright$ for $\lambda ::= \bar{x} \mid x \mid \tau$ reflect an agent's *capability* to engage in some behaviour, e.g. $\bar{x}.a|b$ has the labelled transition $\xrightarrow{\bar{x}}\!\!\triangleright$ because $\bar{x}.a|b$ can perform an output on x. However, if we shift our emphasis from characterising the capabilities of an agent to the *contexts* that cause the agent to react, then we gain an approximate answer to question 1, namely we define

$$a \xrightarrow{F}\!\!\triangleright\, a' \qquad \text{iff} \qquad Fa \longrightarrow\!\!\triangleright\, a' \tag{1}$$

for all contexts F. (We denote context composition and application by juxtaposition throughout.) Instead of observing that $\bar{x}.a|b$ "can do an $\bar{x}$" we might instead see that it "interacts with an environment that offers to input on x, i.e. reacts when placed in the context $-|x$". Thus, $\bar{x}.a|b \xrightarrow{-|x}\!\!\triangleright\, a|b$.

The definition of labelled transition in (1) is attractive when applied to an arbitrary process calculus because it in no way depends upon the presence or absence of structural congruence. Furthermore, it is generated entirely from the reaction relation $\longrightarrow\!\!\triangleright$ (question 1); and, bisimulation over $\xrightarrow{F}\!\!\triangleright$ is a congruence, (question 2). The proof of the latter is straightforward: let C be an arbitrary context and suppose $a \sim b$; we show that $Ca \sim Cb$. Suppose $Ca \xrightarrow{F}\!\!\triangleright\, a'$; by definition, $FCa \longrightarrow\!\!\triangleright\, a'$, hence $a \xrightarrow{FC}\!\!\triangleright\, a'$. Since $a \sim b$, there exists b' such that $b \xrightarrow{FC}\!\!\triangleright\, b'$ and $a' \sim b'$. Hence $Cb \xrightarrow{F}\!\!\triangleright\, b'$, as desired.

Nonetheless, the definition in (1) is unsatisfactory: the label F comes from the set of *all* agent contexts – not the "small set" asked for in question 1 –

thus making bisimulation proofs intolerably heavy. Also, the definition fails to capture its intended meaning that $a \xrightarrow{F} a'$ holds when a *requires* the context F to enable a reaction: there is nothing about the reaction $Fa \longrightarrow a'$ that forces all of F – or indeed any of F – to be used. In particular, if $a \longrightarrow a'$ then for all contexts F that preserve reaction, $Fa \longrightarrow Fa'$, hence $a \xrightarrow{F} Fa'$; thus a has many labelled transitions that reflect nothing about the behaviour of a itself.

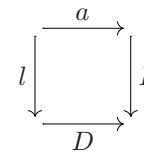
Let us unpack (1) to understand in detail where it goes wrong. Consider an arbitrary reactive system equipped with a set Reacts of reaction rules; the reaction relation $\longrightarrow$ contains Reacts and is preserved by all contexts:

$$l \longrightarrow r \quad \text{if } (l, r) \in \mathsf{Reacts} \qquad \frac{a \longrightarrow a'}{Ca \longrightarrow Ca'} \quad .$$

Expanding (1) according to this definition of $\longrightarrow$ we have:

$$\begin{aligned} a \xrightarrow{F} a' \quad & \text{iff} \quad Fa \longrightarrow a' \\ & \text{iff} \quad \exists (l, r) \in \mathsf{Reacts}, D. \quad Fa = Dl \;\&\; a' = Dr \quad . \end{aligned} \tag{2}$$

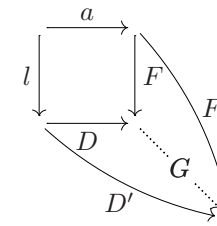
The requirement $Fa = Dl$ in (2) is rendered by a commuting square (as shown) in some category whose arrows are the agents and contexts of the reactive system. This requirement reveals the flaw described earlier: nothing in (2) forces F and D to be a "small upper bound" on a and l.



For the past few years we have been searching for a result for action calculi which we call a "dissection lemma", having roughly the form: given $Fa = Dl$, there exists a "maximum" C, such that for some F' and D' we have $F'a = D'l$, $F = CF'$ and $D = CD'$. Sewell's already cited congruence proofs [22] indeed used dissection lemmas, even though they did not assert maximality. To capture our intuition of maximality, we construct below a category-theoretic framework, in which we then obtain an elegant and general proof of congruence for the induced bisimulation equivalence.

4 Relative Pushouts

The standard way of characterising that F and D are a "least upper bound" for a and l is to assert that the square for (2) is a *pushout*, i.e. has the property: $Fa = Dl$, and for every F' and D' satisfying $F'a = D'l$ there exists a unique G such that $GF = F'$ and $GD = D'$, as shown.



Unfortunately, pushouts rarely exist in the categories that interest us. Consider, for example, a category of term contexts over a signature Σ; its objects consist of 0 and 1; its arrows $0 \to 1$ are terms over Σ; its arrows $1 \to 1$ are one-hole contexts over Σ; there are no arrows $1 \to 0$ and exactly one arrow $\mathsf{id}_0 : 0 \to 0$. Now, if Σ contains only constant symbols, say $\Sigma = \{\alpha, \alpha'\}$, then there is no pushout completing Fig. 1(1) because there are no contexts other

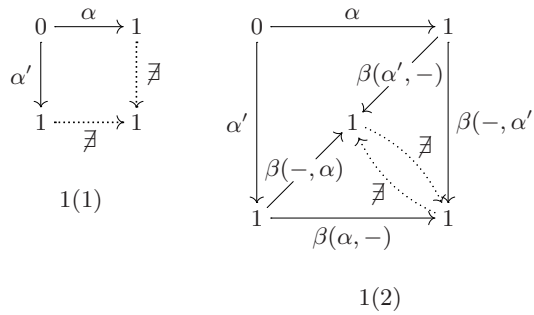


Figure 1. Non-existence of pushouts

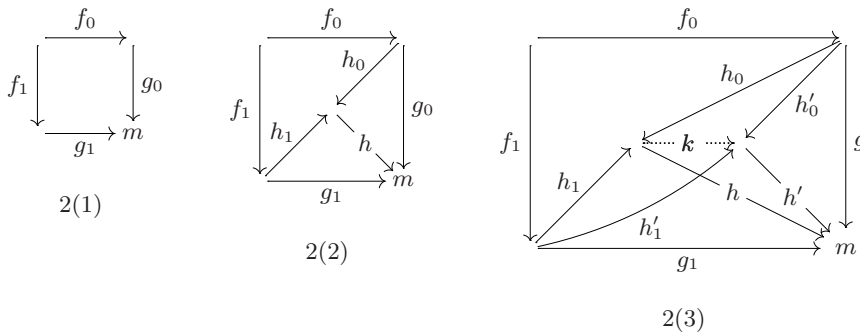


Figure 2. Construction of an RPO

than the identity. If we introduce a 2-place function symbol β into Σ, we can construct an upper bound for α and α' but still no pushout (Fig. 1(2)).

We now define *relative pushouts* (RPOs) which exist, unlike pushouts, in many categories of agent contexts (illustrated in later sections). Let $\mathbf{C}$ be an arbitrary category whose arrows and objects we denote by f, g, h, k and m, n; in pictures we omit labels on the objects when possible.

Definition 1 (RPO). *Given a commuting square (Fig. 2(1)) consisting of* $g_0 f_0 = g_1 f_1$, *an* RPO *is a triple* h_0, h_1, h *satisfying two properties:*

commutation: $h_0 f_0 = h_1 f_1$ *and* $h h_i = g_i$ *for* $i = 0, 1$ *(Fig. 2(2));*
universality: for any h'_0, h'_1, h' *satisfying* $h'_0 f_0 = h'_1 f_1$ *and* $h' h'_i = g_i$ *for* $i = 0, 1$, *there exists a unique* k *such that* $h' k = h$ *and* $k h_i = h'_i$ *(Fig. 2(3)).*

(An RPO for Fig. 2(1) is just a pushout in the slice category of $\mathbf{C}$ over m.)

A square is called an *idem pushout* (IPO) if it has an RPO of a special kind:

Definition 2 (IPO). *The commuting square in Fig. 2(1) is an IPO if the triple* g_0, g_1, id_m *is an RPO.*

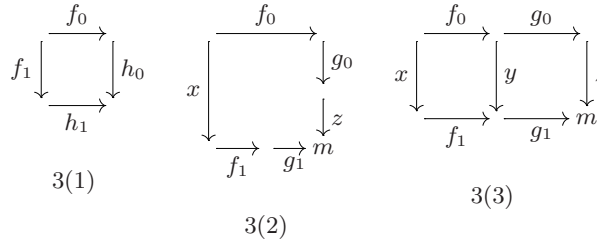


Figure 3. IPO lemmas

The difference between a pushout and an IPO is clearest in a partial order category: a pushout is a least upper bound (i.e. less than any other upper bound) and an IPO is a minimal upper bound (i.e. not greater than any other upper bound). IPOs form the basis of our abstract definition of labelled transition and, by the following proposition, their existence follows from that of RPOs:

Proposition 1 (IPOs from RPOs). *If Fig. 2(2) is an RPO diagram then the square in Fig. 3(1) is an IPO.*

IPOs can be pasted together as shown by the following proposition, analogous to the standard pasting result for pushouts:

Proposition 2 (IPO pasting). *Suppose that both squares in Fig. 3(3) commute and that Fig. 3(2) has an RPO.*

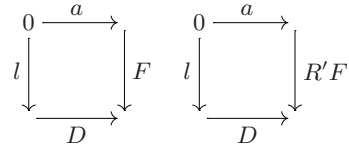
(i) *If the two squares of Fig. 3(3) are IPOs then so is the big rectangle.*
(ii) *If the big rectangle and the left square of Fig. 3(3) are IPOs then so is the right square.*

5 Labelled Transitions and Bisimulation Congruence

We have set up in the previous section the categorical technology we need. We now give a formal definition of a "reactive system" and proceed to derive therefrom a labelled transition system. We then prove that, subject to the existence of sufficiently many RPOs, the associated bisimulation equivalence is a congruence.

Definition 3 (reactive system). *A* reactive system *consists of a category* $\mathbf{C}$ *with added structure. We let* m, n *range over objects.* $\mathbf{C}$ *has the following extra components:*

- *a distinguished object* 0 *(not necessarily initial);*
- *a set* $\mathsf{Reacts} \subseteq \bigcup_m \mathbf{C}(0,m)^2$ *of* reaction rules*;*
- *a subcategory* $\mathbf{D}$ *of* $\mathbf{C}$*, whose arrows are the* reactive contexts*, with the property that* $D_1 D_0 \in \mathbf{D}$ *implies* $D_1, D_0 \in \mathbf{D}$*.*



4(1) 4(2)

Figure 4. IPO squares for labelled transitions

We think of the arrows of $\mathbf{C}$ as agents and contexts; the reactive contexts $\mathbf{D}$ are those in which reaction will be permitted. We write e.g. $D \in \mathbf{D}$ to mean that D is an arrow of $\mathbf{D}$. We let C, D, F range over arrows; we use a, b, l, r for arrows (the "agents") with domain 0. Note that if $(l, r) \in \mathsf{Reacts}$ then $l, r : 0 \to m$ for some m. The objects m of $\mathbf{C}$ represent interfaces between contexts. At this level of abstraction we specify no structure on objects, except to distinguish 0.

The reaction relation $\longrightarrow\!\!\triangleright$ is generated from Reacts by closing up under all reactive contexts:

Definition 4 (reaction). $a \longrightarrow\!\!\triangleright a'$ *iff there exists* $(l, r) \in \mathsf{Reacts}$ *and* $D \in \mathbf{D}$ *such that* $a = Dl$ *and* $a' = Dr$.

We now give our main definition. We replace the commuting square of (2) (Section 3) with an IPO, defining labelled transitions as follows:

Definition 5 (labelled transition). $a \xrightarrow[1]{F}\!\!\triangleright\, a'$ *iff there exists* $(l, r) \in \mathsf{Reacts}$ *and* $D \in \mathbf{D}$ *such that Fig. 4(1) is an IPO and* $a' = Dr$.

This definition assures that F, D provides a minimal upper bound on a and l, as required in Section 3. For suppose there is another upper bound F', D', i.e. $F'a = D'l$, and also $F = RF'$ and $D = RD'$ for some R. Then the IPO property for Fig. 4(1) ensures that for some R' (with $RR' = \mathsf{id}$) we have $F' = R'F$ and $D' = R'D$ – so F, D provides a "lesser" upper bound than F', D' after all.

Proposition 3. *For all contexts* F *we have that* $a \xrightarrow[1]{F}\!\!\triangleright\, a'$ *implies* $Fa \longrightarrow\!\!\triangleright\, a'$.

The converse fails in general (which is good, given the remarks made in Section 3 about the tentative definition (1) of labelled transitions). We return to the converse property later in the special case that F is an isomorphism.

Bisimulation over $\xrightarrow[1]{F}\!\!\triangleright$ follows its usual scheme [21]:

Definition 6 (bisimulation over $\xrightarrow[1]{F}\!\!\triangleright$). *Let* $\mathcal{S} \subseteq \bigcup_m \mathbf{C}(0, m)^2$. $\mathcal{S}$ *is a* simulation *over* $\xrightarrow[1]{F}\!\!\triangleright$ *iff for* $(a, b) \in \mathcal{S}$, *if* $a \xrightarrow[1]{F}\!\!\triangleright\, a'$ *then there exists* b' *such that* $b \xrightarrow[1]{F}\!\!\triangleright\, b'$ *and* $(a', b') \in \mathcal{S}$. $\mathcal{S}$ *is a* bisimulation *iff* $\mathcal{S}$ *and* $\mathcal{S}^{-1}$ *are simulations. Let* $\sim_1$ *be the largest bisimulation over* $\xrightarrow[1]{F}\!\!\triangleright$.

We now state and prove the central result of this paper: if $\mathbf{C}$ has a sufficiently rich collection of RPOs then $\sim_1$ is a congruence.

Definition 7 (redex-RPOs). *We say that* $\mathbf{C}$ has all redex-RPOs *if for all* $(l, r) \in \mathsf{Reacts}$ *and arrows* a, F, D *such that* $D \in \mathbf{D}$ *and* $Fa = Dl$*, the square in Fig. 4(1) has an RPO.*

Theorem 1 (strong congruence). *If* $\mathbf{C}$ *has all redex-RPOs then* $\sim_1$ *is a congruence, i.e.* $a \sim_1 b$ *implies* $Ca \sim_1 Cb$ *for all* C*.*

Proof. It is sufficient to show that the following relation is a bisimulation:

$$\mathcal{S} \triangleq \{(Ca, Cb) \mid a \sim_1 b\} \quad .$$

The proof falls into three parts, each of which is an implication as illustrated in Fig. 5(1). Dashed lines connect pairs of points contained within the relation annotating the line. Each arrow "⇓" is tagged by the part of the proof below that justifies the implication.

(i) If $Ca \xrightarrow[1]{F} a'$ then, by definition, there exists $(l, r) \in \mathsf{Reacts}$ and $D \in \mathbf{D}$ such that the big rectangle in Fig. 5(2) is an IPO and $a' = Dr$. Because $\mathbf{C}$ has all redex-RPOs, there exists F', D', C' forming an RPO as in Fig. 5(2); moreover, $D', C' \in \mathbf{D}$ since $C'D' = D \in \mathbf{D}$. By Prop. 1, Fig. 5(3) is an IPO. Because $\mathbf{C}$ has all redex-RPOs, Prop. 2 implies that Fig. 5(4) is an IPO too. By definition, $a \xrightarrow[1]{F'} D'r$ and $a' = C'D'r$.

(ii) Since $a \sim_1 b$, there exists b'' such that $b \xrightarrow[1]{F'} b''$ and $D'r \sim_1 b''$. By definition there exists $(l', r') \in \mathsf{Reacts}$ and $E' \in \mathbf{D}$ such that Fig. 5(5) is an IPO and $b'' = E'r'$.

(iii) Because $\mathbf{C}$ has all redex-RPOs, Prop. 2 implies that we can paste Fig. 5(5) with Fig. 5(4) (both IPOs) along F' and conclude that Fig. 5(6) is an IPO. Hence $Cb \xrightarrow[1]{F} C'E'r'$ and $(C'D'r, C'E'r') \in \mathcal{S}$ because $D'r \sim_1 E'r'$, as desired.

The crux of the above proof is that Fig. 5(4), which mediates between an F'-labelled transition of a and an F-labelled transition of Ca, can be pasted onto a new diagram, serving the same function for b and Cb. This essential idea appears to be robust under variation both of the definition of labelled transition and of the congruence being established.

We now define two variants of $\xrightarrow[1]{F}$ for which transitions labelled by an isomorphism F recover the reaction relation, i.e. $a \xrightarrow[i]{F} a'$ iff $Fa \longrightarrow a'$ for $i = 2, 3$ (cf. Prop. 3): here the isomorphisms play the role of the τ-label in π-calculus. The first is defined by brute-force case analysis:

Definition 8. $a \xrightarrow[2]{F} a'$ *iff* $\begin{cases} Fa \longrightarrow a' & \text{, if } F \text{ is an isomorphism} \\ a \xrightarrow[1]{F} a' & \text{, otherwise.} \end{cases}$

The second involves the existence of a retraction, a pair R, R' with $RR' = \mathsf{id}$, that adds just enough flexibility to the IPO condition:

Definition 9. $a \xrightarrow[3]{F} a'$ *iff there exists* $(l, r) \in \mathsf{Reacts}$*,* $D \in \mathbf{D}$*,* R*, and* R' *such that Fig. 4(2) is an IPO,* $a' = RDr$*, and* $RR' = \mathsf{id}$*.*

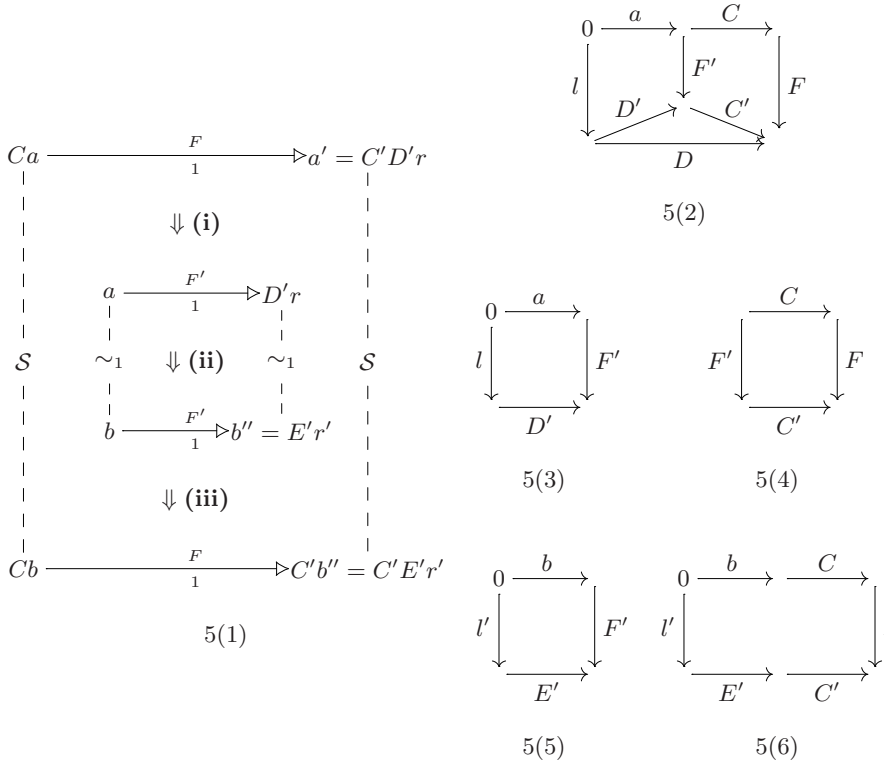


Figure 5. Congruence proof

Finally, let $\sim_4$ be the bisimulation induced by the definition of labelled transition given in (1) (Section 3). The induced bisimulations of the different labelled transition relations are congruences and related as follows:

Theorem 2. *If* $\mathbf{C}$ *has all redex-RPOs then* $\sim_i$ *is a congruence for* $i = 2, 3, 4$ *and* $\sim_1 \subseteq \sim_2 \subseteq \sim_3 \subseteq \sim_4$.

We expect that some of these congruences coincide in specific applications. We shall also seek category theoretic conditions under which they provably coincide.

This theory generalises smoothly both to weak bisimulation [15] and to trace equivalences. For weak bisimulation, we think of the isomorphism labels as "silent moves":

Definition 10 (weak labelled transition).

$$a \overset{F}{\Longrightarrow} a' \quad \textit{iff} \quad \begin{cases} Fa \longrightarrow^* a' & \text{, if } F \textit{ is an isomorphism} \\ a \xrightarrow[1]{F} \longrightarrow^* a' & \text{, otherwise} \end{cases}$$

Let $\approx$ be the largest bisimulation over $\overset{F}{\Longrightarrow}$.

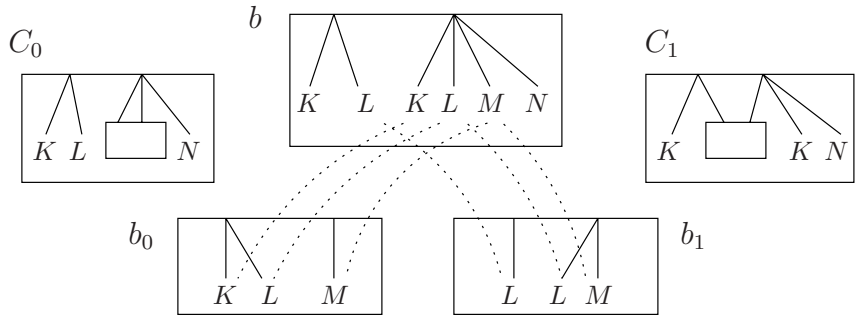


Figure 6. Composition of bunch contexts and bunches

Theorem 3 (weak congruence). *If* **C** *has all redex-RPOs then* $\approx$ *is a congruence.*

The original weak bisimilarity of CCS employed a looser definition of $\overset{F}{\Longrightarrow}$, in which (using current notation) the compound transition $\xrightarrow{F}_1 \longrightarrow^*$ was replaced by $\longrightarrow^* \xrightarrow{F}_1 \longrightarrow^*$. It was not a congruence for CCS, though it was preserved by the CCS equivalent of reactive contexts. Interestingly, if the above replacement is made in Def. 10, then $\approx$ is preserved by reactive contexts.

6 Example 1: Wiring and Bunches

In this section and the next we present two examples of reactive systems in which RPOs exist. They are not chosen to represent practical systems, but to illustrate clearly the three main features of action calculi [17]: *parallel composition*, *wiring*, and *nesting of agents*.

Our first example is motivated by parallel composition and wiring. We study a simple class of agents which we call *bunches*, that exhibit some of the variety of copied wiring (naming) inherent in, for example, the π-calculus and clearly apparent in a graphical presentation. A bunch is a finite ordered set of unordered trees of depth one, each leaf vertex possessing a character from a fixed *character set* $\mathcal{K} = \{K, L, M, \dots\}$. Three bunches b, b_0, b_1 are shown in Fig. 6 (ignore the dotted lines for now). Two *bunch contexts* C_0, C_1 are also shown, each with a single hole; putting b_0 into C_0 and b_1 into C_1 yields $b = C_0 b_0 = C_1 b_1$. We define **Bun** formally as follows:

Definition 11 (interfaces and bunch contexts). *The bunch category* **Bun** *has* interfaces (U, m) *as objects, where* U *is a finite set of* vertices *and the ordinal* $m = \{0, \dots, m-1\}$ *represents an ordered set of* roots. *An arrow of* **Bun** *is a* bunch context $C = (t, root, char) : (U_0, m_0) \to (U_1, m_1)$ *whose components,*

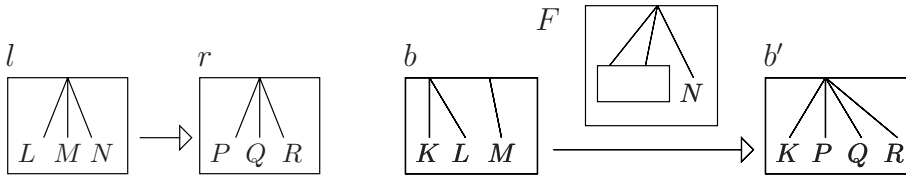


Figure 7. A reaction rule $l \longrightarrow\!\!\triangleright\, r$ and a labelled transition $b \xrightarrow{F}\!\!\triangleright\, b'$

where $V = U_1 - t(U_0)$ *is the* vertex set*, are:*

$$\begin{aligned} t : U_0 &\rightarrowtail U_1 && \textit{the} \text{ trail } \textit{(injective)} \\ root : V \oplus m_0 &\twoheadrightarrow m_1 && \textit{the} \text{ parent map } \textit{(surjective)} \\ char : V &\to \mathcal{K} && \textit{the} \text{ character map.} \end{aligned}$$

If $(U_0, m_0) = (\emptyset, 0)$ *we call* C *a* bunch*; we typically use* b *for a bunch. In* **Bun***, every context is reactive.*

(In this example we use "$\oplus$" to combine disjoint sets and functions with disjoint domains.) Composition of contexts is easy to understand graphically. Formally:

Definition 12 (identities and composition). *The* identity *context* $\mathsf{id}_{(U,m)} \triangleq (id_U, id_m, \emptyset)$*. For two contexts* $C_i = (t_i, root_i, char_i) : (U_i, m_i) \to (U_{i+1}, m_{i+1})$ *(*$i = 0, 1$*), their* composition $C_1 C_0 \triangleq (t, root, char) : (U_0, m_0) \to (U_2, m_2)$ *is determined as follows, where* V_i *are the vertex sets of* C_i *and* $V = V_1 \oplus t_1(V_0)$*:*

$$\begin{aligned} t &\triangleq t_1 \circ t_0 \\ root : V \oplus m_0 \twoheadrightarrow m_2 &\triangleq root_1 \circ (id_{V_1} \oplus (root_0 \circ (t_1^{-1} \oplus id_{m_0}))) \\ char : V \to \mathcal{K} &\triangleq char_1 \oplus (char_0 \circ t_1^{-1}). \end{aligned}$$

This definition yields a category. To see how interfaces and trails work, consider C_0 in Fig. 6. When a graph with 3 vertices and 2 roots is placed in the hole, the result is a graph with 6 vertices and 2 roots. Naming the vertices suitably we have $C_0 : (\{v_0, v_1, v_2\}, 2) \to (\{v_0, \ldots, v_5\}, 2)$. Then the trail of C_0 is $t_0 : v_i \mapsto v_{i+2}$ $(i = 0, 1, 2)$; this is indicated in Fig. 6 by the dotted lines from the vertices of b_0 to those of b. Trails are a version of Sewell's notion of colouring. They assure the following:

Theorem 4. Bun *has all RPOs, hence all redex-RPOs.*

There is a natural alternative version of **Bun** without trails; we lack space for it, but a counter-example indicates that the RPO property is then lost.

The labels obtained via IPOs in **Bun** are pleasantly simple. A reaction rule $l \longrightarrow\!\!\triangleright\, r$ is shown in Fig. 7, with an example of a corresponding labelled transition $b \xrightarrow{F}\!\!\triangleright\, b'$; the label context F supplies the parts of l which are missing in b, both leaves and wiring, required to create an instance of l. By specifying a vertex set U in the interface (U, m), we have fixed the size of bunch which can fit in

a hole. Current work promises to relax this condition, by allowing contexts to retain their trail components but to be *polymorphic*, in that they apply to holes of any size. We do not treat this generalisation here.

7 Example 2: Term-Rewriting and Multi-Hole Contexts

Our second example is motivated by the nesting of agents, which occurs in its most familiar form in term-rewriting. In Section 4 we argued that pushouts do not exist for free term contexts, thus motivating the exploration of RPOs in the abstract. We now address the specific properties of RPOs for term contexts. If we apply our theory to the category of one-hole contexts, then RPOs exist, as a corollary of Sewell's dissection result for terms (Lemma 1 in [22]). Consequently, all the definitions of labelled transition in Section 5 induce bisimulation equivalences that are congruences for term rewriting systems. The resulting labels are unnecessarily heavy, though. For consider the reaction rule $(\gamma(\alpha), \alpha')$; we have $\alpha \xrightarrow{\gamma(-)} \alpha'$ which corresponds to our intuition that α needs $\gamma(-)$ to react. Unfortunately, we also have a labelled transition where the label contains a complete copy of the redex:

$$\alpha' \xrightarrow{\beta(-,\gamma(\alpha))} \beta(\alpha', \alpha') \quad .$$

To attack this problem we consider multi-hole contexts where we shall find that this transition is prohibited. We then modify the definition of labelled transition and the statement of the congruence theorem to cater explicitly for multi-hole contexts in any reactive system. We end by asserting that multi-hole term contexts satisfy the hypotheses of this new congruence theorem.

Definition 13 (multi-hole term contexts). *Given a* signature Σ *of function symbols then the category of* multi-hole term contexts $\mathbf{T}^*(\Sigma)$ *over* Σ *is constructed as follows: the objects are the natural numbers; an arrow* $j \to k$ *is a* k*-tuple of terms over the signature* $\Sigma \cup \{-_1, \ldots, -_j\}$ *containing exactly one use of each hole* $-_i$ *(*$1 \leq i \leq j$*). The* identities *are:* $\mathsf{id}_j \triangleq \langle -_1, \ldots, -_j \rangle$*. For* $f = \langle a_1, \ldots, a_k \rangle : j \to k$ *and* $g : k \to m$*, their* composition *is the substitution*

$$gf \triangleq \{a_1/-_1, \cdots, a_k/-_k\}g \quad .$$

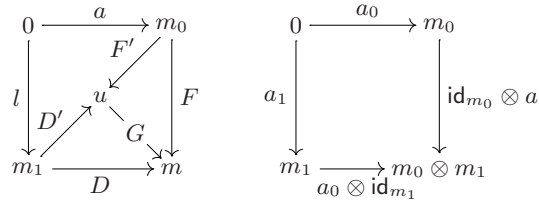
(When $j = 1$ we write $-_1$ as $-$.)

We now refine the abstract definitions of reactive system and of labelled transition to cater explicitly for multi-hole contexts in *any* reactive system, not just $\mathbf{T}^*(\Sigma)$. This refinement is part of our programme to express abstractly the phenomena of real reactive systems.

We require the notion of a *strict monoidal category* $(\mathbf{C}, \otimes, 0)$, a category $\mathbf{C}$ equipped with a functor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ and an object 0 such that $\otimes$ is strictly associative and has unit 0.

The role of the tensor $\otimes$ in the definition of reactive system is to "tuple" objects (e.g. $1 \otimes 1 = 2$ in $\mathbf{T}^*(\Sigma)$) and arrows.

Definition 14 (multi-hole reactive system). *A* reactive system *consists of a strict monoidal category* $(\mathbf{C}, \otimes, 0)$ *and the following added structure:*



8(1) 8(2)

Figure 8. Redex-RPOs and tensor-IPOs

- *a subset* Z *of objects (we use* $m, m', \ldots$ *to range over* Z*);*
- *a set* $\mathsf{Reacts} \subseteq \bigcup_{m \in Z} \mathbf{C}(0,m)^2$ *of* reaction rules*;*
- *a subcategory* $\mathbf{D}$ *of* $\mathbf{C}$*, whose arrows are the* reactive contexts*, with two properties:* $D_1 D_0 \in \mathbf{D}$ *implies* $D_1, D_0 \in \mathbf{D}$*; and* $a \otimes \mathsf{id}_m \in \mathbf{D}$ *for* $a : 0 \to m'$*.*

The *agents* of a reactive system are arrows $0 \to m$ and the *agent contexts* are arrows $m \to m'$, for $m, m' \in Z$. (Thus, for example, in $\mathbf{T}^*(\Sigma)$, we take $Z = \{1\}$ to mark out the singleton terms.) We adapt the definition of labelled transition:

Definition 15 (labelled transition). $a \xrightarrow{F} a'$ *iff* a, a' *are agents,* F *an agent context, and there exists* $(l, r) \in \mathsf{Reacts}$ *and* $D \in \mathbf{D}$ *such that Fig. 4(1) is an IPO and* $a' = Dr$*.*

Two conditions replace "redex-RPOs" (Def. 7):

Definition 16 (redex-RPOs). $\mathbf{C}$ has all redex-RPOs *if for all* $(l, r) \in \mathsf{Reacts}$ *and arrows* a, F, D*, where* a *is an agent,* F, D *agent contexts,* $D \in \mathbf{D}$*, and* $Fa = Dl$*, then the square in Fig. 8(1) has an RPO, as shown, such that either* $u \in Z$*, or there exists an isomorphism* $k : u \to m_0 \otimes m_1$ *such that* $kF' = \mathsf{id}_{m_0} \otimes l$ *and* $kD' = a \otimes \mathsf{id}_{m_1}$*.*

Definition 17 (tensor-IPOs). $\mathbf{C}$ has all tensor-IPOs *if Fig. 8(2) is an IPO square for all* $a_i : 0 \to m_i$ *with* $m_i \in Z$ *and* $i = 0, 1$*.*

Theorem 5 (congruence). *If* $\mathbf{C}$ *has all redex-RPOs and all tensor-IPOs then* $\sim$ *is preserved by all agent contexts.*

Let us return to the category $\mathbf{T}^*(\Sigma)$ and see how we may apply Theorem 5 to it. We have a choice of Z; we here confine ourselves to the case $Z = \{1\}$. Also, we may choose any subcategory of $\mathbf{T}^*(\Sigma)$ to be the reactive contexts, subject to the conditions in Def. 14. Then

- an agent of $\mathbf{T}^*(\Sigma)$ is a term $a : 0 \to 1$;
- an agent context of $\mathbf{T}^*(\Sigma)$ is a term context $C : 1 \to 1$, i.e. a term containing a single hole.

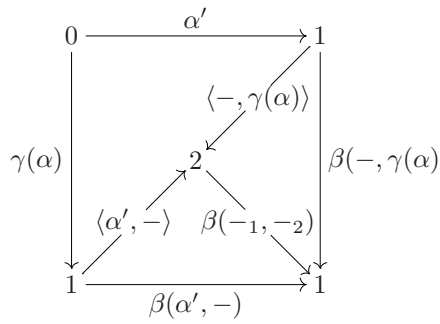
The labels of $\mathbf{T}^*(\Sigma)$ depend, of course, on the reaction rules. Once these are specified, we have determined the labelled transition relation $\xrightarrow{F}\!\triangleright$ over $\mathbf{T}^*(\Sigma)$, and hence the induced bisimulation $\sim$. Formally we have:

Theorem 6. *If we take $Z = \{1\}$ then $\mathbf{T}^*(\Sigma)$ has all redex-RPOs and all tensor-IPOs. Hence from Theorem 5 the induced bisimilarity $\sim$ over $\mathbf{T}^*(\Sigma)$ is preserved by all term contexts.*

Let us now revisit the reactive system whose only reaction rule is $(\gamma(\alpha), \alpha')$. It contains exactly the following labelled transitions:

$$D(\gamma(\alpha)) \xrightarrow{-}\!\triangleright D(\alpha') \qquad \text{for all reactive term contexts } D$$
$$\alpha \xrightarrow{\gamma(-)}\!\triangleright \alpha'$$

These agree with the transitions found by Sewell in the case of ground-term rewriting. We believe that the labels in our reactive system $\mathbf{T}^*(\Sigma)$ coincide exactly with Sewell's. Note that the heavy transition mentioned earlier is absent. We indicate why this is so with the help of the diagram below.



If we work in the category of *one-hole* contexts then the outer square is an IPO, which gives rise the transition $\xrightarrow{\beta(-,\gamma(\alpha))}\!\triangleright$ mentioned earlier. By admitting multi-hole contexts we have given the outer-square a simpler RPO.

Note also that, though working with multi-hole contexts, we have been free to choose the set Z as small as we wish, and thus obtain a simpler requirement for the existence of RPOs.

8 Current and Future Work

We have presented here a single key idea which allows labelled transitions, and thence behavioural congruences, to be derived for reactive systems. This is part of a larger programme of work, aiming at a theory of behavioural equivalence relevant to calculi designed for a wide range of practical purposes.

We are at present identifying those action calculi, or subcategories thereof, which possess RPOs; this will greatly clarify the status of action calculi as a framework. In particular, sharing graphs [1,11] and Gardner's closed action calculi [8] should be addressed. We also have to develop the notion of "polymorphic" context alluded to in Section 6.

Calculi already equipped with LTSs and congruence proofs need to be checked to see how close our uniformly derived LTSs and equivalences come to theirs. We would like to study in generality the situation in which redexes themselves are contexts, as Sewell [22] has done for term rewriting with parallel composition. Jeffrey and Rathke [13] have recently studied the relationship between contextual equivalence and labelled transitions for the ν-calculus of Pitts and Stark [20]; this will provide a good test for our uniform derivation of LTSs. We do not expect yet to achieve the fine-tuning present in some calculi; but we see no obvious limit to what can be achieved using general categorical (and other) methods as distinct from working in each individual calculus. More generally, links with other lines of research must be explored. Our method does not appear to overlap with the categorical approach by Joyal, Nielsen and Winskel [14] in defining bisimulation from open maps, but one should attempt to integrate the category-theoretic study of LTS-based equivalences. Categorical methods have also been productive in graph-rewriting; for example, in 1991 Corradini and Montanari [6] were already combining categorical and algebraic methods in concurrent graph-rewriting. Such work has developed further and should be related to ours.

In the longer term, we believe that graphical models represent the best hope for a theory of interactive systems – including the internet – which design engineers and analysts can actually use, with the help of computerised visualisation backed by rigorous machine-assisted verification. We hope the present work will provide part of the necessary theoretical background for this development.

Acknowledgements

The authors thank Luca Cattani, Philippa Gardner, Georges Gonthier, Martin Hyland, Ole Jensen, Jean-Jacques Lévy, Andrew Pitts, and Peter Sewell for stimulating discussions, and the anonymous referees for their helpful comments. Leifer was supported by an NSF Graduate Research Fellowship and a Trinity College Senior Rouse Ball Studentship.

References

1. Ariola, Z. M. and Klop, J. W., Equational term graph rewriting. Fundamentae Informaticae, 26(3,4), pp. 207–240, 1996. 256
2. Abadi, M. and Gordon, A. D., A calculus for cryptographic protocols: the spi calculus. Proc. Fourth ACM Conf. on Computer and Communications Security, Zürich, ACM Press, pp. 36–47, 1997.
3. Baeten, J. C. and Weiland, W. P., Process algebra. CUP, 1990. 244
4. Berry, G. and Boudol, G., The chemical abstract machine. Theor. Comp. Sci. 96, pp. 217–248, 1992. 244
5. Cardelli, L. and Gordon, A. D., Mobile ambients. Foundations of System Specification and Computational Structures, LNCS 1378, pp. 140–155, 1998.
6. Corradini, A. and Montanari, U., An algebra of graphs and graph rewriting. Proc. Fourth Biennial Conf. on Category Theory and Computer Science, LNCS 530, pp. 236–260, 1991. 257

7. Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D., A calculus of mobile agents. Proc. CONCUR'96, LNCS 1119, pp. 406–421, 1996.
8. Gardner, P., Closed action calculi. Theor. Comp. Sci. 228(1,2), pp. 77–103, 1999. 256
9. van Glabbeek, R. J., The linear time - branching time spectrum. Proc. CONCUR'90, LNCS 458, pp. 278–297, 1990. 244
10. Groote, J. F. and Vaandrager, F. W., Structural operational semantics and bisimulation as a congruence. Information and Computation 100(2), pp. 202–260, 1992. 244
11. Hasegawa, M. Models of sharing graphs (a categorical semantics of let and letrec). PhD thesis, LFCS, University of Edinburgh, 1997. 256
12. Hoare, C. A. R., Communicating Sequential Processes. Prentice Hall, 1985. 244
13. Jeffrey, A. and Rathke, J., Towards a theory of bisimulation for local names. Proc. LICS'99, IEEE Press, pp. 56–66, 1999. 257
14. Joyal, A., Nielsen, M. and Winskel, G., Bisimulation from open maps. Information and Computation 127(2), pp. 164–185, 1996. 257
15. Milner, R., Communication and Concurrency. Prentice Hall, 1989. 244, 245, 251
16. Milner, R., Functions as processes. Mathematical Structures in Computer Science 2(2), pp. 119–141, 1992. 244
17. Milner, R., Calculi for interaction. Acta Informatica 244, 245, 252
33(8), pp. 707–737, 1996.
18. Milner, R., Parrow, J. and Walker, D., A calculus of mobile processes, Parts 1 and 2. Information and Computation 100(1), pp. 1–77, 1992. 244
19. Milner, R. and Sangiorgi, D., Barbed bisimulation. Proc. ICALP'92, LNCS 623, pp. 685–695, 1992. 244
20. Pitts, A. M. and Stark, I. D. B., Observable properties of higher order functions that dynamically create local names, or: What's new? Proc. MFCS, LNCS 711, pp 122–141, 1993. 257
21. Park, D., Concurrency and automata on infinite sequences. LNCS 104, 1980. 249
22. Sewell, P., From rewrite rules to bisimulation congruences. Proc. CONCUR'98, LNCS 1466, pp. 269–284, 1998. [Revised version to appear in a special issue of Theor. Comp. Sci.] 244, 245, 246, 254, 257
23. Sewell, P., Global/local subtyping and capability inference for a distributed pi-calculus. Proc. ICALP'98, LNCS 1443, pp. 695–706, 1998.
24. Turi, D. and Plotkin, G. Towards a mathematical operational semantics. Proc. LICS'97, IEEE Press, pp. 280–291, 1997. 244

Bisimilarity Congruences for Open Terms and Term Graphs via Tile Logic*

Roberto Bruni[1], David de Frutos-Escrig[2], Narciso Martí-Oliet[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italia
{bruni,ugo}@di.unipi.it
[2] Dept. Sistemas Informáticos y Programación, Univ. Complutense de Madrid, Spain
{defrutos,narciso}@sip.ucm.es

Abstract. The definition of SOS formats ensuring that bisimilarity on closed terms is a congruence has received much attention in the last two decades. For dealing with open terms, the congruence is usually lifted from closed terms by instantiating the free variables in all possible ways; the only alternatives considered in the literature are Larsen and Xinxin's *context systems* and Rensink's *conditional transition systems*. We propose an approach based on *tile logic*, where closed and open terms are managed uniformly, and study the 'bisimilarity as congruence' property for several tile formats, accomplishing different concepts of open system.

1 Introduction

The semantics of many languages can be conveniently expressed via *labelled transition systems* (LTSs) whose states are terms over a certain signature and whose labels give some abstract information about system evolution. If such information is sufficient to model the interactions between the various (sub)systems when they are composed through the operators of the language, then the specification can be assembled compositionally. Plotkin's *structural operational semantics* (SOS) [18] is surely one of the most successful frameworks for LTS descriptions, where transitions are defined by induction on the structure of states; then, one is interested in equating all those states with the same behavior. In this paper we rely on *bisimulation* equivalence [17]: two states are equivalent if every action of the one can be simulated by the other, still ending in equivalent states. The need of a compositional framework, where each subcomponent of a state can be safely replaced by any equivalent subcomponent without affecting the overall behavior, motivated a lot of efforts in the definition of SOS formats (e.g., *De Simone* [8], GSOS [1], `tyft/tyxt` [12]) whose syntax guarantees that *bisimilarity is a congruence* — bisimilarity meaning the largest bisimulation.

* Research supported by CNR Integrated Project *Progettazione e Verifica di Sistemi Eterogenei*, Esprit WG *CONFER2* and *COORDINA*, MURST project *TOSCA*, and CICYT project *Desarrollo Formal de Sistemas Distribuidos* (TIC97–0669–C03–01).

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 259–274, 2000.

However, the congruence is usually defined on closed terms only; it can be extended to *contexts* (terms with variables) by letting two contexts be equivalent if they are equivalent under all possible instantiations of their arguments. This yields the coarsest conservative extension of the equivalence that is preserved by instantiation and, to some extent, is analogous to reducing an equivalence on closed terms to the congruence respected by all contexts.

When designing *open systems*, the preservation of a given equivalence under all possible instantiations should of course be a necessary property; however, it is not very practical to define it by universal quantification on all possible instantiations, since proofs of properties will become difficult. It would be preferable to extend the bisimulation game from closed states to contexts, providing a uniform framework. Thus, the variables of a context C can be viewed as forming its *input interface*, used by C for communicating with its arguments. Then, the label of a transition with source C must include some information about the moves accomplished by each argument for allowing the context to evolve. Moreover, for nonlinear contexts, the degree of subterm sharing may make a difference. In fact, for partially specified components it is convenient to distinguish between the situation in which different copies of a nondeterministic resource are available, which can evolve independently, and the situation in which the resource is shared between many components (and thus the resource must offer the same behavior to all its users). Conceptually, this originates two kinds of open systems that we call *incomplete systems* and *coordinators*.

A first situation would be the case in which variables represent software components possibly used by several processes. For example, one can take a CCS-like context $C[x]|D[x]$, with x a process variable, where the same protocol specification should be employed by both C and D to provide a certain service. In such a case, the two instances of x in $C[x]|D[x]$ must be instantiated with two copies of the same pattern (a suitable agent), which can progress independently.

On the other hand, if the process $C[x]|D[x]$ represents a coordinator which regulates the execution of the argument x then, when x is instantiated to a process q, replication must be avoided: both C and D become connected to *the same* agent q. In this case the two occurrences of q must evolve in the same way and at the same time. Notationally this situation can be expressed by writing **let** $x = q$ **in** $C[x]|D[x]$ instead of $C[q]|D[q]$. A concrete example for this situation is when several consumers receive information from a single producer by means of possibly replicated handshaking channels: We desire all the consumers to receive the same information through all their channels at the same time.

Ordinary SOS formats do not seem suitable to deal with this latter view in the correct way. For example, let us consider the well-known SOS specification for Milner's finite CCS [17], whose rules are in De Simone format, and take the contexts $C[x] = x\backslash_\alpha|x\backslash_\alpha$ and $D[x] = (x|x)\backslash_\alpha$. Then, the two contexts cannot react as coordinators to any move of x and thus they are equivalent according to bisimilarity (since x is shared, each move of x is replicated along its two occurrences and it is not possible to have complementary actions). However,

if we instantiate x with the process $p = \alpha.nil + \bar{\alpha}.nil$, then $C[p]$ is no longer bisimilar to $D[p]$, because the former is stuck and the second can make a τ.

These two cases do not exhaust all the possible ways to treat nonlinear contexts, but they give us a motivation to study different ways to cope with the problem: (1) considering only linear contexts; (2) allowing free duplication of shared components, as if, e.g., the CCS context $(x|x)\backslash_\alpha$ could make a transition to $(x_1|x_2)\backslash_\alpha$, disconnecting the shared resource, and observing that its argument x has been duplicated; (3) employing *term graphs* [9] instead of terms to describe states. The first approach avoids sharing at all. The second proposal enriches the observation algebra. The third proposal is especially attractive for the specification and analysis of distributed systems. In fact, term graphs are finer than terms and allow one to express the sharing of closed subterms, hence distinguishing, e.g., the coordinated system **let** $x = p$ **in** $(x|x)\backslash_\alpha$ from $(p|p)\backslash_\alpha$.

As far as we know, the extension of good SOS formats to incomplete systems has been addressed only recently by Rensink [19], who exploited a previous idea of De Simone. Rensink formalized several extensions to open terms of the bisimilarity on closed terms, based on the notion of *conditional transition systems*. We propose instead the use of *tile systems*, where the extensions (1)–(3) discussed above can be straightforwardly handled, at the levels of both specifications and computational models. The *tile model* [11] is a formalism for modeling open compositional systems. It relies on certain rewrite rules with side effects, called *basic tiles*, which are reminiscent of both SOS rules and *context systems* [14]. It collects intuitions coming from *structured transition systems* [7] and *rewriting logic* (RL) [15], and by analogy with RL, the tile model has a logical presentation, called *tile logic*, where tiles are seen as sequents subject to certain inference rules. Tiles are written $\alpha\colon s \xrightarrow[b]{a} t$, where s and t can be open terms, stating that the *initial configuration* s evolves to the *final configuration* t producing the *effect* b. However, such a step is allowed only if the subcomponents supplied as arguments to s evolve to the subcomponents of t, producing the observation a, which is the *trigger* of the tile α. Triggers and effects are called *observations*.

In this paper, we want to stress that tiles are designed for open systems, and in fact, the notion of bisimulation can be generalized to that of *tile bisimulation*, which directly operates over contexts. More precisely, we investigate tile formats guaranteeing that tile bisimilarity is a congruence. The first tile format appeared in the literature is the *algebraic tile format* (ATF) [10,11]. It has a cartesian structure of configurations and a monoidal structure of observations.[1] We show that the ATF is not completely satisfactory for our aims, due to the different structure of configurations and observations. We focus instead on three tile formats that reflect the approaches — linearity, free duplication and explicit

[1] Essentially, cartesian configurations can be seen as substitutions over a given signature (e.g., substitution $[t_1(x_1,\ldots,x_n)/y_1,\cdots,t_m(x_1,\ldots,x_n)/y_m]$ corresponds to a configuration $t\colon n \to m$). In monoidal configurations terms $t_1,\ldots,t_m$ are linear in their variables. In gs-monoidal configurations, sharing of subterms is possible, but it cannot be eliminated by copying, i.e., **let** $x = t_1$ **in** t_2 is different from $t_2[t_1/x]$ whenever x does not occur, or occurs more than once, in t_2.

sharing — discussed above: (1) the *monoidal tile format* (MTF) [16], whose configurations and observations have a monoidal structure; (2) the *term tile format* (TTF) [4], whose configurations and observations have a cartesian structure; (3) the *gs-monoidal tile format* (GSTF, from graph substitution), whose configurations have a gs-monoidal structure and observations have a monoidal structure.

By imposing the so called *basic source* constraint (initial configurations of basic tiles must consist of a basic operator) on the ATF, one can recover a slightly more general format than De Simone but stricter than GSOS. In this case, tile bisimilarity recovers the ordinary congruences for closed terms, but not for contexts. The basic source on MTF and TTF yields respectively De Simone and a format more general than (positive) GSOS. In both cases, tile bisimilarity is a congruence also for open terms. The GSTF is hard to recast in existing SOS formats, due to its treatment of subterm sharing. We are indeed confident that GSTF can find a meaningful application in the modeling of systems with shared resources. Again, the basic source guarantees that tile bisimilarity is a congruence.

Outline. In Section 2 we fix the notation and recall the notions of bisimulation and tile bisimulation, while in Section 3 we review the most common SOS formats and several tile formats. In the presentation, category theory and algebraic theories are employed in a mild way. Sections 4, 5 and 6 deal with MTF, TTF and GSTF respectively, showing the main results of the paper, namely that for all of them the basic source implies that tile bisimilarity is a congruence. Due to space limitation, we omit all proofs, which can be found in the technical report [3]. Section 7 compares our approach with Rensink's proposal in [19].

2 Tiles and Bisimulation

Notation. To ease the presentation we will consider only one-sorted signatures, though our results extend to the many-sorted case. A *one-sorted signature* is a set of *operators* Σ together with an arity function $ar\colon \Sigma \to \mathbb{N}$. For $n \in \mathbb{N}$, we let $\Sigma_n = \{f \in \Sigma \mid ar(f) = n\}$. Operators in Σ_0 are called *constants*. We denote by $\mathbb{T}_\Sigma(X)$ the term algebra over Σ and variables in X (disjoint from Σ), with $\mathbb{T}_\Sigma = \mathbb{T}_\Sigma(\varnothing)$. For $t \in \mathbb{T}_\Sigma(X)$ we denote by $var(t)$ the set of variables that appear in t. If $var(t) = \varnothing$ then t is called *closed*, otherwise *open*. A term is *linear* if each variable occurs at most once in it.

A *substitution* is a mapping $\sigma\colon X \to \mathbb{T}_\Sigma(X)$. It is closed if each variable is mapped into a closed term. Substitutions uniquely extend to mappings from terms to terms as usual: $\sigma(t)$ is the term obtained by simultaneously replacing all occurrences of variables x in t by $\sigma(x)$. The substitution mapping x_i to t_i for $i \in [1, n]$ is denoted by $[t_1/x_1, \ldots, t_n/x_n]$, and it is *linear* if each t_i is linear and $var(t_i) \cap var(t_j) = \varnothing$ for $i \neq j$. A substitution σ' can be applied elementwise to $\sigma = [t_1/x_1, \ldots, t_n/x_n]$ yielding $\sigma;\sigma' = [\sigma'(t_1)/x_1, \ldots, \sigma'(t_n)/x_n]$.

A *context* $t = C[x_1, \ldots, x_n]$ denotes a term in which at most the distinct variables $x_1, \ldots, x_n$ appear. The term $C[t_1, \ldots, t_n]$ is then obtained by applying the substitution $[t_1/x_1, \ldots, t_n/x_n]$ to the context $C[x_1, \ldots, x_n]$, which can thus be regarded as a function from n arguments to 1. Note that the x_i may as well not

appear in $C[x_1, \ldots, x_n]$. For example, the context $x_2[x_1, x_2, x_3]$ is a substitution from three arguments to one, which is the projection on the second argument.

Bisimulation. A *labelled transition system* (LTS) is a triple $L = (S, \Lambda, \rightarrow)$, where S is a set of *states*, Λ is a set of *labels*, and $\rightarrow \subseteq S \times \Lambda \times S$. We let $s, t, \ldots$ range over S and $a, b, c, \ldots$ range over Λ. For $\langle s, a, t\rangle \in \rightarrow$ we say that s is the *source*, t is the *target* and a is the *observable*, and use the notation $s \xrightarrow{a} t$.

Definition 1. *Given an* LTS $L = (S, \Lambda, \rightarrow)$*, a* bisimulation *on L is a symmetric, reflexive relation $\sim \subseteq S \times S$ such that if $s \sim t$ then for any transition $s \xrightarrow{a} s'$ there exists a transition $t \xrightarrow{a} t'$ such that $s' \sim t'$. We denote by $\simeq$ the largest bisimulation, called* bisimilarity*, and we say that two states s and t are* bisimilar *whenever $s \simeq t$, i.e., whenever there exists a bisimulation $\sim$ such that $s \sim t$.*

When $S = \mathbb{T}_\Sigma$, an essential property of bisimilarity is *compositionality* w.r.t. operators in Σ, guaranteeing that operationally equivalent subsystems can be safely replaced in any context. This amounts to requiring that $\simeq$ is a congruence, i.e., that for all $f \in \Sigma$, $s_i \simeq t_i$ for $i \in [1, ar(f)]$ implies $f(s_1, \ldots, s_{ar(f)}) \simeq f(t_1, \ldots, t_{ar(f)})$. Bisimilarity can be lifted to contexts by letting $C[x_1, \ldots, x_n] \simeq D[x_1, \ldots, x_n]$ if for all $t_1, \ldots, t_n \in \mathbb{T}_\Sigma$ we have $C[t_1, \ldots, t_n] \simeq D[t_1, \ldots, t_n]$.

Tile bisimulation. The point of view of the tile model [11] is that bisimulation can be defined uniformly on both closed terms and contexts, without resorting to instantiation closure. In the following we use monoidal categories for providing an abstract presentation of tile systems. As a matter of notation, sequential composition and monoidal tensor product are denoted by $\_;\_$ and $\_ \otimes \_$, respectively. The unfamiliar reader may consult, e.g., [11,2].

A *tile system* is a tuple $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ where $\mathcal{H}$ and $\mathcal{V}$ are monoidal categories with the same set of objects $O_\mathcal{H} = O_\mathcal{V}$, N is the set of rule names and $R\colon N \rightarrow \mathcal{H} \times \mathcal{V} \times \mathcal{V} \times \mathcal{H}$ is a function such that for all $\alpha \in N$, if $R(\alpha) = \langle s, a, b, t\rangle$ then $s\colon x \rightarrow y$, $a\colon x \rightarrow z$, $b\colon y \rightarrow w$, and $t\colon z \rightarrow w$ for suitable objects x, y, z and w. We will write such rule α either as $\alpha\colon s \xrightarrow[b]{a} t$, or as the tile

$$\begin{array}{ccc} x & \xrightarrow{\ s\ } & y \\ {\scriptstyle a}\downarrow & \alpha & \downarrow{\scriptstyle b} \\ z & \xrightarrow[\ t\ ]{} & w \end{array}$$

The category $\mathcal{H}$ is called *horizontal* and its arrows are called *configurations*. The category $\mathcal{V}$ is called *vertical* and its arrows are called *observations*. The objects of $\mathcal{H}$ and $\mathcal{V}$ are called *interfaces*. Starting from the basic tiles, more complex tiles can be constructed by means of horizontal, vertical and parallel composition. Moreover, the horizontal and vertical identities are always added and composed together with the basic tiles. All this is illustrated in Figure 1.

Depending on the chosen tile format (see Section 3), $\mathcal{H}$ and $\mathcal{V}$ can be specialized (e.g., to cartesian categories) and suitable *auxiliary tiles* are added and composed with basic tiles and identities in all the possible ways. The set of resulting tiles (also called *flat sequents*) define the *flat tile logic* associated to $\mathcal{R}$.

$$\frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[c]{b} f}{s;h \xrightarrow[c]{a} t;f} \qquad \frac{s \xrightarrow[b]{a} t \quad t \xrightarrow[d]{c} h}{s \xrightarrow[b;d]{a;c} h} \qquad \frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[d]{c} f}{s \otimes h \xrightarrow[b \otimes d]{a \otimes c} t \otimes f} \qquad \frac{t : x \to y \in \mathcal{H}}{t \xrightarrow[y]{x} t} \qquad \frac{a : x \to z \in \mathcal{V}}{x \xrightarrow[a]{a} z}$$

Fig. 1. Composition and identity rules for tile logic

$$\text{(a)}\ \frac{\{s_i \xrightarrow{a_i} t_i \mid i \in I\}}{s \xrightarrow{a} t} \qquad \frac{\{x_i \xrightarrow{a_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_n) \xrightarrow{a} D[y_1, \ldots, y_n]}\ \text{(b)}$$

Fig. 2. SOS schema (a), and De Simone format (b)

We say that $s \xrightarrow[b]{a} t$ is *entailed* by the logic, written $\mathcal{R} \vdash s \xrightarrow[b]{a} t$, if the sequent $s \xrightarrow[b]{a} t$ can be expressed as the composition of basic and auxiliary tiles.

Definition 2. *Let $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ be a tile system. A symmetric relation $\sim_{\mathrm{t}}$ on configurations is called* tile bisimulation *if whenever $s \sim_{\mathrm{t}} t$ and $\mathcal{R} \vdash s \xrightarrow[b]{a} s'$, then there exists t' such that $\mathcal{R} \vdash t \xrightarrow[b]{a} t'$ and $s' \sim_{\mathrm{t}} t'$. The maximal tile bisimulation is called* tile bisimilarity *and denoted by $\simeq_{\mathrm{t}}$.*

An interesting question concerns suitable conditions under which tile bisimilarity yields a congruence (w.r.t. the operations of the underlying horizontal structure); two main properties have been investigated: *basic source* and *tile decomposition* [11]. The former is strongly related to tile formats and will be discussed later. Tile decomposition has a completely abstract formulation that applies to all tile systems, and it will be very useful in our congruence proofs.

Definition 3. *A tile system $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ enjoys the* decomposition property *if for all arrows $s \in \mathcal{H}$ and for all sequents $s \xrightarrow[b]{a} t$ entailed by $\mathcal{R}$, then: (1) if $s = s_1; s_2$ then $\exists c \in \mathcal{V}$, $t_1, t_2 \in \mathcal{H}$ such that $\mathcal{R} \vdash s_1 \xrightarrow[c]{a} t_1$, $\mathcal{R} \vdash s_2 \xrightarrow[b]{c} t_2$ and $t = t_1; t_2$; (2) if $s = s_1 \otimes s_2$ then $\exists a_1, a_2, b_1, b_2 \in \mathcal{V}$, $t_1, t_2 \in \mathcal{H}$ such that $\mathcal{R} \vdash s_1 \xrightarrow[b_1]{a_1} t_1$, $\mathcal{R} \vdash s_2 \xrightarrow[b_2]{a_2} t_2$, $a = a_1 \otimes a_2$, $b = b_1 \otimes b_2$ and $t = t_1 \otimes t_2$.*

This property characterizes compositionality: It amounts to saying that if a system s can undergo a transition α, then for every subsystem s_1 of s there exists some transition α', such that α can be obtained by composing α' with a transition of the rest.

Proposition 1 (cf. [11]). *If $\mathcal{R}$ enjoys decomposition, then $\simeq_{\mathrm{t}}$ is a congruence.*

3 Relating SOS and Tile Formats

SOS formats. LTSs over $\mathbb{T}_\Sigma$ and Λ can be more conveniently specified via a collection of inductive proof rules, called *transition system specification* (TSS).

Such rules have the form in Figure 2(a), where the s_i, t_i, s and t are in $\mathbb{T}_\Sigma(X)$ and the a_i, a are in Λ. The generated LTS has set of states $\mathbb{T}_\Sigma$ and all transitions $s \xrightarrow{a} t$ that can be proved by using the TSS rules. In the SOS approach [18] the behavior of s is defined in terms of the behaviors of its subterms, but in general this is not enough to guarantee that bisimilarity is a congruence. To ensure this important property, suitable rule formats have been defined. For example, in the De Simone format (DSF) [8] rules have the form in Figure 2(b), where $f \in \Sigma_n$, $I \subseteq \{1, \ldots, n\}$, the context D is linear and the variables x_i and y_i are distinct, except for $y_i = x_i$ if $i \notin I$. Hence each x_i can be used at most once in the premises, and if used cannot appear in D. The (positive) GSOS format [1] extends the DSF in several ways (e.g., the same x_i can appear more than once in the premises and also in the D, which can be nonlinear). The `tyft/tyxt` format [12] generalizes GSOS by allowing generic terms t_i as sources of the premises. However, in all such formats, the main requirement is that rule conclusions must have the form $f(x_1, \ldots, x_n) \xrightarrow{a} t$, i.e., their sources must consist of a single $f \in \Sigma_n$ applied to n different variables, a crucial fact in the proof that bisimilarity is a congruence.

Tile formats. Since we are particularly interested in considering tile systems where configurations and observations are *freely generated* by the horizontal signature Σ and by (the signature associated to) the set of labels Λ, in what follows we shall present tile systems as tuples of the form $\mathcal{R} = (\Sigma, \Lambda, N, R)$. In particular, we employ categories of substitutions on Σ and Λ. In fact, substitutions and their composition $\_;\_$ form a cartesian category $\mathbf{Subs}_\Sigma$, with linear substitutions forming a monoidal subcategory. An alternative presentation of $\mathbf{Subs}_\Sigma$ can be obtained resorting to *algebraic theories* [13]. The free algebraic theory associated to a signature Σ is the category $\mathbf{Th}[\Sigma]$: Its objects are 'underlined' natural numbers, the arrows from $\underline{m}$ to $\underline{n}$ are n-tuples of terms in the free Σ-algebra with (at most) m variables, and composition of arrows is term substitution. The arrows of $\mathbf{Th}[\Sigma]$ are generated from Σ by the inference rules in Figure 3, modulo the axioms in Table 1. It is folklore that $\mathbf{Th}[\Sigma]$ is isomorphic to $\mathbf{Subs}_\Sigma$, and the arrows from $\underline{0}$ to $\underline{1}$ are in bijective correspondence with the closed terms over Σ.

An object $\underline{n}$ (interface) can be thought of as representing the n (ordered) variables $x_1, \ldots, x_n$. This allows us to denote $[t_1/x_1, \ldots, t_n/x_n]$ just by the tuple $\langle t_1, \ldots, t_n \rangle$, since a standard naming of substituted variables is assumed. We omit angle brackets if no confusion can arise. The rule `op` defines basic substitutions $[f(x_1, \ldots, x_n)/x_1] = f(x_1, \ldots, x_n)$ for all $f \in \Sigma_n$. The rule `id` yields identity substitutions $\langle x_1, \ldots, x_n \rangle$. The rule `seq` represents application of α to β. The rule `mon` composes substitutions in parallel (in $\alpha \otimes \beta$, α goes from $x_1, \ldots, x_n$ to $x_1, \ldots, x_m$, while β goes from $x_{n+1}, \ldots, x_{n+k}$ to $x_{m+1}, \ldots, x_{m+l}$). Three 'auxiliary' operators (i.e., not dependent on Σ) are introduced that recover the cartesian structure (rules `sym`, `dup` and `dis`). The *symmetry* $\gamma_{\underline{n},\underline{m}}$ is the permutation $\langle x_{n+1}, \ldots, x_{n+m}, x_1, \ldots, x_n \rangle$. The *duplicator* $\nabla_{\underline{n}} = \langle x_1, \ldots, x_n, x_1, \ldots, x_n \rangle$ introduces sharing and hence nonlinear substitutions. The *discharger* $!_{\underline{n}}$ is the empty substitution on $x_1, \ldots, x_n$, recovering cartesian projections. Due to space limitations we cannot detail the axioms in Table 1 (consult, e.g., [2,4]).

$$\mathtt{op}\ \frac{f \in \Sigma_n}{f\colon \underline{n} \to \underline{1}} \qquad \mathtt{id}\ \frac{n \in \mathbb{N}}{id_{\underline{n}}\colon \underline{n} \to \underline{n}} \qquad \mathtt{seq}\ \frac{\alpha\colon \underline{n} \to \underline{m} \quad \beta\colon \underline{m} \to \underline{k}}{\alpha;\beta\colon \underline{n} \to \underline{k}} \qquad \mathtt{mon}\ \frac{\alpha\colon \underline{n} \to \underline{m} \quad \beta\colon \underline{k} \to \underline{l}}{\alpha \otimes \beta\colon \underline{n+k} \to \underline{m+l}}$$

$$\mathtt{sym}\ \frac{n, m \in \mathbb{N}}{\gamma_{\underline{n},\underline{m}}\colon \underline{n+m} \to \underline{m+n}} \qquad \mathtt{dup}\ \frac{n \in \mathbb{N}}{\nabla_{\underline{n}}\colon \underline{n} \to \underline{n+n}} \qquad \mathtt{dis}\ \frac{n \in \mathbb{N}}{!_{\underline{n}}\colon \underline{n} \to \underline{0}}$$

Fig. 3. The inference rules for the generation of $\mathbf{Th}[\Sigma]$

Table 1. Axiomatization of $\mathbf{Th}[\Sigma]$

category	$\alpha;(\beta;\delta) = (\alpha;\beta);\delta$	$\alpha; id_{\underline{m}} = \alpha = id_{\underline{n}};\alpha$
tensor product	$(\alpha;\alpha') \otimes (\beta;\beta') = (\alpha \otimes \beta);(\alpha' \otimes \beta')$	$id_{\underline{n+m}} = id_{\underline{n}} \otimes id_{\underline{m}}$
	$\alpha \otimes (\beta \otimes \delta) = (\alpha \otimes \beta) \otimes \delta$	$\alpha \otimes id_{\underline{0}} = \alpha = id_{\underline{0}} \otimes \alpha$
symmetries	$\gamma_{\underline{n},\underline{m+k}} = (\gamma_{\underline{n},\underline{m}} \otimes id_{\underline{k}});(id_{\underline{m}} \otimes \gamma_{\underline{n},\underline{k}})$	$\gamma_{\underline{n},\underline{0}} = id_{\underline{n}}$ $\quad \gamma_{\underline{n},\underline{m}};\gamma_{\underline{m},\underline{n}} = id_{\underline{n+m}}$
duplicators	$\nabla_{\underline{n+m}} = (\nabla_{\underline{n}} \otimes \nabla_{\underline{m}});(id_{\underline{n}} \otimes \gamma_{\underline{n},\underline{m}} \otimes id_{\underline{m}})$	$\nabla_{\underline{0}} = id_{\underline{0}}$
	$\nabla_{\underline{n}};(id_{\underline{n}} \otimes \nabla_{\underline{n}}) = \nabla_{\underline{n}};(\nabla_{\underline{n}} \otimes id_{\underline{n}})$	$\nabla_{\underline{n}};\gamma_{\underline{n},\underline{n}} = \nabla_{\underline{n}}$
discharger	$!_{\underline{n+m}} = !_{\underline{n}} \otimes !_{\underline{m}}$ $\quad !_{\underline{0}} = id_{\underline{0}}$	$\nabla_{\underline{n}};(id_{\underline{n}} \otimes !_{\underline{n}}) = id_{\underline{n}}$
naturality	$(\alpha \otimes \beta);\gamma_{\underline{m},\underline{l}} = \gamma_{\underline{n},\underline{k}};(\beta \otimes \alpha)$ $\quad \alpha;\nabla_{\underline{m}} = \nabla_{\underline{n}};(\alpha \otimes \alpha)$	$\alpha;!_{\underline{m}} = !_{\underline{n}}$

Monoidal theories relate to algebraic theories as linear substitutions relate to generic substitutions. This originates a subcategory $\mathbf{M}[\Sigma]$ of $\mathbf{Th}[\Sigma]$ whose arrows are those generated by rules `op`, `id`, `seq`, and `mon` in Figure 3 modulo the axioms of category and tensor product (first and second rows in Table 1). *Term graphs* [9] are in some sense situated between linear and cartesian terms, because they allow for explicit sharing and discharging, in such a way that this information is preserved by composition. In fact, it has been shown in [6] that by abandoning the naturality of ∇ and !, one obtains a *gs-monoidal* category $\mathbf{GS}[\Sigma]$ which is isomorphic to the category of (ranked) term graphs on Σ. For example, in $\mathbf{GS}[\Sigma]$ the composition $[t_1/x_1];C[x_1]$ can be written as **let** $x_1 = t_1$ **in** $C[x_1]$, with the convention that it evaluates to $C[t_1]$ when x_1 occurs exactly once in C.

The tile format proposed in the original presentation of tiles [11] is the so-called *algebraic tile format* (ATF) that recollects the perspective of most TSS: configurations are terms, and observations are the arrows of the monoidal category freely generated by labels (regarded as unary operators). Auxiliary tiles lift the horizontal cartesian structure to the horizontal composition of tiles. In the ATF basic tiles have the form in Figure 4(a), where the a_i and a can be either labels (viewed as arrows from $\underline{1}$ to $\underline{1}$) or identities and $s, t \in \mathbb{T}_\Sigma(\{x_1, \ldots, x_n\})$. The ATF corresponds to SOS rules as in Figure 4(b), where $I \subseteq \{1, \ldots, n\}$, C and D are contexts (that correspond to s and t in the tile), and all the y_i and x_i are different if $i \in I$, but $y_k = x_k$ whenever $k \notin I$. The correspondence follows since for all $s, t \in \mathbb{T}_\Sigma$ and for all $a \in \Lambda$, $\mathcal{R} \vdash s \xrightarrow[a]{id_{\underline{0}}} t$ holds if and only if $s \xrightarrow{a} t$ in the LTS associated to the SOS specification. Typical auxiliary tiles for the ATF

$$\frac{\{x_i \xrightarrow{a_i} y_i \mid i \in I\}}{C[x_1, \ldots, x_n] \xrightarrow{a} D[y_1, \ldots, y_n]}$$

Fig. 4. A generic ATF tile (a) and its SOS counterpart (b)

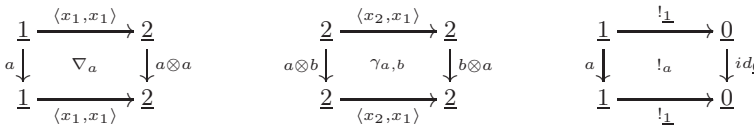
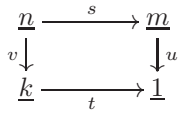


Fig. 5. Auxiliary tiles for ATF

are those in Figure 5: ∇_a duplicates the observation a (trigger of the tile) propagating it to two instances of the unique variable in the initial interface, while $\gamma_{a,b}$ swaps the subcomponents in the initial interface, together with their observations, and $!_a$ discharges the initial interface and its move a. We refer to [11] for more details.

In the ATF, $\mathcal{H}$ is cartesian, whereas $\mathcal{V}$ is only monoidal. We will show in Section 5 that this combination can compromise the tile bisimilarity as congruence property. In particular, it will be convenient either to consider the *term tile format* (TTF) [4], where also $\mathcal{V}$ is cartesian, or renounce the horizontal cartesian structure. This latter option can be achieved in (at least) two ways: either using the *monoidal tile format* (MTF) [16] where also $\mathcal{H}$ is monoidal, or resorting to the *gs-monoidal tile format* (GSTF), where a faithful account of subterm sharing is provided. The MTF deals only with linear terms and has no auxiliary tiles. Its basic tiles are similar to those of the ATF (but configurations must be linear). Though the auxiliary tiles of the GSTF are the same as those of the ATF, the former deals with term graphs rather than terms and thus the naturality axioms of ∇ and $!$ are not valid. Basic TTF tiles have the form:



with $s, t \in \mathbf{Th}[\Sigma]$ and $v, u \in \mathbf{Th}[\Lambda]$. If labels in Λ are regarded as unary operators, then $m = 1$. We present term tiles more concisely as sequents $n \triangleleft s \xrightarrow[u]{v} t$, where the number of variables in the 'upper-left' interface is explicit. Auxiliary term tiles consist of all tiles that perform consistent rearrangements in the two dimensions, i.e., tiles $n \triangleleft s \xrightarrow[u]{v} t$ such that $s; u = v; t$ with either (1) s, t, u and v are terms over the empty signature $\varnothing$ (hence $s, t, v, u \in \mathbf{Th}[\Sigma] \cap \mathbf{Th}[\Lambda]$); or (2) $s, t \in$

Table 2. Features of MTF, GSTF, ATF and TTF

	$\mathcal{H}$	$\mathcal{V}$	auxiliary tiles
monoidal tile format	$\mathbf{M}[\Sigma]$	$\mathbf{M}[\Lambda]$	none
gs-monoidal tile format	$\mathbf{GS}[\Sigma]$	$\mathbf{M}[\Lambda]$	$\gamma_{a,b}$, ∇_a, $!_a$
algebraic format	$\mathbf{Th}[\Sigma]$	$\mathbf{M}[\Lambda]$	$\gamma_{a,b}$, ∇_a, $!_a$
term tile format	$\mathbf{Th}[\Sigma]$	$\mathbf{Th}[\Lambda]$	$\gamma_{a,b}$, ∇_a, $!_a$, $\gamma_{s,t}$, ∇_t, $!_t$, $\sigma_{\underline{1},\underline{1}}$, $\tau_{\underline{1}}$, $\pi_{\underline{1}}$,...

$\mathbf{Th}[\Sigma]$ and $u, v \in \mathbf{Th}[\varnothing]$; or (3) $u, v \in \mathbf{Th}[\Lambda]$ and $s, t \in \mathbf{Th}[\varnothing]$. Typical auxiliary term tiles are $\pi_{\underline{1}} = 1 \triangleleft x_1 \xrightarrow[x_1,x_1]{x_1} x_1, x_1$ and $\tau_{\underline{1}} = 1 \triangleleft x_1, x_1 \xrightarrow[x_1,x_2]{x_1,x_1} x_1, x_2$ that duplicate the unary interface, $\sigma_{\underline{1},\underline{1}} = 2 \triangleleft x_2, x_1 \xrightarrow[x_1,x_2]{x_2,x_1} x_1, x_2$ that swaps the two components of the initial interface, and $\nabla_f = 2 \triangleleft f(x_1, x_2) \xrightarrow[\nabla_{\underline{1}}]{\nabla_{\underline{2}}} \langle f(x_1, x_2), f(x_3, x_4) \rangle$ that vertically duplicates the configuration $f(x_1, x_2)$ (see [4,2] for more details). Table 2 summarizes the differences between the tile formats we consider.

Our aim is to find syntactic constraints on basic tiles that enforce the 'tile bisimilarity as congruence' property. One such constraint, already noted in [11], is called *basic source* (it is reminiscent of analogous restrictions required by most well-behaved SOS formats): A tile system enjoys the basic source property if the initial configuration of each basic tile consists of a single operator $f\colon \underline{ar(f)} \to \underline{1}$.

Proposition 2 (cf. [11]). *If an algebraic tile system $\mathcal{R}$ enjoys the basic source property, then tile bisimilarity on closed terms is a congruence.*

3.1 Tiles and CCS

In this section, we recall the operational semantics of Milner's *calculus for communicating systems* (CCS) [17] and the tile systems proposed in the literature for recovering the behavior of (finite) agents. We let Δ (ranged over by α) be the set of *basic actions*, and $\bar{\Delta}$ the set of *complementary actions* (with $\bar{\_}$ an involutive function such that $\Delta = \bar{\bar{\Delta}}$ and $\Delta \cap \bar{\Delta} = \varnothing$). We denote $\Delta \cup \bar{\Delta}$ by Λ (ranged over by λ), let $\tau \notin \Lambda$ be a *distinguished* action, and let $Act = \Lambda \cup \{\tau\}$ (ranged over by μ) be the set of CCS actions. Then, CCS *processes* (also *agents*) are generated by the grammar: $P ::= nil \mid \mu.P \mid P\backslash_\alpha \mid P + P \mid P|P$.

Assuming the reader familiar with the notation, we just recall that the operations above correspond to the *inactive* process, *action prefix*, *restriction*, *nondeterministic sum* and *parallel composition*. We let P, Q, R, ... range over the set **Proc** of CCS processes. The SOS system for CCS processes is defined by the set of De Simone rules in Figure 6 (the obvious symmetric rules for sum and asynchronous communication are omitted).

The basic ATF tiles for CCS proposed in [11] are given in Figure 7. The basic MTF tiles for CCS proposed in [16] differ from those in the ATF only in the

$$\frac{}{\mu.P \xrightarrow{\mu} P} \qquad \frac{P \xrightarrow{\mu} Q}{P\backslash_\alpha \xrightarrow{\mu} Q\backslash_\alpha} \text{ (if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$\frac{P \xrightarrow{\mu} Q}{P + R \xrightarrow{\mu} Q} \qquad \frac{P \xrightarrow{\mu} Q}{P|R \xrightarrow{\mu} Q|R} \qquad \frac{P \xrightarrow{\lambda} Q,\ P' \xrightarrow{\bar{\lambda}} Q'}{P|P' \xrightarrow{\tau} Q|Q'}$$

Fig. 6. De Simone rules for finite CCS

$$\mathtt{act}_\mu : \mu.x_1 \xrightarrow[\mu]{id} x_1 \qquad \mathtt{res}_{\mu,\alpha} : x_1\backslash_\alpha \xrightarrow[\mu]{\mu} x_1\backslash_\alpha \text{ (if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$\langle +_\mu : x_1 + x_2 \xrightarrow[\mu]{\mu\otimes id} x_1 \qquad \rfloor_\mu : x_1|x_2 \xrightarrow[\mu]{\mu\otimes id} x_1|x_2 \qquad \|_\lambda : x_1|x_2 \xrightarrow[\tau]{\lambda\otimes\bar{\lambda}} x_1|x_2$$

Fig. 7. ATF tiles for finite CCS

$$\mathtt{act}_\mu : 1 \lhd\ \mu.x_1 \xrightarrow[\mu(x_1)]{x_1} x_1 \qquad \mathtt{res}_{\mu,\alpha} : 1 \lhd\ x_1\backslash_\alpha \xrightarrow[\mu(x_1)]{\mu(x_1)} x_1\backslash_\alpha \quad (\text{if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$\langle +_\mu : 2 \lhd\ x_1 + x_2 \xrightarrow[\mu(x_1)]{\mu(x_1)} x_1 \qquad \rfloor_\mu : 2 \lhd\ x_1|x_2 \xrightarrow[\mu(x_1)]{\mu(x_1),x_2} x_1|x_2 \qquad \|_\lambda : 2 \lhd\ x_1|x_2 \xrightarrow[\tau(x_1)]{\lambda(x_1),\bar{\lambda}(x_2)} x_1|x_2$$

Fig. 8. TTF tiles for finite CCS

treatment of nondeterministic sum. This is because the MTF has no horizontal discharger and therefore an explicit unary operator !(_) must be introduced for garbaging discarded arguments. Thus, the MTF tile for (left choice in) nondeterministic sum is $\langle +_\mu : x_1 + x_2 \xrightarrow[\mu]{\mu\otimes id} x_1 \otimes !(x_2)$. There are no tiles with source $!(x_1)$ and therefore the operator !(_) locks its argument. The basic GSTF tiles for CCS have not been presented in the literature, but they are essentially the same as those of the MTF case, where the operator !(_) is the discharger $!_{\underline{1}}$. The basic TTF tiles for CCS proposed in [4] are given in Figure 8. Apart from the notation, the main difference w.r.t. the ATF tiles resides again in the rule for sum (the unused argument can be discarded at the level of observations).

4 The Monoidal Tile Format

If the MTF is employed, the basic source property establishes a bijective correspondence between basic tiles and rules in DSF (as noted in [11]). Hence it is easy to show that tile bisimilarity on closed terms coincides with bisimilarity on the LTS generated by the TTS associated to the tile system. However, we can extend the congruence to contexts in a different way from instantiation closure.

Theorem 1. *If a monoidal tile system $\mathcal{R} = (\Sigma, \Lambda, N, R)$ enjoys the basic source property, then tile bisimilarity defines a congruence also on contexts.*

Of course a congruence on contexts is closed under instantiations, and therefore $\simeq_t \subseteq \simeq$. It can be shown that $\simeq_t$ is in general finer than $\simeq$ on contexts. We propose the following example, inspired by [19].

Example 1. Let us consider finite CCS (see Section 3.1) extended with the family of unary operators $do^n_\mu(\_)$ with $n \in \mathbb{N}$ and μ an action. For all $n > 0$, the behaviour of $do^n_\mu(\_)$ is described by the DSF rule in Figure 9(a), whose corresponding MTF tile is $do^n_\mu(x_1) \xrightarrow[\mu]{\mu} do^{n-1}_\mu(x_1)$. There are no rules for $do^0_\mu(\_)$. Then the contexts $C_1[x_1] = \beta.do^1_\alpha(x_1) + \beta.\alpha.nil + \beta.nil$ and $C_2[x_1] = \beta.\alpha.nil + \beta.nil$ are bisimilar but not tile bisimilar. In fact, if p is a process, then $C_1[p] \xrightarrow{\beta} do^1_\alpha(p)$, and $do^1_\alpha(p)$ has either no transition (when p cannot do α) or a transition leading to the deadlocked state $do^0_\alpha(q)$ for some q with $p \xrightarrow{\alpha} q$. Thus $C_2[p] = \beta.\alpha.nil + \beta.nil$ can always match the β move, i.e., $C_1[p] \simeq C_2[p]$ for all closed p. On the other hand, the tile $C_1[x_1] \xrightarrow[\beta]{id} do^1_\alpha(x_1)$ cannot be bisimulated by $C_2[x_1]$. In fact, if $C_2[x_1] \xrightarrow[\beta]{id} \alpha.nil$, then $\alpha.nil \xrightarrow[\alpha]{id} nil$ cannot be matched by $do^1_\alpha(x_1)$. If instead $C_2[x_1] \xrightarrow[\beta]{id} nil$ then $do^1_\alpha(x_1) \not\simeq_t nil$. Note that we have slightly abused the notation by writing, e.g., nil instead of $nil[x_1] = nil \otimes !(x_1)$ as required by MTF.

5 Algebraic vs. Term Tile Formats

When dealing with nonlinear contexts and tile bisimulation, the ATF can give rise to unexpected results, as the following example illustrates.

Example 2. Let us consider the ATF tiles for finite CCS in Section 3.1. It is straightforward that on closed terms tile bisimilarity $\simeq_t$ coincides with bisimilarity $\simeq$ on the ordinary LTS. Moreover, we know that $\simeq$ is a congruence. However, on open terms $\simeq_t$ is different from $\simeq$. In fact, let us consider the contexts $C_1[x_1] = (x_1|x_1)\backslash_\alpha$ and $C_2[x_1] = (x_1\backslash_\alpha|x_1\backslash_\alpha)$. Then $C_1[x_1] \simeq_t C_2[x_1]$, but $C_1[x_1] \not\simeq C_2[x_1]$, since there exists a process p (e.g., $p = \alpha.nil + \bar{\alpha}.nil$) such that $C_1[p] \xrightarrow{\tau} Q$ and $C_2[p]$ cannot move. But then, of course, $C_1[p] \not\simeq_t C_2[p]$ and therefore $\simeq_t$ is not a congruence on contexts. Finite CCS admits also a presentation in TTF (also recalled in Section 3.1). Again we have that, on closed terms, $\simeq_t$ and $\simeq$ coincide. However, thanks to the auxiliary TTF tiles, $C_1[x_1] \not\simeq_t C_2[x_1]$, because the tile $1 \triangleleft C_1[x_1] \xrightarrow[\tau(x_1)]{\alpha(x_1),\bar{\alpha}(x_1)} (x_1|x_2)\backslash_\alpha$ (see Figure 9(b), noting that $C_1[x_1] = \nabla_{\underline{1}}; (x_1|x_2)\backslash_\alpha$) cannot be mimicked by $C_2[x_1]$.

A more general result demonstrates that the fact that the TTF for CCS behaves better than its ATF counterpart is not a mere coincidence.

Theorem 2. *If a term tile system $\mathcal{R} = (\Sigma, \Lambda, N, R)$ enjoys the basic source property, then tile bisimilarity is a congruence (also for open terms).*

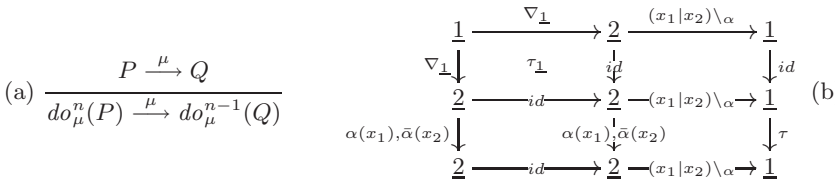


Fig. 9. DSF rule for do^n_μ (a) and the cell pasting of Example 2 (b)

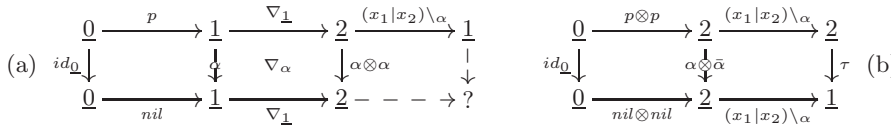


Fig. 10. Tile pastings described in Example 3

The previous theorem states that if a system can be expressed in TTF with basic source, then we have a more satisfactory equivalence on open terms than the one obtained by closing contexts under all possible instantiations. In fact it is completely analogous and consistent with that of closed terms and takes care of the specification constraints (initial configurations can be seen, e.g., as incomplete open software modules) rather than just of the effective realizations (closed systems). As for MTF, tile bisimilarity for TTF is in general finer than $\simeq$.

6 The Gs-Monoidal Format for Explicit Sharing

The definition of good SOS formats for process algebras based on term graphs rather than terms is not straightforward, since the interplay between configurations and observations is difficult to manage. Tiles can be used to overcome this inconvenience. The idea is to consider a monoidal category of observations and a gs-monoidal category of configurations. Though the auxiliary and basic tiles of GSTF are much like those of ATF, this time the gs-monoidal (not the cartesian) structure is lifted from configurations to (horizontal composition of) tiles.

Example 3. The term graph representation **let** $x_1 = \alpha.nil + \bar{\alpha}.nil$ **in** $(x_1|x_1)\backslash_\alpha$ of the CCS agent $((\alpha.nil + \bar{\alpha}.nil)|(\alpha.nil + \bar{\alpha}.nil))\backslash_\alpha$ can only evolve by synchronizing the moves that $p = \alpha.nil + \bar{\alpha}.nil$ can perform with the open behavior of the nonlinear context $(x_1|x_1)\backslash_\alpha = \nabla_{\underline{1}}; (x_1|x_2)\backslash_\alpha$. Since p can execute either α or $\bar{\alpha}$ but not both at the same time, then no such synchronization is possible (see the incomplete tile pasting in Figure 10(a)). This is not the case of $(p|p)\backslash_\alpha$, where there are two subcomponents p that can perform complementary moves and synchronize (see Figure 10(b), noticing that $p; \nabla_{\underline{1}} \neq \nabla_{\underline{0}}; (p \otimes p) = p \otimes p$).

Likewise MTF and TTF, also tile bisimilarity for GSTF enjoys a nice congruence property, providing a good format for the specification of resource aware systems.

Theorem 3. *If a gs-monoidal tile system $\mathcal{R} = (\Sigma, \Lambda, N, R)$ enjoys the basic source property, then tile bisimilarity is a congruence (also for open terms).*

7 Related Work: Conditional Transition Systems

The basic ingredients of Rensink's *conditional transition systems* (CTSs) are *conditional transitions* $\Gamma \vdash s \xrightarrow{a} t$, where s and t are open terms, and the *environment* Γ is a finite graph $\{x_1 \xrightarrow{a_1} y_1, \ldots, x_n \xrightarrow{a_n} y_n\}$ that provides suitable assumptions on the arguments of s and t for the open transition to exist. Building on this, Rensink proposes two kinds of bisimilarities called *formal hypothesis* ($\simeq^{\mathrm{fh}}$) and *hypothesis preserving* ($\simeq^{\mathrm{hp}}$). The difference between them is that in the latter the Γ are persistent, and thus can be reused during bisimulation.

Although one could expect that conditional transitions correspond to tiles $s \xrightarrow[a]{\Gamma} t$ for Γ belonging to a suitable category of observations, a comparison with the tile model is not straightforward. One important difference is that in Rensink's approach the environment Γ is observed, which provides all the *potential* triggers of the systems, while in the tile approach the observation of a particular step includes the actual trigger. Moreover, a different operation closure is introduced by Rensink via *conditional transition system specifications* (CTSSs) that come equipped with three administrative rules (called *variable*, *weakening* and *substitution*) that operate on the basic conditional transitions of any specification. The main theorem of [19] states that for (recursion based) CTSSs both $\simeq^{\mathrm{fh}}$ and $\simeq^{\mathrm{hp}}$ are congruences and $\simeq^{\mathrm{fh}} \subset \simeq^{\mathrm{hp}} \subset \simeq$. Rensink also conjectured that in the ATF setting $\simeq_{\mathrm{t}} = \simeq^{\mathrm{fh}} = \simeq^{\mathrm{hp}}$, probably inspired by the fact that tile triggers define acyclic environments, where the assumptions made in the past cannot influence successive steps. The fact that for the ATF the basic source property does not imply that $\simeq_{\mathrm{t}}$ is a congruence gives evidence that the models built via CTSS and tiles can be very different, and indeed a formal correspondence between administrative rules and tile operations is hard to state — informally, variable, substitution and weakening rules correspond respectively to horizontal identities, horizontal composition and parallel composition. Nonlinear terms also introduce tiles such as $x_1|x_1 \xrightarrow[\alpha;\alpha]{\alpha} x_1|x_1$ (it appears in all the tile systems for CCS that we have considered, except MTF); this corresponds to using the same assumption to prove 'atomically' two consecutive steps, which can have some interpretation for $\simeq^{\mathrm{hp}}$, but not for $\simeq^{\mathrm{fh}}$.

8 Concluding Remarks and Future Work

We have proposed several tile formats for defining bisimilarity congruences directly on both closed and open terms. The simpler MTF is designed for linear systems. The more expressive TTF reflects nonlinearity of configurations at the

level of observations. The GSTF provides a sound formal framework for the treatment of resource aware systems, previously missing in the literature. In many such cases the congruence proofs (via the decomposition property) can be carried out at the pictorial level as tile pastings (see details in the technical report [3]).

Though at a first look open systems seem just the natural extensions of closed systems, we have noted an initial classification that would distinguish between incomplete systems, which define the behavior (at the top level) of the system to be refined by providing the corresponding components, and coordinators, which define services to be used by the processes instantiating their free arguments. Thus, the latter class corresponds to bottom-up design and the most usual notion of reusable component. As discussed in the Introduction, it is reasonable to cope with variables in these two classes of open systems in a different way, specially when their interfaces contain repeated occurrences of the same variables.

Another interesting point is the duality between the extension of a process by (partial) instantiation of its variables, and by embedding it in a context (seen as an incomplete system by itself). In [5], the notion of *ground* tile bisimulation has been developed, which thanks to additional transitions labelled by contexts (in the style of dynamic bisimilarity) is a congruence, practically without any format limitation on basic tiles. It seems rather interesting to try to unify the advances in both directions to obtain a uniform treatment of both internal and external contexts. Finally, an interesting open problem is the extension of our results to the adequate notion of weak tile bisimulation.

Acknowledgements

We thank Arend Rensink for some preliminary discussions on the relationship between conditional transition systems and tiles. We also thank the anonymous referees for their helpful comments.

References

1. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995. 259, 265
2. R. Bruni, Tile Logic for Synchronized Rewriting of Concurrent Systems, Ph.D. Thesis TD-1/99, Computer Science Department, University of Pisa, 1999. 263, 265, 268
3. R. Bruni, D. de Frutos-Escrig, N. Martí-Oliet, and U. Montanari. Tile bisimilarity congruences for open terms and term graphs. Technical Report TR-00-06, Computer Science Department, University of Pisa, 2000. 262, 273
4. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, 1998. 262, 265, 267, 268, 269
5. R. Bruni, U. Montanari, and V. Sassone. Open ended systems, dynamic bisimulation and tile logic. In *Proc. IFIP-TCS 2000*, *LNCS*. Springer, 2000. To appear. 273
6. A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–331, 1999. 266

7. A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Th. Comput. Sci.*, 103:51–106, 1992. 261
8. R. De Simone. Higher level synchronizing devices in MEIJE–SCCS. *Theoret. Comput. Sci.*, 37:245–267, 1985. 259, 265
9. M. C. van Eekele, M. J. Plasmeijer, and M. R. Sleep, editors. *Term Graph Rewriting: Theory and Practice*, Wiley, London, 1993. 261, 266
10. F. Gadducci and U. Montanari. Rewriting rules and CCS. in *Proceeding WRLA'96*, vol. 4 of *Elect. Notes in Th. Comput. Sci.*, Elsevier Science, 1996. 261
11. F. Gadducci and U. Montanari. The tile model. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1999. To appear. 261, 263, 264, 266, 267, 268, 269
12. J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992. 259, 265
13. F. W. Lawvere. Functorial semantics of algebraic theories. *Proc. National Academy of Science*, 50:869–872, 1963. 265
14. K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In *Proc. ICALP'90*, vol. 443 of *LNCS*, pages 526–539, Springer, 1990. 261
15. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoret. Comput. Sci.*, 96:73–155, 1992. 261
16. J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In *Proc. WADT'97*, vol. 1376 of *LNCS*, pages 62–91. Springer, 1998. 262, 267, 268
17. R. Milner. *A Calculus of Communicating Systems*, vol. 92 of *LNCS* Springer, 1980. 259, 260, 268
18. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981. 259, 265
19. A. Rensink. Bisimilarity of open terms. *Inform. and Comput.* 156:345–385, 2000. 261, 262, 270, 272

Process Languages for Rooted Eager Bisimulation

Irek Ulidowski[1] and Shoji Yuen[2]

[1] Department of Mathematics and Computer Science,
University of Leicester, University Road, Leicester LE1 7RH, U.K.
I.Ulidowski@mcs.le.ac.uk
[2] Department of Information Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8603 Japan
yuen@nuie.nagoya-u.ac.jp

Abstract. We present a general class of process languages for rooted eager bisimulation preorder and prove its congruence result. Also, we present classes of process languages for the rooted versions of several other weak preorders. The process languages we propose are defined by the *Ordered* SOS method which combines the traditional SOS approach, based on transition rules, with the *ordering* on transition rules. This new feature specifies the order of application of rules when deriving transitions of process terms. We support and illustrate our results with examples of process operators and process languages which benefit from the OSOS formulation.

1 Introduction

Structural Operational Semantics (SOS) [17,21] is a powerful and widely used method for assigning operational meaning to language operators of process languages, programming languages and specification formalisms. It provides a global perspective on, and improves understanding of, the many existing process languages. Also, it points the way to a theoretically sound method for the development of new, task-specific process languages.

In the SOS approach the meaning of language's operators is defined by sets of *transition rules*, which are proof rules with possibly several premises and a conclusion. For each operator transition rules describe how the behaviour of a process term constructed with the operator as the outer-most operator depends on the behaviour of its subterms, or on the behaviour of other process terms constructed with these subterms. In general, the syntactic structure of rules determines the kind of behavioural information that can be used to define operators. Rules with the simplest structure are De Simone rules [10]: for example, all CCS operators [19] are defined by De Simone rules. De Simone rules with *negative premises* (expressions like $X \stackrel{\tau}{\not\rightarrow}$) and *copying* (multiple use of identical process variables) are called GSOS rules [8,9]. Furthermore, GSOS rules with composite terms in the premises are *ntyft* rules [14], and additionally with predicates are *panth* rules [29]. A process language with its SOS definition generates

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 275–289, 2000.

a *labelled transition system*, where states are the closed terms over the set of operators for the language, and the transitions between states are derived from transition rules for the language's operators.

The operational method allows us to classify and analyse process operators, and thus process languages, according to the form of transition rules that can be used to define them. Also, it helps to generalise many theoretical results for the particular process languages into meta results for the whole classes of process languages as, for example, for the GSOS class [9,2,7,4]. In [25,26] an extension of the SOS method with orderings on transition rules, called Ordered SOS, is developed to give an alternative and equally expressive formulation of the GSOS class. Instead of GSOS rules, which may contain negative premises, OSOS uses only positive GSOS rules but compensates the lack of negative premises with orderings on rules, the new feature that specifies the order of application of rules when deriving transitions of process terms. The advantages of the OSOS method are that (a) many process operators, for example the sequential composition, delay and the timed versions of standard operators, have simpler definitions, (b) these definitions suggest possible ways of implementing the operators, and (c) in our view it is easier to represent and analyse the interplay between the use of *silent* actions and negative process behaviour in transition rules.

In this paper we distinguish between external behaviour of processes and their internal, unobservable behaviour, and we take into account *divergence* of processes. Hence, we work with *weak* preorders on processes, particularly those based upon the notions of *eager bisimulation* [18,1,30,23,25,28]. [25,26] introduces a class of process languages for eager bisimulation preorder, i.e. any process operator for any language in the class *preserves* eager bisimulation preorder. Informally, an n-ary operator is said to preserve a relation on processes, here a preorder, if it produces two related processes from any two sets of n pairwise related subprocesses. The eager bisimulation class is defined within the OSOS approach by imposing several simple conditions on the structure of rules and on orderings on rules. These conditions describe the intended and well understood character of observable and unobservable actions, and guarantee that the difference between these actions is observed in the OSOS definitions. For example, the *uncontrollable* and the *environment-independent* character of silent actions is represented via τ-rules, due to Bloom [6], by insisting that the set of rules for operator f contains a τ-rule τ_i of the form

$$\frac{X_i \xrightarrow{\tau} X_i'}{f(X_1, \ldots, X_i, \ldots, X_n) \xrightarrow{\tau} f(X_1, \ldots, X_i', \ldots, X_n)}$$

for each *active* argument X_i. Informally, argument X_i of f is active if there is a rule for f with premises describing the behaviour of X_i. One of the consequences of requiring τ-rules for active arguments is that some standard process operators, for example the CCS '+', cannot be defined by this method. We will remedy this problem in this paper.

A notion closely related to the uncontrollable character of silent actions is divergence. We define divergence as the ability to perform an infinite sequence

of silent actions. Results in [22,23,6] show that, in a setting with τ-rules, if one treats divergence as a form of deadlock, then rules with negative premises are unacceptable—operators defined by rules with negative premises do not preserve (divergence insensitive) weak equivalences. On the other hand, treating divergence as different from deadlock allows one to use rules with negative premises, or rules with orderings, safely. We distinguish between divergence and deadlock and, consequently, we work with divergence sensitive version of eager bisimulation and other weak relations. Since several popular process operators, for example the CCS '+', do not preserve many weak preorders we consider the *rooted* versions of these weak preorders.

The paper proposes a new class of process languages called rebo (*rooted eager bisimulation ordered*). Operators of any rebo process language are defined by the OSOS method together with additional conditions on rules and on rule orderings. In particular, we insist that for each active ith argument of operator f we either have τ_i amongst the rules for f or the following rule, which we call *silent choice* rule and denote by τ^i.

$$\frac{X_i \xrightarrow{\tau} X_i'}{f(X_1, \ldots, X_i, \ldots, X_n) \xrightarrow{\tau} X_i'}$$

Thus, we have the ability to define operators such as the CCS '+', the Kleene star operator [3], the *interrupt* operator [19] and *delay* operator [16] as well as many *timed* versions of process operators.

Our main congruence theorem states that, for any rebo process language, rooted eager bisimulation preorder, defined over the labelled transition system for the language, is preserved by all process operators of the language. The theorem's proof is somewhat long, and the reader is referred to [27] for full details. The proof employs several intermediate results. One of them, the alternative characterisation of rooted eager bisimulation preorder, is interesting in itself. It shows that although the notion of *stability*, the inability to perform silent action, is not directly used in the definition of rooted eager bisimulation preorder, it is vital in the proof of the congruence theorem for the preorder.

The paper is organised as follows. In Section 2 we recall the definitions of labelled transition system and the eager bisimulation relation. We also define the rooted eager bisimulation relation. In Section 3 we introduce the OSOS method and use it to define the rebo class of process languages. In Section 4 we state our congruence theorem. Section 5 defines classes of process languages for the rooted versions of testing preorder, branching bisimulation and refusal simulation preorders. The last section contains rebo definitions of several popular process operators and process languages.

2 Eager and Rooted Eager Bisimulations

In this section we recall the definitions of eager bisimulation and rooted eager bisimulation, compare them with weak bisimulation, and give an alternative characterisation of eager bisimulation.

Definition 1. A labelled transition system, abbreviated as LTS, is a structure $(\mathcal{P}, A, \rightarrow)$, where $\mathcal{P}$ is the set of states, A is the set of actions and $\rightarrow \subseteq \mathcal{P} \times A \times \mathcal{P}$ is a *transition relation*.

We model concurrent systems by process terms (processes) which are the states in an LTS. Transitions between the states, defined by a transition relation, model the behaviour of systems.

$\mathcal{P}$, the set of processes, is ranged over by $p, q, r, s, \ldots$. Vis is a finite set of visible actions and it is ranged over by a, b, c. Action τ is the silent action and $\tau \notin \mathsf{Vis}$. $\mathsf{Act} = \mathsf{Vis} \cup \{\tau\}$ is ranged over by α, β. We will use the following abbreviations. We write $p \xrightarrow{\alpha} q$ for $(p, \alpha, q) \in \rightarrow$ and call it a *transition*. We write $p \xrightarrow{\alpha}$ when there is q such that $p \xrightarrow{\alpha} q$, and $p \not\xrightarrow{\alpha}$ otherwise. Expressions $p \stackrel{\tau}{\Rightarrow} q$ and $p \stackrel{\alpha}{\Rightarrow} q$, where $\alpha \neq \tau$, denote $p(\xrightarrow{\tau})^* q$ and $p(\xrightarrow{\tau})^* \xrightarrow{\alpha} q$ respectively. Given a sequence of actions $s = \alpha_1 \ldots \alpha_n$, where $\alpha_i \in \mathsf{Act}$ for $1 \leq i \leq n$, we say that q is a s-derivative of p if $p \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} q$. The expression $p\Uparrow$, read as p is divergent, means $p(\xrightarrow{\tau})^\omega$. We say p is convergent, written as $p\Downarrow$, if p is not divergent. We assume that, if $\alpha = \tau$ then $p \stackrel{\widehat{\alpha}}{\rightarrow} p'$ means $p \xrightarrow{\tau} p'$ or $p \equiv p'$, else it is simply $p \xrightarrow{\alpha} p'$.

Definition 2. Given $(\mathcal{P}, \mathsf{Act}, \rightarrow)$, a relation $R \subseteq \mathcal{P} \times \mathcal{P}$ is an eager bisimulation if, for all $(p, q) \in R$, the following properties hold.

(a) $\forall \alpha.\, (p \xrightarrow{\alpha} p' \;\text{ implies }\; \exists q', q''.\, (q \stackrel{\tau}{\Rightarrow} q' \stackrel{\widehat{\alpha}}{\rightarrow} q'' \text{ and } p' R q''))$

(b) $p\Downarrow \;\text{ implies }\; q\Downarrow$

(c) $p\Downarrow \;\text{ implies }\; \forall \alpha.\, (q \xrightarrow{\alpha} q' \text{ implies } \exists p', p''.\, (p \stackrel{\tau}{\Rightarrow} p' \stackrel{\widehat{\alpha}}{\rightarrow} p'' \text{ and } p'' R q'))$

$p \sqsubseteq q$ if there exists an eager bisimulation R such that pRq.

Example 1. Consider processes p and q defined as follows: $p \xrightarrow{a} \mathbf{0}$, $q \xrightarrow{a} \mathbf{0}$ and $q \xrightarrow{\tau} q$. Process p can perform action a and evolve to the deadlocked process $\mathbf{0}$. Process q can perform a after any number of silent actions, but it can also compute internally by performing silent actions forever. Thus, $q \sqsubseteq p$ but $p \not\sqsubseteq q$. Moreover, consider CCS-like processes $r \equiv a.(b.\mathbf{0} + \tau.c.\mathbf{0})$ and $s \equiv a.(b.\mathbf{0} + \tau.c.\mathbf{0}) + a.c\mathbf{0}$. Processes r and s are equivalent according to the weak bisimulation relation of Milner [19] ($s = r$ is an instance of the third τ-law), but $r \not\sqsubseteq s$ since after $s \xrightarrow{a} c.\mathbf{0}$ there is no r' such that $r \stackrel{\tau}{\Rightarrow} \xrightarrow{a} r'$ and $r' \sqsubseteq c.\mathbf{0}$.

We easily check that $\sqsubseteq$ is a preorder. $\sqsubseteq$ is the version of bisimulation with silent actions and divergence studied in [18,1,30,23,25,26], where modal logic, testing and axiomatic characterisations were proposed and congruence results for the ISOS format [23], and eb and ebo formats [25,26] were proved. $\sqsubseteq$ coincides with a *delay* bisimulation [31,13] for LTSs with no divergence. We prefer eager bisimulation to the standard weak bisimulation [19] because languages for eager bisimulation [22,23,25,26] have simpler formulations than those for weak bisimulation [6]. Moreover, since languages for eager bisimulation allow rules with negative premises [22,23] or rules with orderings [25,26], which have the

effect of negative premises, such languages are more general than languages for weak bisimulation. Another reason for choosing eager bisimulation is that, unlike eager bisimulation, weak bisimulation is not preserved by some simple and useful operators, and the problem is not due to the initial silent actions. For example, action refinement operator [26] preserves eager bisimulation but not weak bisimulation. Having said that, we agree that both bisimulation relations are equally suitable for process languages which are defined by rules with no negative premises and where divergence is not modelled.

It is well known that eager bisimulation and many other weak process relations are not preserved by some popular process operators, for example the CCS '+'. We have $a.\mathbf{0} \sqsubseteq\!\!\approx \tau.a.\mathbf{0}$ but not $a.\mathbf{0} + b.\mathbf{0} \sqsubseteq\!\!\approx \tau.a.\mathbf{0} + b.\mathbf{0}$. The standard solution to this problem is to use the *rooted* version of the preferred relation instead the relation itself [19,5].

Definition 3. Given $(\mathcal{P}, \mathsf{Act}, \rightarrow)$, a relation $R \subseteq \mathcal{P} \times \mathcal{P}$ is a *rooted eager bisimulation* if, for all $(p, q) \in R$, the following properties hold.

$$\begin{aligned} \forall \alpha.\, (p \xrightarrow{\alpha} p' \;\text{ implies }\; & \exists q', q''.\, (q \stackrel{\tau}{\Rightarrow} q' \xrightarrow{\alpha} q'' \text{ and } p' \sqsubseteq\!\!\approx q'') \\ p{\Downarrow} \;\text{ implies }\; & \forall \alpha.\, (q \xrightarrow{\alpha} q' \text{ implies } \exists p', p''.\, (p \stackrel{\tau}{\Rightarrow} p' \xrightarrow{\alpha} p'' \text{ and } p'' \sqsubseteq\!\!\approx q')) \end{aligned}$$

$p \sqsubseteq\!\!\approx_r q$ if there exists a rooted eager bisimulation R such that pRq.

We easily show that $\sqsubseteq\!\!\approx_r$ is a preorder. One may wonder why our definition does not include a property corresponding to $(E.b)$ of Definition 2. It is because $p \sqsubseteq\!\!\approx_r q$ and $p{\Downarrow}$ imply $q{\Downarrow}$. Finally, we present an alternative characterization of an eager bisimulation which is essential in the proof of our congruence theorem. Its proof can be found in [27].

Proposition 1. *Given $(\mathcal{P}, \mathsf{Act}, \rightarrow)$, a relation $E \subseteq \mathcal{P} \times \mathcal{P}$ is an eager bisimulation if and only if, for all p and q such that pEq, properties (a) and (c) of Definition 2 as well as the following properties hold.*

$$\begin{aligned} & p \stackrel{\tau}{\not\rightarrow} \;\; \textit{implies} \;\; q{\Downarrow} \\ & p \stackrel{\tau}{\not\rightarrow} \;\; \textit{implies} \;\; \forall q'.\, [q \stackrel{\tau}{\Rightarrow} q' \;\textit{implies}\; (pEq' \;\textit{and} \\ & \qquad \forall a, p'.\, (p \xrightarrow{a} p' \;\textit{implies}\; \exists q'', q'''.\, (q' \stackrel{\tau}{\Rightarrow} q'' \xrightarrow{a} q''' \;\textit{and}\; p'Eq''')) \;\textit{and} \\ & \qquad \forall a, q''.\, (q' \xrightarrow{a} q'' \;\textit{implies}\; \exists p'.\, (p \xrightarrow{a} p' \;\textit{and}\; p'Eq''))))] \\ & (p{\Downarrow} \;\; \textit{and} \;\; q \stackrel{\tau}{\not\rightarrow}) \\ & \qquad \textit{implies} \;\; \forall p'.\, [p \stackrel{\tau}{\Rightarrow} p' \;\textit{implies}\; (p'Eq \;\textit{and} \\ & \qquad \forall a, p''.\, (p' \xrightarrow{a} p'' \;\textit{implies}\; \exists q'.\, (q \xrightarrow{a} q' \;\textit{and}\; p''Eq')) \;\textit{and} \\ & \qquad \forall a, q'.\, (q \xrightarrow{a} q' \;\textit{implies}\; \exists p'', p'''.\, (p' \stackrel{\tau}{\Rightarrow} p'' \xrightarrow{a} p''' \;\textit{and}\; p'''Eq'))))] \end{aligned}$$

3 Ordered Process Languages

In this section we present the Ordered SOS method for defining process operators [25,26] and illustrate its usefulness with examples. We establish several

intuitive conditions on OSOS rules and orderings such that OSOS operators that satisfy these conditions preserve $\sqsubseteq_r$.

The OSOS approach has the same expressive power as the GSOS approach [26], but we find OSOS more convenient than GSOS for expressing restrictions that are required for the preservation of weak preorders. In Section 5 we show that, by varying the set of conditions on OSOS rules and orderings, we can easily formulate classes of process languages for the rooted versions of testing preorder, branching bisimulation and refusal simulation preorders.

3.1 Ordered SOS

Var is a countable set of variables ranged over by $X, X_i, Y, Y_i, \ldots$. Σ_n is a set of operators with arity n. A signature Σ is a collection of all Σ_n and it is ranged over by $f, g, \ldots$. The members of Σ_0 are called *constants*; $\mathbf{0} \in \Sigma_0$ is the deadlocked process operator. The set of *open terms* over Σ with variables in $V \subseteq \mathsf{Var}$, denoted by $\mathbb{T}(\Sigma, V)$, is ranged over by $t, t', \ldots$. $Var(t) \subseteq \mathsf{Var}$ is the set of variables in a term t.

The set of *closed terms*, written as $\mathrm{T}(\Sigma)$, is ranged over by $p, q, u, v, \ldots$. In the setting of process languages these terms are called process terms. A Σ context with n holes $C[X_1, \ldots, X_n]$ is a member of $\mathbb{T}(\Sigma, \{X_1, \ldots, X_n\})$. If $t_1, \ldots, t_n$ are Σ terms, then $C[t_1, \ldots, t_n]$ is the term obtained by substituting t_i for X_i for $1 \leq i \leq n$.

We will use bold italic font to abbreviate the notation for sequences. For example, a sequence of process terms $p_1, \ldots, p_n$, for any $n \in \mathbb{N}$, will often be written as $\boldsymbol{p}$ when the length is understood from the context. Given any binary relation R on closed terms and $\boldsymbol{p}$ and $\boldsymbol{q}$ of length n, we will write $\boldsymbol{p}R\boldsymbol{q}$ to mean $p_i R q_i$ for all $1 \leq i \leq n$. Moreover, instead of $f(X_1, \ldots, X_n)$ we will often write $f(\boldsymbol{X})$ when the arity of f is understood. A preorder $\sqsubseteq$ on $\mathrm{T}(\Sigma)$ is a precongruence if $\boldsymbol{p} \sqsubseteq \boldsymbol{q}$ implies $C[\boldsymbol{p}] \sqsubseteq C[\boldsymbol{q}]$ for all $\boldsymbol{p}$ and $\boldsymbol{q}$ of length n and all Σ contexts $C[\boldsymbol{X}]$ with n holes. Similarly, an operator $f \in \Sigma_n$ preserves $\sqsubseteq$ if, for $\boldsymbol{p}$ and $\boldsymbol{q}$ as above, $\boldsymbol{p} \sqsubseteq \boldsymbol{q}$ implies $f(\boldsymbol{p}) \sqsubseteq f(\boldsymbol{q})$. A process language preserves $\sqsubseteq$ if all its operators preserve $\sqsubseteq$. A class of process languages preserves $\sqsubseteq$ if all languages in the class preserve $\sqsubseteq$.

A *substitution* is a mapping $\mathsf{Var} \to \mathrm{T}(\Sigma)$. Substitutions are ranged over by ρ and ρ' and they extend to $\mathbb{T}(\Sigma, \mathsf{Var}) \to \mathrm{T}(\Sigma)$ mappings in a standard way. For t with $Var(t) \subseteq \{X_1, \ldots, X_n\}$ we write $t[p_1/X_1, \ldots, p_n/X_n]$ or $t[\boldsymbol{p}/\boldsymbol{X}]$ to mean t with each X_i replaced by p_i, where $1 \leq i \leq n$.

The notion of transition is extended to expressions $t \xrightarrow{\alpha} t'$.

Definition 4. A (transition) rule is an expression of the form

$$\frac{\{\ X_i \xrightarrow{\alpha_{ij}} Y_{ij}\ \}_{i \in I, j \in J_i}}{f(\boldsymbol{X}) \xrightarrow{\alpha} C[\boldsymbol{X}, \boldsymbol{Y}]}\ ,$$

where $\boldsymbol{X}$ is the sequence $X_1, \ldots, X_n$ and $\boldsymbol{Y}$ is the sequence of all Y_{ij}, and all process variables in $\boldsymbol{X}$ and $\boldsymbol{Y}$ are distinct. Variables in $\boldsymbol{X}$ are the *arguments*

of f. Moreover, $I \subseteq \{1, \ldots, n\}$ and all J_i are finite subsets of $\mathbb{N}$, and $C[\boldsymbol{X}, \boldsymbol{Y}]$ is a context. Let r be the above rule. Then, f is the *operator* of r and $rules(f)$ is the set of all rules with the operator f. The set of transitions above the horizontal bar in r is called the *premises*, written as $pre(r)$. The transition below the bar in r is the *conclusion*, written as $con(r)$. α in the conclusion of r is the *action* of r, written as $act(r)$, and $C[\boldsymbol{X}, \boldsymbol{Y}]$ is the *target* of r. Given $i \in I$, the set of all actions α_{ij} in the premises of r is denoted by $actions(r, i)$. The set of all actions in the premises of r is denoted by $actions(r)$. A rule is an *action* rule if $\tau \notin actions(r)$.

Next, we define *orderings* on rules [25,26]. Let $<_f$ be a binary relation on $rules(f)$. Expression $r <_f r'$ is interpreted as r' having higher priority than r when deriving transitions of terms with f as the outermost operator. Given a signature Σ, the relation $<_\Sigma$, or simply $<$ if Σ is understood from the context, is defined as $\bigcup_{f \in \Sigma} <_f$. We will write $higher(r)$ for $\{r' \mid r < r'\}$.

Note, that, following [26], orderings need not be transitive. This is required for the general expressiveness theorem [26] which states that GSOS and OSOS formalisms have the same expressive power. However, most of the popular operators, that are definable by GSOS rules with negative premises, for example all such operators discussed in this paper, are definable by OSOS rules with transitive orderings. We refer the reader to [26] for further details concerning expressiveness results.

Example 2. Ordered rules have the same effect as rules with negative premises. We give an alternative definition of the sequential composition operator ';' from [9]. The rules are as follows.

$$\frac{X \xrightarrow{a} X'}{X;Y \xrightarrow{a} X';Y}\ r_{a*} \qquad \frac{X \xrightarrow{\tau} X'}{X;Y \xrightarrow{\tau} X';Y}\ \tau_1 \qquad \frac{Y \xrightarrow{c} Y'}{X;Y \xrightarrow{c} Y'}\ r_{*c} \qquad \frac{Y \xrightarrow{\tau} Y'}{X;Y \xrightarrow{\tau} Y'}\ \tau^2$$

The ordering $<$ is defined by $r_{*c} < r_{a*}$, $r_{*c} < \tau_1$ and $\tau^2 < r_{a*}$, $\tau^2 < \tau_1$ for all a and c.

Definition 5. An *Ordered SOS* (or OSOS, for short) *process language* is a tuple $(\Sigma, A, R, <)$, where Σ is a finite set of operators, $A \subseteq \mathsf{Act}$, R is a finite set of rules for operators in Σ such that all actions mentioned in the rules belong to A, and $<$ is the ordering on the rules for the operators in Σ.

Given an OSOS process language $G = (\Sigma, A, R, <)$, we recall how to associate a unique transition relation $\rightarrow$ with G [25,26]. Let $d : \mathrm{T}(\Sigma) \rightarrow \mathbb{N}$ be a function which specifies the depth of ground terms over Σ. Function d is defined inductively as follows: $d(p) = 0$ if p is a constant, and $d(f(p_1, \ldots, p_n)) = 1 + \max\{d(p_i) \mid 1 \leq i \leq n\}$ otherwise.

Definition 6. Given an OSOS process language $(\Sigma, A, R, <)$, we associate with it a transition relation, $\rightarrow \subseteq \mathrm{T}(\Sigma) \times A \times \mathrm{T}(\Sigma)$, which is defined by $\rightarrow = \bigcup_{l < \omega} \rightarrow^l$, where each of transition relations $\rightarrow^l \subseteq \mathrm{T}(\Sigma) \times A \times \mathrm{T}(\Sigma)$ is defined as follows:

$p \xrightarrow{\alpha} p' \in \rightarrow^l$ if and only if $d(p) = l$ and there exist $r \in R$ and a ground substitution ρ such that $\rho(con(r)) = p \xrightarrow{\alpha} p'$, $\rho(pre(r)) \subset \bigcup_{k<l} \rightarrow^k$ and, for all $r' \in higher(r)$, $\rho(pre(r')) \not\subset \bigcup_{k<l} \rightarrow^k$.

The definition states that rule r can be used to derive a transition $p \xrightarrow{\alpha} p'$ if $p \xrightarrow{\alpha} p'$ is the conclusion of r under a substitution ρ, all premises of r are valid under ρ and no rule in $higher(r)$ is applicable. The last means that each rule in $higher(r)$ has a premise which is *not* valid under ρ.

Definition 7. Let $r \in rules(f)$ and $pre(r) = \{X_i \xrightarrow{\alpha_{ij}} Y_{ij} \mid i \in I, j \in J_i\}$. Rule r *applies* to $f(\boldsymbol{u})$ if and only if $u_i \xrightarrow{\alpha_{ij}}$ for all relevant i and j. Rule r is *enabled* at term $f(\boldsymbol{u})$ if and only if r applies to $f(\boldsymbol{u})$ and all rules in $higher(r)$ do not apply.

Returning to Example 2, process $p; q$ can perform an initial action of q (inferred by τ^2 or r_{*c}) if neither r_{a*} nor τ_1 is applicable by Definition 6. That is if $p \xrightarrow{\tau} \!\!\!\!\!\!\rightarrow$ and $p \xrightarrow{a} \!\!\!\!\!\!\rightarrow$ for all a. When p is a purely divergent process, for example defined by the rule $p \xrightarrow{\tau} p$ with p being a constant, then q will never start in $p; q$ since τ_1 is always applicable.

Having defined the transition relation for a given language $(\Sigma, A, R, <)$, we easily construct $(\mathrm{T}(\Sigma), A, \rightarrow)$, the LTS for the language. Eager bisimulation and its rooted version are defined over this LTS as in Definitions 2 and 3.

3.2 rebo Process Languages

Since an arbitrary OSOS process language may contain operators which do not preserve $\sqsubseteq_r$ we introduce several conditions on rules and orderings which guarantee that this does not happen. For this purpose we introduce the notions of *active arguments*, *silent rules* and *copies*.

The ith argument X_i is *active* in rule r, written as $i \in active(r)$, if it appears in a premise of r [6]. Overloading the notation we write $active(f)$ instead of $\{i \mid i \in active(r) \text{ and } r \in rules(f)\}$. The ith argument of $f(\boldsymbol{X})$ is active if it is active in a rule for f.

Consider rules that describe the unobservable behaviour of processes in terms of the unobservable behaviour of their components. We shall call these rules the silent rules and define them as rules r such that $act(r) = \tau$ and $actions(r) = \{\tau\}$. The most widely used rules among the silent rules are the τ-rules, also called *patience* rules [6], as defined in the Introduction. The τ-rule for the ith argument of f is denoted by τ_i when f is clear from the context. The second form of silent rules that we use in this paper are the silent choice rules as presented in the Introduction. The silent choice rule for the ith argument of f is denoted by τ^i when f is clear from the context. Given f, we shall write $tau(i)$ to denote either τ_i or τ^i for f.

The silent choice rules are instances of *choice rules*. Given n-ary operator f, a rule is a choice rule for f and its ith argument if it has the following form.

$$\frac{X_i \xrightarrow{\alpha} X_i'}{f(X_1, \ldots, X_i, \ldots, X_n) \xrightarrow{\alpha} X_i'}$$

$$\textbf{if } \tau \in \mathit{actions}(r,i) \textbf{ then } r = \mathit{tau}(i) \tag{1}$$

$$\textbf{if } i \in \mathit{active}(f) \textbf{ then } \mathit{tau}(i) \in \mathit{rules}(f) \tag{2}$$

$$\textbf{if } i \in \mathit{active}(f) \textbf{ then } \mathit{tau}(i) = \tau_i \tag{3}$$

$$\textbf{if } r' < r \textbf{ and } i \in \mathit{active}(r) \textbf{ then } r' < \mathit{tau}(i) \tag{4}$$

$$\textbf{if } \mathit{tau}(i) < r \textbf{ and } i \in \mathit{active}(r') \cup \mathit{active}(\mathit{higher}(r')) \textbf{ then } r' < r \tag{5}$$

$$\textbf{not } (\mathit{tau}(i) < \mathit{tau}(i)) \tag{6}$$

$$\textbf{if } i \in \mathit{implicit\text{-}copies}(r) \textbf{ then } r < \mathit{tau}(i) \tag{7}$$

$$\textbf{if } i \in \mathit{active}(r) \textbf{ then either } \mathit{tau}(i) = \tau_i \textbf{ or } \mathit{tau}(i) = \tau^i \tag{8}$$

$$\textbf{if } i \in \mathit{active}(r) \textbf{ and } \mathit{tau}(i) = \tau^i \textbf{ then } r \text{ is a choice rule} \tag{9}$$

Fig. 1. Conditions for rebo process operators

Multiple occurrences of process variables in rules are called *copies*. They can be divided into *explicit* and *implicit* copies [22,23]. Given a rule r as in Definition 4, explicit copies are the multiple occurrences of variables Y_{ij} and X_i, for $i \notin I$, in the target t. The implicit copies are the multiple occurrences of X_i in the premises of r and the occurrences, not necessarily multiple, of variables X_i in t when $i \in I$. The set of all implicit copies of process variables in a rule r for f is denoted by $\mathit{implicit\text{-}copies}(r)$ when f is clear from the context.

We are ready to define a general class of OSOS process languages for $\trianglelefteq_r$. Consider the conditions in Figures 1, where f is an operator of an OSOS process language, $<$ is the ordering on the rules for f, r and r' range over $\mathit{rules}(f)$, and τ_i and τ^i are the τ-rule and silent choice rule for the ith argument of f respectively. Conditions (1), (4), (5) and (7)–(9) are implicitly quantified over all r and r' in $\mathit{rules}(f)$, and (1)–(9) are also quantified over all appropriate i.

Definition 8. An operator f is *τ-preserving* if the set of its rules and the ordering on the rules satisfy (1)–(7). An operator f is *τ-sensitive* if the set of its rules and the ordering on the rules satisfy (1)–(2) and (4)–(9). An OSOS process language is rooted eager bisimulation ordered (rebo) if its signature can be partitioned into τ-preserving and τ-sensitive operators and the targets of all rules, except for the τ-rules of τ-sensitive operators, contain only τ-preserving operators. Operators of rebo process languages are called rebo operators.

The silent rules for an operator f that are guaranteed by (2) and either by (3) or (8) shall be henceforth called the silent rules *associated with* f. Similarly, given any rule r for f, all silent rules $\mathit{tau}(i)$ for f such that $i \in \mathit{active}(r)$ shall be called the silent rules *associated with* r.

Condition (1) requires that the only rules with actions τ in the premises that we allow are the silent rules. (2) insists that for each active argument of an operator there is a silent rule for that argument among its rules. We group

operators into τ-preserving and τ-sensitive operators. We require that silent rules associated with τ-preserving operators are purely τ-rules: condition (3). Condition (8) insists that silent rules associated with τ-sensitive operators are either silent choice rules or τ-rules. Conditions (4)–(7) restrict the ordering on rules. The intuition for (4) is that before we apply r' we must check that no rules with higher priority, and thus their associated silent rules, are applicable. (5) requires that if a rule disables a silent rule for argument i, then it also disables all rules with active i (call this set R), and all other rules with active i that are below the rules in R. Notice that (5) implies the following limited form of transitivity.

$$\text{if } \; r' < tau(i) < r \;\; \text{ then } \;\; r' < r$$

Condition (6) insists that silent rules are always enabled, and (7) allows only implicit copies of stable arguments. Finally, (9) insists that if the silent rule for active ith argument of a rule is a silent choice rule, then the rule itself must be a choice rule.

The reader is referred to [26,27] for examples of τ-preserving operators showing that (1)–(7) are necessary. Similar examples will work for τ-sensitive operators, and (8) and (9) are added to allow operators like '+' and others discussed in Section 6.

rebo process languages are descendants of process languages for eager bisimulation preorder introduced in [26]. They are similar to process languages presented in [25]. The idea of partitioning process operators of a language into two groups, where one of the groups contains τ-sensitive operators, so that the language preserves the rooted versions of several weak equivalences is due to Bloom [6]. It has been employed by Fokkink [11] to propose a general class of process languages for (divergence-insensitive) rooted branching bisimulation.

4 Congruence Theorem

This section states our main congruence result.

Theorem 1. *All* rebo *process languages preserve rooted eager bisimulation preorder.*

The proof of the theorem, as well as the proofs of all auxiliary results, can be found in [27]. It uses the alternative characterisation of $\sqsubseteq_r$, Proposition 1, and it shows why each of the conditions in Figure 1 is needed.

The importance of the theorem is that it permits compositionality of specifications written in rebo process languages.

5 Process Languages for Other Weak Preorders

We discuss several conditions on OSOS rules and their orderings which, together with conditions in Figure 1, define classes of process languages for the rooted

versions of testing preorder and branching bisimulation preorder. We also argue that rebo process languages preserve rooted refusal simulation preorder.

Firstly, we consider *testing preorder* [20,15]. For technical reasons we work with an alternative characterisation of testing preorder which is defined in terms of sequences and acceptance sets as in Definition 4.4.5 in [15] and in [24]. Informally, rooted testing preorder is a subset of the above mentioned characterisation of testing preorder which pairs processes that agree on their initial τ actions, namely they either both have or both do not have initial τ. Results in [24] show that process languages definable by De Simone rules with their associated τ-rules preserve testing preorder. Since De Simone rules cannot have copies of process variables and no negative premises we need the following conditions. $<$ is the ordering on rules for an operator f, r is any rule for f and (11) is implicitly quantified over by all r in $rules(f)$.

$$< \;=\; \emptyset \tag{10}$$

$$copies(r) \;=\; \emptyset \tag{11}$$

Definition 9. A process language is *rooted testing ordered* (rto) if it is rebo and its rules and orderings additionally satisfy conditions (10)–(11).

Note, that condition (10) makes conditions (4)–(6) redundant, and condition (11) makes (7) redundant. As a result, informally, De Simone rules [24], choice rules and their associated silent rules can be used in rto definitions. Additionally, these rules are not ordered, and they can have no copies of process variables.

Theorem 2. *All* rto *process languages preserve rooted testing preorder.*

The proof of this theorem follows closely the proof of the congruence result for testing preorder and De Simone process languages in [24].

Next, consider a 'silent action and divergence sensitive' version [26] of *branching bisimulation* relation [13,12]. It is defined as eager bisimulation in Definition 2 except that formulae pRq' and $p'Rq$ are included as additional conjuncts in conditions (a) and (c) respectively. The rooted version of our branching bisimulation is defined in the same manner as rooted eager bisimulation. OSOS rules for process languages for branching bisimulation, unlike those for eager bisimulation [26], may have implicit copies of process variables in the target. Therefore, we change (7) in Figure 1 to the following condition, which is quantified over all r in $rules(f)$.

$$\textbf{if} \;\; i \in \mathit{implicit\text{-}copies}(pre(r)) \;\; \textbf{then} \;\; r < tau(i) \tag{12}$$

Definition 10. A process language is *rooted branching bisimulation ordered* (rbbo) if it is defined as a rebo process language except that its rules and orderings satisfy condition (12) instead of (7).

(12) says that implicit copies in the premises are only allowed for stable arguments, and together with other conditions implies that implicit copies are freely allowed in the targets of rbbo rules.

Theorem 3. *All* rbbo *process languages preserve rooted branching bisimulation preorder.*

The proof follows closely the proof of Theorem 1. It employs a characterisation of branching bisimulation similar to that of eager bisimulation in Proposition 1.

Refusal simulation preorder [22,23] in a generalisation of ready simulation [9] with silent actions and divergence. Let $\Lambda = \mathsf{Vis} \cup \widetilde{\mathsf{Vis}} \cup \{\tau\}$. Assume Λ is ranged over by μ. For $\widetilde{a} \in \widetilde{\mathsf{Vis}}$ expression $p \xrightarrow{\widetilde{a}}$ denotes $p \stackrel{\tau}{\nrightarrow} \stackrel{a}{\nrightarrow}$, and $p \xrightarrow{\widehat{\mu}} p'$ is the obvious generalisation of $p \xrightarrow{\widehat{\alpha}} p'$.

Definition 11. Given $(\mathcal{P}, \Lambda, \rightarrow)$, relations $R, S \subseteq \mathcal{P} \times \mathcal{P}$ are *L-simulation* and *U-simulation* relations respectively if, for all (p, q) in R and S respectively, the following properties hold.

$$\forall \mu \in \Lambda.\ \forall p'.\ (p \xrightarrow{\mu} p' \text{ implies } \exists q', q''.\ (\ q \stackrel{\tau}{\Rightarrow} q' \xrightarrow{\widehat{\mu}} q'' \text{ and } p' R q''))$$

$$\forall \mu \in \Lambda.\ p{\Downarrow} \text{ implies } [q{\Downarrow} \text{ and } \forall q'.\ (q \stackrel{\mu}{\Rightarrow} q' \text{ implies } \exists p', p''.\ (\ p \stackrel{\tau}{\Rightarrow} p' \xrightarrow{\widehat{\mu}} p'' \text{ and } p'' S q'))]$$

$\sqsubseteq_{\approx_L}$, $\sqsubseteq_{\approx_U}$ and refusal simulation relation, $\sqsubseteq_{\approx_{RS}}$, are binary relations over $\mathcal{P}$ defined as follows:

$$\begin{aligned} p \sqsubseteq_{\approx_L} q &\equiv \exists R.\ R \text{ is an } L\text{-simulation and } pRq, \\ p \sqsubseteq_{\approx_U} q &\equiv \exists S.\ S \text{ is a } U\text{-simulation and } pSq, \\ p \sqsubseteq_{\approx_{RS}} q &\equiv\ p \sqsubseteq_{\approx_L} q \text{ and } p \sqsubseteq_{\approx_U} q. \end{aligned}$$

The rooted version of refusal simulation is defined in the corresponding fashion as the rooted version of eager bisimulation. Results in [22,23,26] for the ISOS and ebo process languages and refusal simulation imply our last result.

Theorem 4. *All* rebo *process languages preserve rooted refusal simulation preorder.*

6 Applications

Most of the popular process operators are τ-preserving rebo. The CCS '+', ';' from Example 2, the Kleene star '*' operator [3] and Milner's interrupt operator '$\wedge$' [19] are τ-sensitive rebo operators. The last two are defined below. Notice that the first and the last rule are choice rules, and recall that $\alpha \in \mathsf{Vis} \cup \{\tau\}$.

$$\frac{X \xrightarrow{\alpha} X'}{a^*X \xrightarrow{\alpha} X'} \qquad a^*X \xrightarrow{a} a^*X \qquad \frac{X \xrightarrow{\alpha} X'}{X \wedge Y \xrightarrow{\alpha} X' \wedge Y} \qquad \frac{Y \xrightarrow{\alpha} Y'}{X \wedge Y \xrightarrow{\alpha} Y'}$$

Temporal Process Language (TPL) of Hennessy and Regan [16] is an example of a process language with several operators definable by rules with negative

premises. These operators are the delay operator '$\lfloor\ \rfloor(\ )$', and the CCS-like parallel composition '$|$'. We show that TPL is rebo, and that in general process languages with (discrete) time have natural and intuitive rebo formulations. The main design issue for process languages with time is what properties of the time passage the resulting LTSs are to have. TPL satisfies several timed properties, for example *time determinism, patience* and *maximal progress*. The last is defined by the formula if $p \xrightarrow{\tau}$ then $p \xrightarrow{\sigma}\!\!\!\!\!\!\rightarrow$ for all TPL processes p. The influence of these properties, particularly maximal progress, on the definitions of operators is as follows. '$|$' is defined by the usual rules together with the rule below, where action σ represents the passage of one time unit.

$$\frac{X \xrightarrow{\sigma} X' \quad Y \xrightarrow{\sigma} Y' \quad X|Y \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow}{X|Y \xrightarrow{\sigma} X'|Y'}$$

The operator '$\lfloor\ \rfloor(\ )$' is defined by the following three rules.

$$\frac{X \xrightarrow{a} X'}{\lfloor X \rfloor(Y) \xrightarrow{a} X'} \qquad \frac{X \xrightarrow{\tau} X'}{\lfloor X \rfloor(Y) \xrightarrow{\tau} X'}\ r_\tau \qquad \frac{X \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow}{\lfloor X \rfloor(Y) \xrightarrow{\sigma} Y}$$

We easily see, from the first and the last rule above, that both operators 'observe' maximal progress. rebo definitions of '$|$' and '$\lfloor\ \rfloor(\ )$' are as follows. '$|$' has the usual rules plus r_σ below

$$\frac{X \xrightarrow{\sigma} X' \quad Y \xrightarrow{\sigma} Y'}{X|Y \xrightarrow{\sigma} X'|Y'}\ r_\sigma$$

and the ordering such that r_σ is below the synchronisation rule for '$|$' and below both the τ-rules for '$|$'. rebo version of '$\lfloor\ \rfloor(\ )$' is defined by the three rules below with the ordering that satisfies $\sigma_\emptyset < r_\tau$.

$$\frac{X \xrightarrow{a} X'}{\lfloor X \rfloor(Y) \xrightarrow{a} X'} \qquad \frac{X \xrightarrow{\tau} X'}{\lfloor X \rfloor(Y) \xrightarrow{\tau} X'}\ r_\tau \qquad \lfloor X \rfloor(Y) \xrightarrow{\sigma} Y\ \ \sigma_\emptyset$$

The delay operator is τ-sensitive, and it introduces the initial σ only when its first argument is stable, thus 'observing' maximal progress. TPL has two operators that introduce actions τ: the parallel composition, which we discussed above, and prefixing with τ which is *urgent*. The last is defined by $\tau.X \xrightarrow{\tau} X$, and clearly observes maximal progress. Usually, non-urgent prefixing with τ has the extra rule $\tau.X \xrightarrow{\sigma} \tau.X$. The remaining TPL operators, including '$+$', are rebo.

The OSOS approach is very convenient for defining process languages with time which satisfy certain timed properties. We start with rebo definitions and then find conditions on the rules and orderings that guarantee the required properties. Let rules that describe the passage of time, for example r_σ and $\sigma_\emptyset$ above, be called *σ-rules*. If, for example, we need to extend our favourite rebo process language with time such that it satisfies maximal progress, we need the

following condition. It requires that σ-rules are below certain rules with the action τ. For all rules r and σ-rules r' for f, and all priority levels J for f

$$\textbf{if} \quad act(r) = \tau \quad \textbf{and} \quad active(r) \cup active(r') \subseteq J \quad \textbf{then} \quad r' < r \qquad (13)$$

where, informally, priority level J for f is a subset of $active(f)$ such that all rules r with $active(r) \subseteq J$ have the same priority, as defined by the ordering on rules for f, when deriving transitions of terms. For example, ';' has two obvious priority levels and '+' has one. Conditions (13) and (4) imply that σ-rules are below the appropriate silent rules. The reader is referred to [28] for details of rebo formulations of process languages with time that satisfy a variety of timed properties.

Acknowledgements

We wish to thank the referees for their comments and suggestions. The first author also thanks Yumi Ishikawa for her constant support.

The first author was supported by the Research Fund of University of Leicester, and the second author by the Oversees Research Programme of the Ministry of Education, Science, Sports and Culture of Japan.

References

1. S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987. 276, 278
2. L. Aceto, B. Bloom, and F. W. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111:1–52, 1994. 276
3. L. Aceto, W. Fokkink, R. van Glabbeek, and A. Ingólfsdóttir. Axiomatizing prefix iteration with silent steps. *Information and Computation*, 127:1–52, 1996. 277, 286
4. L. Aceto and A. Ingólfsdóttir. CPO models for a class of GSOS languages. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Proceedings of TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer, 1995. 276
5. J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990. 279
6. B. Bloom. Structural operational semantics for weak bisimulations. *Theoretical Computer Science*, 146:27–68, 1995. 276, 277, 278, 282, 284
7. B. Bloom. Stuctured operational semantics as a specification language. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, 1995. 276
8. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: preliminary report. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, 1988. 275
9. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of ACM*, 42(1):232–268, 1995. 275, 276, 281, 286
10. R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985. 275

11. W. J. Fokkink. Rooted branching bisimulation as a congruence. *Journal of Computer and System Sciences*, 60(1):13–37, 2000. 284
12. R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, CWI, 1990. 285
13. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996. 278, 285
14. J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118(2):263–299, 1993. 275
15. M. Hennessy. *An Algebraic Theory of Processes*. The MIT Press, 1988. 285
16. M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995. 277, 286
17. R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 275
18. R. Milner. A modal characterisation of observable machine behaviours. In G. Astesiano and C. Böhm, editors, *Proceedings of CAAP'81*, volume 112 of *Lecture Notes in Computer Science*. Springer, 1981. 276, 278
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. 275, 277, 278, 279, 286
20. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984. 285
21. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. 275
22. I. Ulidowski. Equivalences on observable processes. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Science Press, 1992. 277, 278, 283, 286
23. I. Ulidowski. *Local Testing and Implementable Concurrent Processes*. PhD thesis, Imperial College, University of London, 1994. 276, 277, 278, 283, 286
24. I. Ulidowski. Finite axiom systems for testing preorder and De Simone process languages. *Theoretical Computer Science*, 239(1):97–139, 2000. 285
25. I. Ulidowski and I. C. C. Phillips. Formats of ordered SOS rules with silent actions. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Conference on Theory and Practice of Software Development TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*. Springer, 1997. 276, 278, 279, 281, 284
26. I. Ulidowski and I. C. C. Phillips. Ordered SOS rules and process languages for branching and eager bisimulations. Technical Report 1999/15, Department of Mathematics and Computer Science, Leicester University, 1999. 276, 278, 279, 280, 281, 284, 285, 286
27. I. Ulidowski and S. Yuen. Process languages for rooted weak preorders. Full version of this paper, available at `http://www.mcs.le.ac.uk/~iulidowski`. 277, 279, 284
28. I. Ulidowski and S. Yuen. Extending process languages with time. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*. Springer, 1997. 276, 288
29. C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995. 275
30. D. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990. 276, 278
31. W. P. Weijland. *Synchrony and Asynchrony in Process Algebra*. PhD thesis, University of Amsterdam, 1989. 278

Action Contraction

Arend Rensink

Department of Computer Science, University of Twente
Postbus 217, NL–7500 AE Enschede
rensink@cs.utwente.nl

Abstract. The question we consider in this paper is: "When can a combination of fine-grain execution steps be contracted into an atomic action execution"? Our answer is basically: "When no observer can see the difference." This is worked out in detail by defining a notion of *coupled split/atomic simulation refinement* between systems which differ in the atomicity of their actions, and proving that this collapses to Parrow and Sjödin's *coupled similarity* when the systems are composed with an observer.

1 Introduction

One of the most successful abstractions commonly used in the (formal or informal) description of reactive systems is that of an *atomic action* or *atomic transition*, representing an indivisible execution step. Having introduced this concept, for the purpose of formalisation one can proceed to model system behaviour in terms of Kripke structures or transition systems, where the system is considered always to be *in* one of a set of states; the time spent in *transit* from one state to the next is now irrelevant, by virtue of the atomicity assumption.

In most cases, however, alternate models of the same system can be given which show the behaviour in a finer grain of detail, so that the execution steps which were assumed atomic on the abstract level are actually carried out in several stages. Nor is it always possible or desirable to avoid such a level of finer detail: choosing the entities that one wishes to regard as atomic wholes is a very important early design decision, and at that point in time it can be very difficult to choose such abstractions as can be maintained throughout the design process. A prime example of this can be found in *object-based* design, where the *methods* of an object are attractive candidates for atomic actions; yet it is clear that the implementation of a method in general involves a sequence of further method invocations, which we will call *lower-level* — even though in object-based systems a consistent hierarchy of objects does not in general exist.

Indeed, one may be faced with the inverse case: given a fine-grain model of a system, is there an abstraction that allows one to contract sequences of small execution steps into more abstract atomic actions and thereby shrink the model to a more manageable size, without changing the behaviour described in any fundamental way?

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 290–305, 2000.

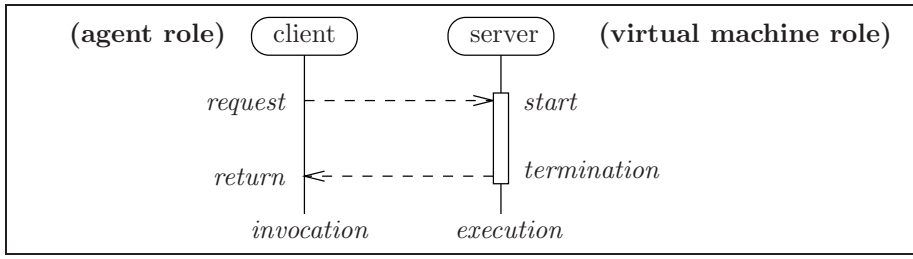


Fig. 1. A single invocation of a primitive, non-atomic action

In either case, once the situation arises where the same behaviour is modelled on different levels of abstraction involving different grains of atomicity, one is forced to consider the following question:

When may a combination of (fine-grain) execution steps be contracted into an (abstract) atomic action?

This is the question we set out to answer in this paper.

Basic assumptions. In developing the theory presented in this paper, we make some rather specific assumptions concerning the execution and synchronisation of actions. These assumptions are inspired by the idea that it should be possible in principle to regard methods of an object (or, more traditionally, procedures in an imperative language) as atomic actions. This is in contrast with the more usual view in reactive systems in which actions are regarded as *messages* that either involve a one-way data transfer (as in CCS [21] or CSP [15]) or a multi-way data exchange (as in ACP [3] or LOTOS [4]). In our view, synchronisation takes place between a *client* and a *server*; data flows from the former to the latter in the form of data parameters and from the latter to the former through a return value. This gives rise to the picture in Fig. 1.

On the abstract level, synchronisation thus involves two atomic actions: an *invocation* by the client and an *execution* by the server. On a finer, non-atomic level of detail, instead we recognise four phases:

- The *request* by the client, which determines the actual parameter values;
- The *start* by the server, which receives the parameter values and models the beginning of the action execution;
- The *termination* by the server, which models the end of the action execution and determines the actual return value;
- The *return* on the side of the client, which receives the return value and models the end of the action invocation.

In the interplay between client and server, the request and start phases are synchronised, as are the termination and return phases. (One can elaborate further on this model by distinguishing the arrival of the request at the server from the actual start of the action — which we will not do, since for our purpose it does not add much to the behavioural aspects we want to model — and also by taking a notion of *abortion* into account; see Sect. 4.

Virtual machines and agents. In Fig. 1, we have deliberately depicted the interaction in a sequence diagram style. In a normal sequence diagram, the model would not be limited to these two parties only: in order to achieve its effects, a server usually calls upon other actions of other parties, and the client's invocation might itself have been part of a longer sequence, in turn triggered by some request from yet another party. In this paper, however, we are interested in those actions that are regarded as *primitive* at the time of modelling; that is, whose decomposition is outside of our range of vision. (It is clear that there must be such primitives, since otherwise there would be an infinite regression of ever-smaller steps.) Another way of expressing this is by saying that we regard the server as a *virtual machine* of which the actions we are modelling are basic statements whose implementation is not our concern.

It should be noted that, although we do not decompose the primitive actions of a virtual machine, this does not imply that those actions are in fact *atomic*. In fact the basic question addressed in this paper can be reformulated as:

When may the primitive actions of a virtual machine be considered atomic?

where (as discussed above) to consider an action atomic means to model its execution as a single step.

If the server is regarded as a virtual machine, the client's invocations must be part of a (possibly concurrent) program running on that machine. That program, or *agent* as we will call it, will usually have some structure whereby high-level actions invoke other actions, which invoke still others until we reach the level of the virtual machine's primitive actions. Of this structure, we are interested only in the bottom level, since this consists of the actions whose atomicity we are trying to establish. We therefore may disregard the agent's internal structure. This indeed leaves us with only the two parties in the interaction that were depicted in Fig. 1; we will henceforth often refer to the server as a virtual machine, and to the client as an agent.

In one respect Fig. 1 is a stark simplification from the scenario we will actually study: it depicts a single interaction only. Where there is a single interaction there can be no interference between interactions; it is in this (potential) interference that the problem of atomicity resides. Thus, in fact we will consider agents that are concurrent, and virtual machines that are in principle capable of serving multiple requests at a time, in whatever (interleaved or concurrent) fashion. However, for the sake of clarity we make the simplifying assumption that *no action is invoked concurrently with itself*; in other words, we limit ourselves to systems that do not display so-called *auto-concurrency*. This assumption is made for this conference version only; the theory is developed without any restriction of this sort in the full report [25].

As a final remark, we point out the essential asymmetry between agent and virtual machine. Virtual machines are considered black boxes; also, they are essentially passive, undertaking observable steps only in response to some invocation. (This passivity is an assumption that is not realistic in all cases, and that we will probably want to lift in the future.) Agents, on the other hand, are

controlled by the programmer and hence transparent in structure; they are conceived of as taking the initiative in an interaction, which at the time of request is surrendered to the virtual machine and reinstated only at the time of return. In this paper, we are not so much interested in the properties of any given agent. Rather, we try to establish a property for a given virtual machine (to wit, the atomicity of its primitive actions) by proving that it interacts in a certain way with *all possible* agents.

Outline. The remainder of this paper is structured as follows: In Sect. 2 we present the basic formalisation of the concepts introduced above. Section 3 presents and discusses the main results of the paper, and provides two small examples. Section 4 concludes the paper and points to related work.

2 Basic Definitions

In order to model the behaviour of virtual machines and agents, we first recall on the well-known model of *labelled transition systems*. We model these as quadruples $T = \langle \Lambda, S, \rightarrow, \iota \rangle$ where Λ is a set of *labels*, with special element $\tau \in \Lambda$ that stands for the *internal action*; S is a set of *states*; $\rightarrow \subseteq S \times \Lambda \times S$ is a set of *transitions*; and $\iota \in S$ is the *initial state*. Λ_T, S_T etc. are used to denote the components of T, and Λ_i, S_i etc. for the components of T_i. The index $_T$ is often omitted when it is clear from the context. Apart from the usual notational conventions, we use $s \Rightarrow s'$ for $s \xrightarrow{\tau}{}^* s'$ and $s \overset{\lambda}{\Longrightarrow} s'$ for $s \Rightarrow \xrightarrow{\lambda} s'$.[1]

The sets Λ we consider will be constructed from the primitive actions of a virtual machine. Throughout the paper, we will use Act, ranged over by a, b, c, to denote a predefined, countably infinite set of such primitive actions. Furthermore, we assume a uniform, countable set Val of *data values*, ranged over by v, w. As discussed at length in the introduction, actions are *invoked* by an agent and *executed* by a virtual machine. Invocation and execution can either be modelled in two phases (non-atomically), as depicted in Fig. 1, or in a single step (atomically). This gives rise to labels of the following kinds:

Invocations. The following labels model the behaviour of an agent:

- $a!$ represents a request for the execution of a, i.e., the first phase of a non-atomic invocation;
- $a{\downarrow}v$ represents either the return of a non-atomic invocation of a with return value v ($\in Val$), or an atomic invocation of a, also with return value v.

Executions. The following labels model the behaviour of a virtual machine:

- $a?$ represents the start of (a non-atomic execution of) the action a;
- $a{\uparrow}v$ represents either the termination of a with return value v (if a is non-atomic) or the complete atomic execution of a, also with return value v.

[1] Note that this differs from the usual definition of $\overset{\lambda}{\Longrightarrow}$, which includes a trailing $\Rightarrow$.

It will be noted that there is an asymmetry here, since we do explicitly model the flow of data from server to client at the time of return, but ignore the parameter values flowing in the opposite direction, at the time of request. This asymmetry is in recognition of the fact that such parameter values can be treated as part of the action being invoked, in the tradition of process algebra with data (see [20,15,4]); not so for the return values, which in general *are not known at the time of request* and hence cannot be part of the action, which is fixed at the time of request. In examples, we will omit the actual return value if it is irrelevant.

2.1 Agents

An *agent* is a labelled transition system T with an associated *invocation alphabet* $A_T \subseteq Act$ partitioned into *atomic actions* A_T^{atom} and *split actions* A_T^{split} such that

$$\Lambda_T = \{a! \mid a \in A_T^{\mathsf{split}}\} \cup \{a{\downarrow}v \mid a \in A_T, v \in Val\} \cup \{\tau\} \ .$$

Each state $s \in S_T$ has a finite set $A_s \subseteq A_T^{\mathsf{split}}$ of *pending actions*. An action is pending if it has been requested but has not yet returned. The following rules guarantee that requests and returns of a given action must occur in strict alternation, starting with a request:[2]

- $A_\iota = \emptyset$;
- If $s \xrightarrow{a!} s'$ then $a \notin A_s$ and $A_{s'} = A_s \cup \{a\}$;
- If $s \xrightarrow{a{\downarrow}v} s'$ then $a \in A_T^{\mathsf{atom}} \cup A_s$ and $A_{s'} = A_s \setminus \{a\}$;
- If $s \xrightarrow{\tau} s'$ then $A_{s'} = A_s$.

Furthermore, we define an *incausality relation* $\not\vdash \subseteq \Lambda_T \times \Lambda_T$ as the smallest relation such that $a! \not\vdash \lambda$ for all $\lambda \neq a{\downarrow}v$, and $\lambda \not\vdash a{\downarrow}v$ for all $a \in A_T^{\mathsf{split}}$ and $\lambda \neq a!$. Incausality expresses that concurrent non-atomic invocations do not influence one another. In particular, the request for one action does not influence any transition that directly follows it, except for the return of that same action; and vice versa, the return of a non-atomic action is "caused" only by the preceding request. Incausality is not symmetric: for instance, the return of an action *can* influence subsequent internal steps or requests of other actions.

An agent T is required to satisfy the following properties for all $s \in S_T$:

Return readiness: If $s \xrightarrow{a!} s'$ then $s' \xrightarrow{a{\downarrow}v}$ for some $v \in Val$. This means that when an action is requested, the agent is immediately ready to receive an answer. (Incausal shuffling — see below — then implies that the agent *remains* ready until the action has actually returned.)

Return value acceptance: If $s \xrightarrow{a{\downarrow}v}$ then $s \xrightarrow{a{\downarrow}w}$ for all $w \in Val$. This means that no return value may be *refused* by the agent.

Return determinism: If $s \xrightarrow{a{\downarrow}v} s'$ and $s \xrightarrow{a{\downarrow}v} s''$ with $a \in A_s$, then $s' = s''$. This means that the effect of a return transition (of a pending action) is completely determined by the return value.

[2] This strict alternation rules out auto-concurrency, as announced in the introduction.

Incausal shuffling: If $s \xrightarrow{\lambda_1} \xrightarrow{\lambda_2} s'$ and $\lambda_1 \not\frown \lambda_2$, then $s \xrightarrow{\lambda_2} \xrightarrow{\lambda_1} s'$. This means that if one transition precedes another that does not causally depend on it, then they can be performed in reverse order with the same effect.

As a consequence, the only difference between atomic and non-atomic invocations is that the latter pass through an intermediate state where the action is pending. By observing what other actions may be invoked in that state, a lot of information can be deduced about the concurrency of the agent. Non-atomic execution is also known as *split semantics* [1,12]; in the absence of auto-concurrency this coincides with the *ST-semantics* [11] that characterises coarsest congruences for action refinement [2,12,27]. In Sect. 2.4, we define the *contraction* of an agent, which abstracts its behaviour by removing precisely these intermediate states.

2.2 Virtual Machines

A *virtual machine* is a labelled transition system T with an associated *execution alphabet* $A_T \subseteq Act$ partitioned into A_T^{atom} and A_T^{split} such that

$$\Lambda_T = \{a? \mid a \in A_T^{\mathsf{split}}\} \cup \{a{\uparrow}v \mid a \in A_T, v \in Val\} \cup \{\tau\} \ .$$

Each state $s \in S_T$ has a finite set $A_s \subseteq A_T^{\mathsf{split}}$ of *running actions*, which are entirely analogous to the pending actions in an agent. We call s an *idle state* if $A_s = \emptyset$, and an *active state* otherwise. The following is required to hold:

- $A_\iota = \emptyset$;
- If $s \xrightarrow{a?} s'$ then $a \notin A_s$ and $A_{s'} = A_s \cup \{a\}$;
- If $s \xrightarrow{a{\uparrow}v} s'$ then $a \in A_T^{\mathsf{atom}} \cup A_s$ and $A_{s'} = A_s \setminus \{a\}$;
- If $s \xrightarrow{\tau} s'$ then $A_{s'} = A_s$.

Furthermore, a virtual machine is required to satisfy the following for all $s \in S_T$:

Potential termination: if $|A_s| > 0$ then $s \xRightarrow{a{\uparrow}v}$ for some $a \in A_s$ and $v \in Val$. This implies that in a virtual machine, an idle state is reachable through termination transitions only, i.e., without starting or executing new actions.

Example 1. As a running example we consider a virtual machine that represents a simple *counter*, whose alphabet A consists of an atomic read action rd and (possibly independent) non-atomic increment actions inc, inc'. Three possible behaviour fragments of the virtual machine are depicted in Fig. 2.

a) Reading is blocked altogether while the increment is executed;
b) Reading is not blocked; initially it keeps on returning the same value as before inc started, whereas after some internal activity, rd returns the incremented value, even before inc itself terminates.
c) The inc_i may be executed concurrently; reading is never blocked, and at every point returns a value that is incremented zero, one or two times.

Figure 2 also includes an example agent T with non-atomic invocation alphabet rd, inc, as well as its A_T*-contraction* (see Sect. 2.4). Both are depicted symbolically, so as to suggest that the return value of rd will be stored as the value of the variable x.

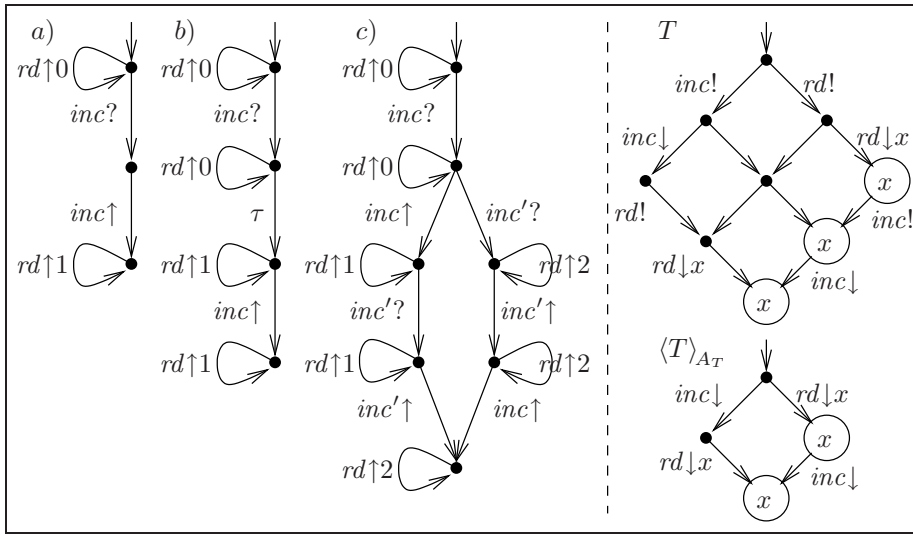


Fig. 2. Virtual counter machines and an agent

2.3 Coupled Simulation

A behavioural model commonly includes much inessential information, which can be discarded by considering the model *up to* some equivalence relation. Such equivalences have been extensively studied; see, e.g, [10]. Quite popular are the equivalences based on *simulation*, in particular *bisimulation*. In this paper we use a variant called *coupled simulation*, due to Parrow and Sjödin [23,24]. (Our definition actually corresponds to the *weakly* coupled similarity of [24].)

Definition 1. *Let T_i for $i = 1, 2$ be two transition systems.*

- *A* simulation *of T_1 by T_2 is a relation $\mathcal{R} \subseteq S_1 \times S_2$ such that for all $(s_1, s_2) \in \mathcal{R}$ (denoted $s_1 \mathcal{R} s_2$), if $s_1 \xrightarrow{\lambda} s_1'$ then one of the following cases holds:*
  - *$\lambda = \tau$ and $s_1' \mathcal{R} s_2$;*
  - *$\lambda \neq \tau$ and $s_2 \stackrel{\lambda}{\Longrightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$.*
- *A relation $\mathcal{R} \subseteq S_1 \times S_2$ is said to be* coupled to *a relation $\mathcal{Q} \subseteq S_2 \times S_1$ if $s_1 \mathcal{R} s_2$ implies $s_2 \Rightarrow s_2'$ for some s_2' such that $s_2' \mathcal{Q} s_1$. $\mathcal{R}$ and $\mathcal{Q}$ are said to be* mutually coupled *or just* coupled *if $\mathcal{R}$ is coupled to $\mathcal{Q}$ and $\mathcal{Q}$ is coupled to $\mathcal{R}$. A* coupled simulation pair *is a pair of mutually coupled relations $\mathcal{R}, \mathcal{Q}$ such that $\mathcal{R}$ is a simulation of T_1 by T_2 and $\mathcal{Q}$ is a simulation of T_2 by T_1.*
- Coupled similarity *between T_1 and T_2 is defined as $\approx = \mathcal{R} \cap \mathcal{Q}^{-1}$, where $\mathcal{R}, \mathcal{Q}$ is the largest coupled simulation pair between T_1 and T_2. $\approx$ is lifted to transition systems by defining $T_1 \approx T_2$ if $\iota_1 \approx \iota_2$.*

We have deviated from the original definition by allowing internal steps in a matching move only *before* a visible transition; in other words, our simulations are actually *delay simulations* (see [13]). However, though delay and weak simulations give rise to different notions of *bi*similarity, the resulting notions of *coupled*

Table 1. Operational rules for contraction and synchronisation

$$\frac{s \xrightarrow{a!} \xrightarrow{a\downarrow v} s' \quad a \in A}{\langle s\rangle_A \xrightarrow{a\downarrow v} \langle s'\rangle_A}\,\mathrm{R}_1 \qquad \frac{s \xrightarrow{\lambda} s' \quad \lambda \notin \{a! \mid a \in A\}}{\langle s\rangle_A \xrightarrow{\lambda} \langle s'\rangle_A}\,\mathrm{R}_2$$

$$\frac{s_1 \xrightarrow{a\downarrow v} s_1' \quad s_2 \xrightarrow{a?} \Rightarrow \xrightarrow{a\uparrow v} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}\,\mathrm{R}_3 \qquad \frac{s_1 \xrightarrow{a!} \xrightarrow{a\downarrow v} s_1' \quad s_2 \xrightarrow{a\uparrow v} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}\,\mathrm{R}_4$$

$$\frac{s_1 \xrightarrow{a!} s_1' \quad s_2 \xrightarrow{a?} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}\,\mathrm{R}_5 \qquad \frac{s_1 \xrightarrow{a\downarrow v} s_1' \quad s_2 \xrightarrow{a\uparrow v} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'}\,\mathrm{R}_6$$

$$\frac{s_1 \xrightarrow{\lambda} s_1' \quad \lambda \notin \{a!, a\downarrow v \mid a \in A_{\mathsf{vm}}\}}{s_1 \| s_2 \xrightarrow{\lambda} s_1' \| s_2}\,\mathrm{R}_7 \qquad \frac{s_2 \xrightarrow{\tau} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1 \| s_2'}\,\mathrm{R}_8$$

similarity coincide. We have chosen the above presentation because it smoothens the proofs of our results. In this light, it is interesting to recall from [13] that weak bisimilarity is not a congruence for action refinement, even in a sequential setting; the coarsest congruence for action refinement within weak bisimilarity is delay bisimilarity (see [7]). It is straightforward to check that coupled similarity is also a congruence for action refinement (in a sequential setting).

2.4 Agent Contraction and Synchronisation

We consider two operations on transition systems: the *contraction* of an agent's invocations and the *synchronisation* of an agent and a virtual machine. Let T_{ag} and T_{vm} denote an arbitrary agent and virtual machine, respectively.

For all $A \subseteq A_{\mathsf{ag}}$, the A-contraction $\langle T_{\mathsf{ag}}\rangle_A$ is essentially the same as T_{ag} except that the actions in A are now invoked atomically. $\langle T_{\mathsf{ag}}\rangle_A$ has atomic alphabet $A_{\mathsf{ag}}^{\mathsf{atom}} \cup A$, non-atomic alphabet $A_{\mathsf{ag}}^{\mathsf{split}} \setminus A$, states $\langle s\rangle_A$ for all $s \in S_{\mathsf{ag}}$ and transitions determined by the SOS-rules in Table 1. Due to the absence of intermediate states, $\langle T_{\mathsf{ag}}\rangle_A$ may be a good deal smaller than T_{ag}; see, e.g., Fig. 2.

The synchronisation of T_{ag} and T_{vm} is denoted $T_{\mathsf{ag}} \| T_{\mathsf{vm}}$ and pronounced as "T_{ag} *running on* T_{vm}" or just "T_{ag} *on* T_{vm}". We only consider synchronisation in cases where the invocation alphabet of T_{ag} is a superset of the execution alphabet of T_{vm}. No requirement, however, is imposed on the atomic and non-atomic actions of T_{ag} and T_{vm}, respectively: actions that are invoked atomically need not be executed atomically, nor vice versa.

$T_{\mathsf{ag}} \| T_{\mathsf{vm}}$ has atomic alphabet $A_{\mathsf{ag}}^{\mathsf{atom}} \setminus A_{\mathsf{vm}}$, non-atomic alphabet $A_{\mathsf{ag}}^{\mathsf{split}} \setminus A_{\mathsf{vm}}$, states $s_1 \| s_2$ for all $s_1 \in S_{\mathsf{ag}}$ and $s_2 \in S_{\mathsf{vm}}$, and transitions determined by the rules in Table 1. The interaction between agent and virtual machine is modelled by R_3–R_6. Rules R_3 and R_4 deal with differences in atomicity: $a \in A_{\mathsf{ag}}^{\mathsf{atom}} \cap A_{\mathsf{vm}}^{\mathsf{split}}$ and $a \in A_{\mathsf{ag}}^{\mathsf{split}} \cap A_{\mathsf{vm}}^{\mathsf{atom}}$, respectively. Note that R_3 is actually a rule *schema*, since the number of τ-transitions between the start and termination of the action in the virtual machine can be arbitrary. Rules R_5 and R_6 cover the cases where

$a \in A_{\mathsf{ag}}^{\mathsf{atom}} \cap A_{\mathsf{vm}}^{\mathsf{atom}}$ (R_6) and $a \in A_{\mathsf{ag}}^{\mathsf{split}} \cap A_{\mathsf{vm}}^{\mathsf{split}}$ (both rules). The other rules deal with cases where either the agent or the virtual machine moves on its own.

It is not difficult to show that contraction and synchronisation both yield agents, with respective invocation alphabets A_{ag} and $A_{\mathsf{ag}} \setminus A_{\mathsf{vm}}$. Furthermore, coupled similarity is a congruence w.r.t. both operators:

Proposition 1. *If* $T_{\mathsf{ag}} \approx T'_{\mathsf{ag}}$ *and* $T_{\mathsf{vm}} \approx T'_{\mathsf{vm}}$ *with* $A_{\mathsf{vm}} \subseteq A_{\mathsf{ag}}$*, then* $\langle T_{\mathsf{ag}} \rangle_A \approx \langle T'_{\mathsf{ag}} \rangle_A$ *for all* $A \subseteq A_{\mathsf{ag}}$ *and* $T_{\mathsf{ag}} \| T_{\mathsf{vm}} \approx T'_{\mathsf{ag}} \| T'_{\mathsf{vm}}$.

Note that there are no meta-results about SOS formats concerning coupled similarity; the proof has to be done "by hand". The first more interesting result of this paper —which actually follows directly by comparing R_4 with the combination of R_1 and R_6— is that atomic actions of a virtual machine might as well be *invoked* atomically. (Although we state the result only up to coupled simulation, in fact it holds up to isomorphism.)

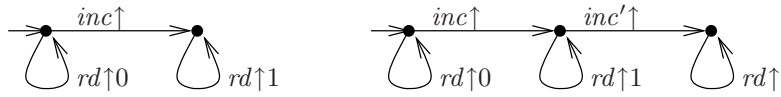
Proposition 2. *If* $A_{\mathsf{vm}} \subseteq A_{\mathsf{ag}}$*, then* $T_{\mathsf{ag}} \| T_{\mathsf{vm}} \approx \langle T_{\mathsf{ag}} \rangle_A \| T_{\mathsf{vm}}$ *for all* $A \subseteq A_{\mathsf{vm}}$.

3 Virtual Machine Contraction

We are now ready to formalise the main question of this paper, discussed in the introduction: *When may the primitive actions of a virtual machine be considered atomic?* For the purpose of formalisation, we rephrase this slightly: *When does a split-action virtual machine correctly implement an atomic one?* We give our answer in terms of observability: *When no agent can see the difference.* "Seeing the difference", in this case, means that synchronising the agent with the virtual machines in question (which should have the same alphabet but possibly different sets of atomic actions) gives rise to behaviour that is *not* coupled similar.

Definition 2. *Two virtual machines,* T_1 *and* T_2*, are* distinguishable *if there exists an agent,* T_{ag}*, such that* $T_{\mathsf{ag}} \| T_1 \not\approx T_{\mathsf{ag}} \| T_2$.

Example 2. We return to the virtual counter of Ex. 1. Consider the following possible behaviours of an atomic counter machine:



It should come as no surprise that behaviour a) in Fig. 2 is indistinguishable from the left hand system. Slightly more surprisingly, this also holds for b), where *rd* is not blocked and the result it returns changes during the execution of *inc*. As for c), despite the overlapping *inc*- and *inc*′-executions that terminate in the reverse starting order, and the fact that the value of *rd* may change from 0 to 2 in a single step, the behaviour is indistinguishable from the right hand atomic counter machine above.

What one would like, of course, is an operational characterisation of indistinguishability. As a first approximation, in this paper we develop a pre-order over virtual machines which implies indistinguishability. The search for the *largest* such relation is entirely open.

3.1 Coupled Split/Atomic Simulation Refinement

Coupled split/atomic simulation refinement relates virtual machines, say T_1, T_2, with identical execution alphabets but such that the atomic actions of T_1 form a superset of the atomic actions of T_2 (the lower-level or more concrete machine). As the name suggests, the relation is defined as a coupling of two simulations: a so-called *split* simulation of T_1 by T_2 and a reverse *atomic* simulation. If $a \in A_1^{\mathsf{atom}} \setminus A_2^{\mathsf{atom}}$, a rough intuition of the simulations and their coupling is:

- *Split simulation* requires that $\xrightarrow{a\uparrow v}$ (in T_1) is simulated by $\xRightarrow{a?}\xRightarrow{a\uparrow v}$ (in T_2);
- *Atomic simulation* requires that $\xrightarrow{a?}$ (in T_2) is not simulated in T_1 at all (T_1 is silent), whereas $\xrightarrow{a\uparrow v}$ is simulated in either of the following ways:
  - By a sequence $\xRightarrow{\boldsymbol{c}\uparrow \boldsymbol{w}}\xRightarrow{a\uparrow v}$ (in T_2), where $\boldsymbol{c}$ is a vector of actions atomic in T_1 but not in T_2, whose execution had already started in T_2 (but had not been simulated yet in T_1). Thus, the actions in $\boldsymbol{c}$ are *pre-simulated*: the abstract machine has executed them completely before they are terminated in the concrete machine.
  - Not at all (T_1 is silent) if a was pre-simulated by T_1, in the above sense.
- Split and atomic simulations are *coupled* in a similar way as (ordinary) simulations (see Def. 1), except that the split simulation is coupled to the atomic simulation only if T_2 is in an idle state.

The precise definition is complicated by the fact that the pre-simulated actions have to be remembered somehow. Moreover, pre-simulation may also occur when T_2 executes an atomic action. The relation is formalised as follows.

Definition 3. *Let T_1, T_2 be virtual machines with $A_1 = A_2$ and $A_1^{\mathsf{atom}} \supseteq A_2^{\mathsf{atom}}$.*

- *A* split simulation *of T_1 by T_2 is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$, such for all $s_1\ \mathcal{R}\ s_2$, if $s_1 \xrightarrow{\lambda} s_1'$ then one of the following holds:*
  - *$\lambda = a?$ or $\lambda = a\uparrow v$ with $a \in A_1^{\mathsf{split}} \cup A_2^{\mathsf{atom}}$; then $s_2 \xRightarrow{\lambda} s_2'$ with $s_1'\ \mathcal{R}\ s_2'$.*
  - *$\lambda = a\uparrow v$ with $a \in A_1^{\mathsf{atom}} \cap A_2^{\mathsf{split}}$; then $s_2 \xRightarrow{a?}\xRightarrow{a\uparrow v} s_2'$ such that $s_1'\ \mathcal{R}\ s_2'$.*
  - *$\lambda = \tau$; then $s_1'\ \mathcal{R}\ s_2$.*
- *An* atomic simulation *of T_2 by T_1 is an $(A_1^{\mathsf{atom}} \rightharpoonup \mathit{Val})$-indexed family $(\mathcal{Q}^\phi)_\phi$ of binary relations $\mathcal{Q}^\phi \subseteq S_2 \times S_1$, such that for all $s_2\ \mathcal{Q}^\phi\ s_1$, $\mathrm{dom}(\phi) \subseteq A_{s_2}$, and if $s_2 \xrightarrow{\lambda} s_2'$ then one of the following holds:*
  - *$\lambda = a?$ where $a \in A_1^{\mathsf{atom}}$, or $\lambda = \tau$; then $s_2'\ \mathcal{Q}^\phi\ s_1$.*
  - *$\lambda = a\uparrow v$ where $a \in \mathrm{dom}(\phi)$ and $v = \phi(a)$; then $s_2'\ \mathcal{Q}^{\phi\setminus a}\ s_1$.*
  - *$\lambda = a\uparrow v$ where $a \in A_1^{\mathsf{atom}} \setminus \mathrm{dom}(\phi)$; then there is a vector of distinct actions $\boldsymbol{c} \in (A_{s_2} \setminus \mathrm{dom}(\phi))^*$ running in s_2, such that $s_1 \xRightarrow{\boldsymbol{c}\uparrow \boldsymbol{w}}\xRightarrow{a\uparrow v} s_1'$ where $s_2'\ \mathcal{Q}^{\phi\{\boldsymbol{w}/\boldsymbol{c}\}}\ s_1'$.*
  - *$\lambda = a?$ or $\lambda = a\uparrow v$ where $a \in A_1^{\mathsf{split}}$; then $s_1 \xRightarrow{\lambda} s_1'$ such that $s_2'\ \mathcal{Q}^\phi\ s_1'$.*
- *A relation $\mathcal{Q} \subseteq S_2 \times S_1$ is said to be* coupled on idle states *to a relation $\mathcal{R} \subseteq S_1 \times S_2$ if whenever s_2 is an idle state, $s_2\ \mathcal{Q}\ s_1$ implies $s_1 \Rightarrow s_1'$ such that $s_1'\ \mathcal{R}\ s_2$. A* coupled split/atomic (s/a) simulation pair *is a pair $\mathcal{R}, (\mathcal{Q}^\phi)_\phi$ where $\mathcal{R}$ is a split simulation of T_1 by T_2, $(\mathcal{Q}^\phi)_\phi$ is an atomic simulation of T_2 by T_1, $\mathcal{R}$ is coupled to $\mathcal{Q}^\perp$ and $\mathcal{Q}^\perp$ is coupled to $\mathcal{R}$ on idle states. The largest coupled s/a simulation pair is denoted $\mathcal{S}, (\mathcal{A}^\phi)_\phi$.*

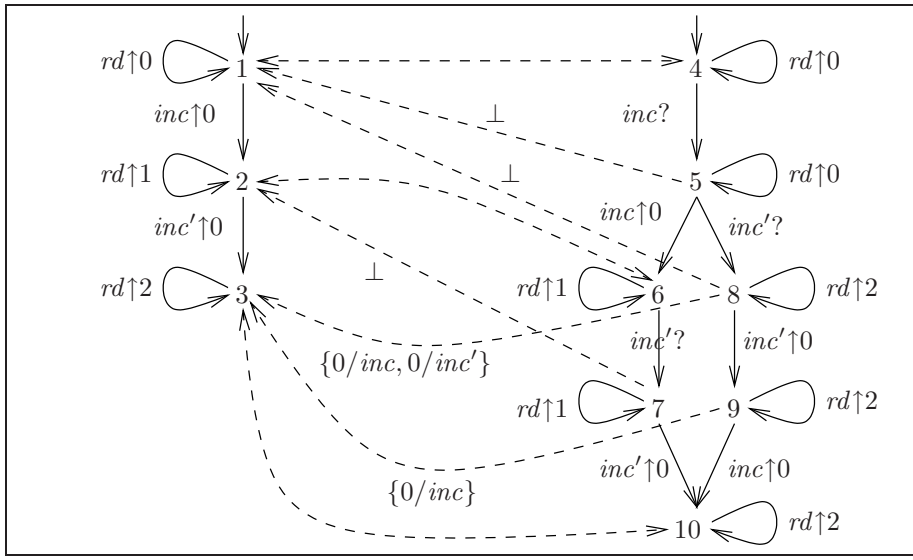


Fig. 3. An example of s/a refinement

– Coupled split/atomic simulation refinement *(or* s/a refinement *for short) between T_1 and T_2 is defined as $\preccurlyeq = \mathcal{S} \cap \mathcal{A}^{\perp\,-1}$, lifted to transition systems as before.*

The role of the index ϕ in the atomic simulation is to store which actions of the concrete machine were "pre-simulated" and with which return values. If $T_1 \preccurlyeq T_2$, we say that T_1 *contracts the actions* of T_2 (or actually only some of them, namely those in $A_1^{\mathsf{atom}} \cap A_2^{\mathsf{split}}$) —hence the title of this paper.

Example 3. Figure 3 demonstrates s/a refinement using behaviour *c*) of Fig. 2 and its atomic counterpart discussed in Ex. 2. In contrast to earlier figures, for the sake of completeness we have included return values for inc, inc' — which are all set to a standard value, in this case 0.

It can be seen that the start of inc and inc' in the concrete system is not simulated (the abstract system is silent). Furthermore, in order to match the transition $8 \xrightarrow{rd\uparrow 2} 8$ given $8\ \mathcal{A}^{\perp}\ 1$, the abstract system has to pre-simulate both inc and inc'; the matching abstract transition sequence is given by $1 \xrightarrow{inc\uparrow 0} \xrightarrow{inc'\uparrow 0} \xrightarrow{rd\uparrow 2} 3$ and $8\ \mathcal{A}^{\phi}\ 3$, where $\phi = \{0/inc, 0/inc'\}$ records the pre-simulated transitions and their return values. Accordingly, both $8 \xrightarrow{inc'\uparrow 0} 9$ and $9 \xrightarrow{inc\uparrow 0} 10$ are matched by the abstract system remaining silent in state 3, due to $9\ \mathcal{A}^{\{0/inc\}}\ 3$ and $10\ \mathcal{A}^{\perp}\ 3$. Pre-simulation also takes place when simulating $8 \xrightarrow{inc'\uparrow 0} 9$ starting from $8\ \mathcal{A}^{\perp}\ 1$; this is matched by $1 \xrightarrow{inc\uparrow 0} \xrightarrow{inc'\uparrow 0} 3$, noting $9\ \mathcal{A}^{\{0/inc\}}\ 3$.

It can be proved that $\preccurlyeq$ is transitive, hence a pre-order; moreover, $\preccurlyeq$ is implied by (ordinary) coupled similarity (however, it does not *coincide* with coupled similarity, even between virtual machines with the same atomic alphabets, because $\preccurlyeq$ requires coupling only on idle states).

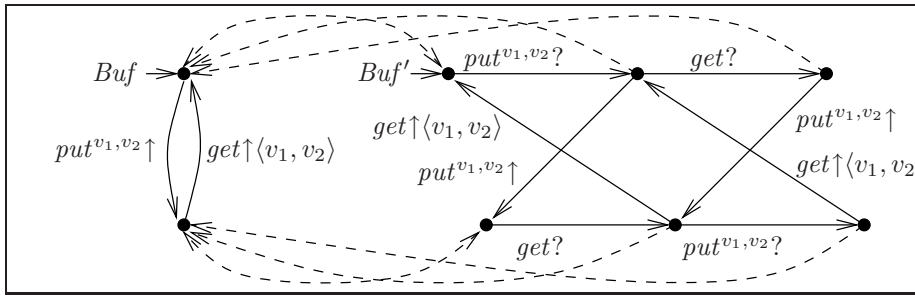


Fig. 4. Split/atomic simulation of a buffer

Proposition 3. *$\preccurlyeq \circ \preccurlyeq = \preccurlyeq$ and $\approx \subseteq \preccurlyeq$.*

We now come to the main result of this paper, which states that s/a refinement implies indistinguishability (up to coupled similarity). For the proof we have to refer to the report version [25]. T_{ag} is arbitrary with $A_{\mathsf{ag}} \supseteq A_1$.

Theorem 1. *If $T_1 \preccurlyeq T_2$, then $T_{\mathsf{ag}} \| T_1 \approx T_{\mathsf{ag}} \| T_2$.*

Theorem 1 has an important side benefit: if an action $a \in A_2^{\mathsf{split}}$ *can* be contracted, for instance according to $T_1 \preccurlyeq T_2$ where $a \in A_1^{\mathsf{atom}}$, *we no longer have to consider non-atomic invocation of that action*, even when T_{ag} runs on T_2; for according to Th. 1, every invocation of a has the same effect in T_2 as in T_1 (where it is executed atomically). In other words, we may contract all agents under consideration with respect to A_1^{atom}. As remarked before, this may drastically reduce the size of agents. The formal basis for this observation is the following corollary of Prop. 2 and Th. 1:

Corollary 1. *If $T_1 \preccurlyeq T_2$, then $T_{\mathsf{ag}} \| T_2 \approx \langle T_{\mathsf{ag}} \rangle_{A_1^{\mathsf{atom}}} \| T_2$.*

3.2 Example: A One-Place Buffer

In order to provide some more evidence for our claim that action contraction can be usefully applied in a variety of circumstances, we treat another small example, inspired by Langerak [18]. It concerns the implementation of a *buffer* with operations *put* (which inserts an element at the end of the buffer) and *get* (which extracts the first element from the buffer). To be able to treat the example in the available space, we limit ourselves to the case of a *one-place* buffer, called *Buf*. Its behaviour can be described in terms of atomic actions put^{v_1,v_2} (which puts the tuple $\langle v_1, v_2\rangle$ into the buffer and always returns a standard value) and *get* (which returns the tuple currently in the buffer). The required behaviour of the atomic virtual machine is given by the left hand side of Fig. 4.

The implementation carries through a design decision to transmit the two tuple elements v_1 and v_2 separately. As a consequence, the *put*- and *get*-actions are implemented non-atomically, since both consist of two buffer accesses. This results in a proposed buffer implementation, *Buf′*, depicted on the right hand

side of Fig. 4. In this implementation, *get* may start already before *put* has terminated; likewise, (the next) *put* may be invoked already before *get* has terminated. The figure indicates a coupled s/a simulation pair between *Buf* and Buf'; hence the implementation is correct in the sense of this paper.

4 Concluding Remarks

We have arrived at an increased understanding of the concept of atomicity in reactive systems. This was achieved by making a clear distinction between the asymmetric roles of client (or agent) and server (virtual machine). For the agent, the difference between the atomic and non-atomic invocation of an action is quite closely related to the exhaustively studied issue of *action refinement* (see also below). For a virtual machine, on the other hand, the situation is more complicated. In the notion of s/a refinement (Sect. 3.1), we have given a tractable characterisation of a correctness criterion for action abstraction that implies that no agent can distinguish between a given virtual machine and a correctly implemented, less atomic version of it. (It may be remarked at this point that a lot of fine-tuning was required to achieve this result; in a sense, all the aspects of the framework are put to work. For instance, the reliance on coupled similarity rather than bisimilarity appears crucial, as does the *pre-simulation* feature of atom simulation, the treatment of return values, the incausal shuffling in agents and the potential termination of virtual machines.)

We assert that the theory presented here can form the basis of a practically useful and realistic design technique whereby actions are at first specified atomically, and later implemented non-atomically.

Although for the purpose of clarity we have ruled out *auto-concurrency* in this conference paper, in fact the results do not depend on this; see [25] for a full treatment (which is achieved at the price of adding identifiers to distinguish between concurrent invocations/executions of the same action).

Open ends. An extension of the current framework begging to be studied is that of *refusal and abortion*. In the current paper we have taken without comment the usual process algebra viewpoint that a system may *refuse* to perform an action (a phenomenon also known as *limited service availability*). When the action is non-atomic, *refusal can only take place at start time*; we do not allow a non-atomic action to *abort* after it has started. This severely limits the applicability of the technique and puts the entire, very relevant theory of *transactions* out of reach; see, e.g., Lynch et al. [19].

There is a number of open questions regarding the precise properties of s/a refinement, such as:

- Can the reverse of Th. 1 be made to hold, maybe by a variation on $\preccurlyeq$?
- Is T_1 uniquely determined (up to $\approx$) by $T_1 \preccurlyeq T_2$?

Furthermore, in the introduction we already expressed our opinion that the theory of action abstraction fits very nicely in an object-based framework. There is clearly a lot of work to be done before this turns into reality.

Related work. This paper could not have been written in the absence of the past decade of research on action refinement. We have already included various references in the main text; worthwhile repeating is the legacy to the work on ST-semantics, introduced in [11] and later proved to give rise to coarsest congruences w.r.t. action refinement in [2,14]. At the same time, this paper presents some innovations w.r.t. the main body of action refinement research.

- Rather than deducing the refined behaviour from the abstract model —which mainly requires that the abstract model already contains enough information to make such deduction possible— we are more concerned with *correctness criteria*: what can be considered atomic once atomicity is gone?
- The interplay of data with action refinement never received much attention. Clearly it was felt that these were independent issues, and that such matters as data parameters and return values would not affect, nor be affected by, changes in the level of atomicity. At least in our framework this turns out not to be true: since the return value is not fixed at invocation time, it has to be modelled explicitly.

With much the same aim in mind as in the current paper, we previously developed a correctness notion based on *vertical implementation* in [26]. Another related line of research (unfortunately not followed up in recent years) is that of *interface refinement* as advocated in [9] and especially [6].

We have also strongly benefited from the previous development of *coupled simulation* as an alternative to bisimulation; indeed, we believe that adapting our main result (Theorem 1) to weak bisimilarity will require strengthening s/a refinement to such a degree that its usefulness will be greatly diminished. The point is that, in contrast to bisimulation, coupled simulation allows the non-atomic virtual machine to have active states that do not have a direct equivalent in the atomic machine. Coupled simulation has been used before in cases where bisimulation is too strong, e.g., in [23] to establish correctness of a distributed implementation of multi-party synchronisation, and in [22] to define correctness of choice encodings.

Last but not least, approaches similar to ours have been worked out in related research areas. While of necessity remaining very incomplete, we would like to mention again the work on atomic transactions in [19], as well as insights in atomicity to be gained from [5,17,16].

References

1. L. Aceto and M. C. B. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103:204–269, 1993. 295
2. L. Aceto and M. C. B. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115:179–247, 1994. 295, 303
3. J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990. 291
4. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987. 291, 294

5. G. Boudol and I. Castellani. Concurrency and atomicity. *Theoretical Comput. Sci.*, 59:25–84, 1988. 303
6. E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In S. Abramsky and T. S. E. Maibaum, eds., *TAPSOFT '91, Volume 2*, vol. 494 of *Lecture Notes in Computer Science*, pp. 297–312. Springer-Verlag, 1991. 303
7. F. Cherief and P. Schnoebelen. τ-bisimulation and full abstraction for refinement of actions. *Information Processing Letters*, 40:219–222, 1991. 297
8. W. R. Cleaveland, ed. *Concur '92*, vol. 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. 304
9. R. Gerth, R. Kuiper, and J. Segers. Interface refinement in reactive systems. In W. R. Cleaveland [8], pp. 77–93. 303
10. R. J. van Glabbeek. The linear time – branching time spectrum II: The semantics of sequential systems with silent moves. In E. Best, ed., *Concur '93*, vol. 715 of *Lecture Notes in Computer Science*, pp. 66–81. Springer-Verlag, 1993. 296
11. R. J. van Glabbeek and F. W. Vaandrager. Petri Net models for algebraic theories of concurrency. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, eds., *PARLE — Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, vol. 259 of *Lecture Notes in Computer Science*, pp. 224–242. Springer-Verlag, 1987. 295, 303
12. R. J. van Glabbeek and F. W. Vaandrager. The difference between splitting in n and $n+1$. *Information and Computation*, 136(2):109–142, 1997. 295
13. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, May 1996. 296, 297
14. R. Gorrieri and C. Laneve. Split and ST bisimulation semantics. *Information and Computation*, 116(1):272–288, Jan. 1995. 303
15. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 291, 294
16. S. Katz. Refinement with global equivalence proofs in temporal logic. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 29:59–78, 1997. 303
17. L. Lamport. The mutual exclusion problem, part I — a theory of interprocess communication. *J. ACM*, 33(2):313–326, Apr. 1986. 303
18. R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, University of Twente, Nov. 1992. 301
19. N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994. 302, 303
20. R. Milner. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980. 294
21. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 291
22. U. Nestmann and B. Pierce. Decoding choice encodings. In U. Montanari and V. Sassone, eds., *Concur '96: Concurrency Theory*, vol. 1119 of *Lecture Notes in Computer Science*, pp. 179–194. Springer-Verlag, 1996. 303
23. J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In W. R. Cleaveland [8], pp. 518–533. 296, 303
24. J. Parrow and P. Sjödin. The complete axiomatization of Cs-congruence. In R. Enjalbert, E. W. Mayr, and K. W. Wagner, eds., *STACS 94*, vol. 775 of *Lecture Notes in Computer Science*, pp. 557–568. Springer-Verlag, 1994. 296
25. A. Rensink. A theory of action contraction, 2000. Full version (including proofs): http://www.cs.utwente.nl/~rensink/contract.ps.gz. 292, 301, 302

26. A. Rensink and R. Gorrieri. Vertical implementation. *Information and Computation*, 2000 (to appear). Extended version of "Vertical Bisimulation" (TAPSOFT '97); see also Hildesheimer Informatik-Bericht 9/98, University of Hildesheim. 303
27. W. Vogler. Failures semantics based on interval semiwords is a congruence for refinement. *Distributed Computing*, 4:139–162, 1991. 295

A Theory of Testing for Markovian Processes

Marco Bernardo[1] and Rance Cleaveland[2]

[1] Università di Torino, Dipartimento di Informatica,
Corso Svizzera 185, 10149 Torino, Italy
bernardo@di.unito.it
[2] State University of New York at Stony Brook, Department of Computer Science,
Stony Brook, NY 11794-4400, USA
rance@cs.sunysb.edu

Abstract. We present a testing theory for Markovian processes in order to formalize a notion of efficiency which may be useful for the analysis of soft real time systems. Our Markovian testing theory is shown to enjoy close connections with the classical testing theory of De Nicola-Hennessy and the probabilistic testing theory of Cleaveland-Smolka et al. The Markovian testing equivalence is also shown to be coarser than the Markovian bisimulation equivalence. A fully abstract characterization is developed to ease the task of establishing testing related relationships between Markovian processes. It is then demonstrated that our Markovian testing equivalence, which is based on the (easier to work with) probability of executing a successful computation whose average duration is not greater than a given amount of time, coincides with the Markovian testing equivalence based on the (more intuitive) probability of reaching success within a given amount of time. Finally, it is shown that it is not possible to define a Markovian preorder which is consistent with reward based performance measures, thus justifying why a generic notion of efficiency has been considered.

1 Introduction

Markovian process algebras (see, e.g., [1,6,7,8]) provide a linguistic means to formally model and analyze performance characteristics of concurrent systems in the early stages of their design. Since timing aspects are described by means of exponentially distributed durations, the memoryless property of exponential distributions is exploited to define the semantics for these calculi in the classical interleaving style. Moreover, performance measures can be effectively computed on Markov chains derived by applying semantic rules to process terms.

Markovian process algebras are equipped with a notion of equivalence which relates terms possessing the same functional and performance properties. Markovian bisimulation equivalence, originally proposed in [8,7] and then further elaborated on in [1,6], constitutes a useful semantic theory as it has been proved to be the coarsest congruence contained in the intersection of a purely functional bisimulation equivalence and a purely performance bisimulation equivalence, to

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 305–319, 2000.

have a clear relationship with the aggregation technique for Markov chains known as ordinary lumping, and to have a sound and complete axiomatization.

From the performance standpoint, Markovian bisimulation equivalence relates terms having the same transient and steady state behavior. However, it would be useful to develop a notion of Markovian preorder which orders functionally equivalent terms according to their performance. As recognized in [9], an application of such a Markovian preorder would be the approximation of a term with another term whose underlying Markov chain is amenable to efficient solution, in the case in which the original term cannot be replaced with a suitable equivalent term. This would allow bounds on the performance of the original term to be efficiently derived.

In this paper, we extend the probabilistic testing framework of [4] by presenting a testing theory for Markovian processes based on a generic notion of efficiency (Sect. 2). Both Markovian processes and tests will be formally described in the Markovian process algebra EMPA [1]. Since EMPA allows for both exponentially timed actions and prioritized weighted immediate actions, Markovian testing preorders can be developed for continuous time processes, discrete time processes, and mixed processes where the duration of a transition can be either exponentially distributed or zero (Sect. 3).

The Markovian testing theory is shown to be a refinement of the classical testing theory of [5]: whenever a process passes a test with probability 1 (greater than 0) within some amount of time, then the process must (may) pass the test in the classical theory. The Markovian testing theory is also shown to be a refinement of the probabilistic testing theory of [4]. Moreover, the Markovian testing equivalence is proved to be coarser than the Markovian bisimulation equivalence. Since verifying that one process is related to another one requires a consideration of the behavior of both processes in the context of all possible tests, following [13] we also present a fully abstract characterization of the Markovian preorder based on extended traces. From such an alternative characterization, we derive a proof technique that simplifies the task of establishing preorder relationships between Markovian processes (Sect. 4).

The quantification of the probability that tests are passed within a given amount of time on which our Markovian testing theory relies is the probability of executing a successful computation whose average duration is not greater than a given amount of time. Such a quantification is relatively easy to work with. A more intuitive approach would quantify over the probability of reaching success within a given period of time. Such an approach is more difficult to treat formally because it requires handling nonexponential distributions. However, we show that the Markovian testing equivalences induced by the two different quantifications coincide, thus justifying why the testing theory has been developed for the less intuitive, but easier to work with, quantification (Sect. 5).

The paper concludes with some counterexamples which show that it is not possible to define a Markovian preorder that can be used to order processes according to reward based performance measures. This justifies the fact that a generic notion of efficiency has been considered (Sect. 6).

Due to space limitations, we present our testing theory only for continuous time Markovian processes. The interested reader is referred to [2] for discrete time and mixed Markovian processes, proofs of results, and comparisons with other efficiency preorders.

2 Efficiency Preorders for Concurrent Processes

The concept of efficiency for concurrent processes can be characterized both in terms of probability and in terms of time. A process is more efficient than another one if it is able to perform certain computations with a greater probability or by taking a lesser time. In our Markovian framework, both probabilistic and timing aspects are present, so we should set up a definition of efficiency which takes both into account.

Three types of Markovian processes can be identified: continuous time, discrete time, and mixed. They can be viewed as state transition graphs with initial state probabilities labeling states and execution rates or probabilities labeling transitions. Due to lack of space, in the following we consider only continuous time Markovian processes. In such a case, the duration of a transition is described by an exponentially distributed random variable X whose probability distribution function is $F_X(t) = 1 - e^{-\lambda \cdot t}$ with $\lambda \in \mathbb{R}_+$ called the rate and used as label of the transition. The execution probability of a transition turns out to be the ratio of its rate to the sum of the rates of the transitions having the same source state. According to the race policy, the sojourn time in a state is exponentially distributed as well with expected value equal to the reciprocal of the sum of the rates of the outgoing transitions. The average duration of a computation for a continuous time Markovian process is the sum of the average sojourn times of the visited states (except the last one), while the execution probability of a computation is the product of the execution probabilities of the involved transitions.

The notion of efficiency for Markovian processes we shall develop in this paper is based on a quantification of the probability with which processes pass tests within a given amount of time. More precisely, given a notion of success related to test passing, the quantification we shall consider is concerned with the probability of executing a successful computation whose average duration is not greater than a given amount of time. Based on such a notion of efficiency and the probabilistic testing theory of [4], in this paper we define a Markovian preorder in such a way that a Markovian process is less than another one if the probability that it executes a successful computation whose average duration is not greater than a given amount of time is less than the probability that the other process executes a successful computation whose average duration is not greater than the same amount of time. For notational conciseness, in the sequel Markovian processes will be formalized as terms in EMPA instead of as state transition graphs.

3 Extended Markovian Process Algebra

In this section we present actions, operators, and semantics for EMPA_{ct}, the sublanguage of EMPA [1] for continuous time Markovian processes. Each action $<a, \tilde{\lambda}>$ is characterized by its type, a, and its rate, $\tilde{\lambda}$. Its second component indicates the speed at which the action occurs and is used as a concise way to denote the random variable specifying the duration of the action. Based on rates, an action is classified as exponentially timed, if its rate is a positive real number, or passive, if its rate is left unspecified (denoted "$*$"). The set of actions is given by $Act_{\text{ct}} = AType \times ARate_{\text{ct}}$ where $AType$ is the set of types, including τ for invisible actions, and $ARate_{\text{ct}} = \mathbb{R}_+ \cup \{*\}$ is the set of rates.

Let $Const$ be a set of constants (ranged over by A) and let $ATRFun = \{\varphi : AType \longrightarrow AType \mid \varphi^{-1}(\tau) = \{\tau\}\}$ be a set of action type relabeling functions.

Definition 1. *The set $\mathcal{L}_{\text{ct}}$ of process terms of EMPA_{ct} is generated by the following syntax*

$$E ::= \underline{0} \mid <a, \tilde{\lambda}>.E \mid E/L \mid E[\varphi] \mid E + E \mid E \parallel_S E \mid A$$

where $L, S \subseteq AType - \{\tau\}$. Constant defining equations are written $A \triangleq E$. We denote by $\mathcal{G}_{\text{ct}}$ the set of closed and guarded terms. □

Term $\underline{0}$ cannot execute any action. Term $<a, \tilde{\lambda}>.E$ can execute action $<a, \tilde{\lambda}>$ and then behaves as E. Term E/L behaves as term E except that the type of each executed action is turned into τ whenever it belongs to L. Term $E[\varphi]$ behaves as term E except that the type a of each executed action becomes $\varphi(a)$. Term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed first. If the involved actions are exponentially timed, then the choice is resolved according to the race policy: the action sampling the least duration succeeds. If only passive actions are involved, then the choice is purely nondeterministic. Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 whose type does not belong to S and synchronously executes actions of E_1 and E_2 whose type belongs to S if at least one of the two actions is passive (the other action determines the rate of the resulting action).

The integrated interleaving semantics for EMPA_{ct} is represented by a labeled transition system (LTS for short) whose labels are actions. The two layer definition of the semantics given in Table 1 is needed to correctly handle transition rates without burdening transitions with auxiliary labels. Transition relation $\longrightarrow$, which is the least subset of $\mathcal{G}_{\text{ct}} \times Act_{\text{ct}} \times \mathcal{G}_{\text{ct}}$ satisfying the inference rule in the first part of Table 1, merges together the potential moves having the same action type, the same priority level, and the same derivative term by computing the resulting rate according to the race policy:

$$Melt(PM) = \{\!|(<a, \tilde{\lambda}>, E) \mid \exists \tilde{\mu} \in ARate_{\text{ct}}.\, (<a, \tilde{\mu}>, E) \in PM \wedge \tilde{\lambda} = Aggr\{\!|\, \tilde{\gamma} \mid (<a, \tilde{\gamma}>, E) \in PM \wedge PL(<a, \tilde{\gamma}>) = PL(<a, \tilde{\mu}>) \,|\!\}\}$$

where $PL(<a, *>) = -1$, $PL(<a, \lambda>) = 0$, $* \, Aggr \, * = *$, and $\lambda_1 \, Aggr \, \lambda_2 = \lambda_1 + \lambda_2$. The multiset [1] $PM(E) \in \mathcal{M}u_{fin}(Act_{\text{ct}} \times \mathcal{G}_{\text{ct}})$ of potential moves of

[1] We use "$\{\!|$" and "$|\!\}$" as brackets for multisets, "$\_ \oplus \_$" to denote multiset union, $\mathcal{M}u_{fin}(S)$ ($\mathcal{P}_{fin}(S)$) to denote the collection of finite multisets (sets) over set S, and $M(s)$ to denote the multiplicity of element s in multiset M.

Table 1. $\mathrm{EMPA}_{\mathrm{ct}}$ integrated interleaving semantics

$$\frac{(<a,\tilde{\lambda}>,E') \in Melt(PM(E))}{E \stackrel{a,\tilde{\lambda}}{\longrightarrow} E'}$$

$$\begin{aligned}
PM(\underline{0}) &= \emptyset \\
PM(<a,\tilde{\lambda}>.E) &= \{\!|\, (<a,\tilde{\lambda}>,E) \,|\!\} \\
PM(E/L) &= \{\!|\, (<a,\tilde{\lambda}>,E'/L) \mid (<a,\tilde{\lambda}>,E') \in PM(E) \wedge a \notin L \,|\!\} \oplus \\
&\quad \{\!|\, (<\tau,\tilde{\lambda}>,E'/L) \mid \exists a \in L.\, (<a,\tilde{\lambda}>,E') \in PM(E) \,|\!\} \\
PM(E[\varphi]) &= \{\!|\, (<\varphi(a),\tilde{\lambda}>,E'[\varphi]) \mid (<a,\tilde{\lambda}>,E') \in PM(E) \,|\!\} \\
PM(E_1 + E_2) &= PM(E_1) \oplus PM(E_2) \\
PM(E_1 \parallel_S E_2) &= \{\!|\, (<a,\tilde{\lambda}>,E_1' \parallel_S E_2) \mid a \notin S \wedge (<a,\tilde{\lambda}>,E_1') \in PM(E_1) \,|\!\} \oplus \\
&\quad \{\!|\, (<a,\tilde{\lambda}>,E_1 \parallel_S E_2') \mid a \notin S \wedge (<a,\tilde{\lambda}>,E_2') \in PM(E_2) \,|\!\} \oplus \\
&\quad \{\!|\, (<a,\tilde{\gamma}>,E_1' \parallel_S E_2') \mid a \in S \wedge \exists \tilde{\lambda}_1,\tilde{\lambda}_2 \in ARate. \\
&\qquad\qquad (<a,\tilde{\lambda}_1>,E_1') \in PM(E_1) \wedge \\
&\qquad\qquad (<a,\tilde{\lambda}_2>,E_2') \in PM(E_2) \wedge \\
&\qquad\qquad \tilde{\gamma} = Norm(a,\tilde{\lambda}_1,\tilde{\lambda}_2,PM(E_1),PM(E_2)) \,|\!\} \\
PM(A) &= PM(E) \qquad \text{if } A \stackrel{\Delta}{=} E
\end{aligned}$$

$E \in \mathcal{G}_{\mathrm{ct}}$ is defined by structural induction in the second part of Table 1 according to the intuitive meaning of the operators. In the rule for the parallel composition operator a normalization is carried out when computing the rates of the potential moves resulting from the synchronization of the same nonpassive action with several independent or alternative passive actions:

$$Norm(a,\tilde{\lambda}_1,\tilde{\lambda}_2,PM_1,PM_2) = \begin{cases} Split(\tilde{\lambda}_1, 1/(\pi_1(PM_2))(<a,*>)) \text{ if } \tilde{\lambda}_2 = * \\ Split(\tilde{\lambda}_2, 1/(\pi_1(PM_1))(<a,*>)) \text{ if } \tilde{\lambda}_1 = * \end{cases}$$

where $Split(*,p) = *$ and $Split(\lambda,p) = \lambda \cdot p$, while $\pi_1(PM)$ denotes the action multiset obtained by projecting the potential moves in PM on their first component. Applying $Aggr$ to the rates of the synchronizations involving the same nonpassive action gives as a result the rate of the nonpassive action itself, thus complying with the bounded capacity assumption.

Definition 2. *The integrated interleaving semantics of $E \in \mathcal{G}_{\mathrm{ct}}$ is the LTS $\mathcal{I}[\![E]\!] = (S_{E,\mathcal{I}}, Act_{\mathrm{ct}}, \longrightarrow_{E,\mathcal{I}}, E)$ where $S_{E,\mathcal{I}}$ is the least subset of $\mathcal{G}_{\mathrm{ct}}$ reachable from E via $\longrightarrow$ and $\longrightarrow_{E,\mathcal{I}}$ is the restriction of $\longrightarrow$ to $S_{E,\mathcal{I}} \times Act_{\mathrm{ct}} \times S_{E,\mathcal{I}}$. Term $E \in \mathcal{G}_{\mathrm{ct}}$ is performance closed iff $\mathcal{I}[\![E]\!]$ does not contain passive transitions. We denote by $\mathcal{E}_{\mathrm{ct}}$ the set of performance closed terms of $\mathcal{G}_{\mathrm{ct}}$.* □

From $\mathcal{I}[\![E]\!]$ we can derive two projected semantic models by simply dropping action rates and action types, respectively. We define below the functional one as it will be used in the following.

Definition 3. *The functional semantics of $E \in \mathcal{G}_{\rm ct}$ is the LTS $\mathcal{F}[\![E]\!] = (S_{E,\mathcal{F}}, AType, \longrightarrow_{E,\mathcal{F}}, E)$ where $S_{E,\mathcal{F}} = S_{E,\mathcal{I}}$ and $\longrightarrow_{E,\mathcal{F}}$ is the restriction of $\longrightarrow_{E,\mathcal{I}}$ to $S_{E,\mathcal{F}} \times AType \times S_{E,\mathcal{F}}$.* □

The Markovian bisimulation equivalence for $\text{EMPA}_{\rm ct}$ is defined as follows.

Definition 4. *Let function $Rate : (\mathcal{G}_{\rm ct} \times AType \times \{-1, 0\} \times \mathcal{P}(\mathcal{G}_{\rm ct})) \not\to ARate_{\rm ct}$ be defined by*

$$Rate(E, a, l, C) = Aggr\{\!|\, \tilde{\lambda} \mid \exists E' \in C.\, E \xrightarrow{a,\tilde{\lambda}} E' \wedge PL(<a, \tilde{\lambda}>) = l \,|\!\}$$

An equivalence relation $\mathcal{B} \subseteq \mathcal{G}_{\rm ct} \times \mathcal{G}_{\rm ct}$ is a strong extended Markovian bisimulation (strong EMB) iff, whenever $(E_1, E_2) \in \mathcal{B}$, then for all $a \in AType$, $l \in \{-1, 0\}$, and equivalence classes $C \in \mathcal{G}_{\rm ct}/\mathcal{B}$

$$Rate(E_1, a, l, C) = Rate(E_2, a, l, C)$$

The union $\sim_{\rm EMB}$ of all the strong EMBs is called the strong extended Markovian bisimulation equivalence. □

4 A Preorder for Continuous Time Markovian Processes

In this section we develop a testing preorder for continuous time Markovian processes by extending the probabilistic framework of [4]. Our objective is to relate continuous time Markovian processes whose behavior is completely specified from the performance point of view. Such processes never engage in passive actions, although they may have subprocesses that perform passive actions in the context of a synchronization with exponentially timed actions. Formally, continuous time Markovian processes are represented through the set $\mathcal{E}_{\rm ct,ndiv}$ of performance closed terms which are not divergent, i.e. not capable of performing a sequence of infinitely many internal actions. In the testing approach, processes are composed in parallel with tests forcing the synchronization over every observable action. Because of the synchronization discipline on action rates adopted in EMPA, tests are naturally expressed as terms composed of observable passive actions, which must synchronize with observable nonpassive actions of the process, and internal nonpassive actions, which introduce arbitrary delays and probabilistic branches. Formally, tests are represented via $\mathcal{T}_{\rm ct}$, the set of closed and guarded terms defined over the syntax of Def. 1 extended with the deadlocked term *success* but restricted to the action set $TAct_{\rm ct} = \{<a, \tilde{\lambda}> \in Act_{\rm ct} \mid (a \in AType - \{\tau\} \wedge \tilde{\lambda} = *) \vee (a = \tau \wedge \tilde{\lambda} \in \mathbb{R}_+)\}$, which are finite state and acyclic (hence divergence free).

Definition 5. *Let $E \in \mathcal{E}_{\rm ct,ndiv}$ and $T \in \mathcal{T}_{\rm ct}$.*

– *The interaction system of E and T is term $E \parallel_{AType-\{\tau\}} T$.*

- *A configuration is a state of* $\mathcal{I}[\![E \parallel_{AType-\{\tau\}} T]\!]$.
- *A configuration is successful iff its test component is success.*
- *A computation is a maximal sequence*
$$E \parallel_{AType-\{\tau\}} T \equiv s_0 \xrightarrow{a_1,\lambda_1} s_1 \xrightarrow{a_2,\lambda_2} \ldots \xrightarrow{a_n,\lambda_n} s_n$$
where configuration s_i *is not successful for any* $0 \leq i \leq n-1$.
- *A computation is successful iff so is its last configuration.*
- *We denote by* $\mathcal{C}(E,T)$ *and* $\mathcal{S}(E,T)$ *the set of computations and successful computations, respectively.* □

Note that the interaction system is a performance closed term. Moreover, since E is divergence free and T is finite state and acyclic, every computation of the interaction system has finite length. In the following, we use c to range over computations and s to range over configurations: $c = s$ denotes a computation composed of a single configuration. Furthermore, $Rate(E, AType, l, C)$ will stand for $Aggr\{\!|\, Rate(E, a, l, C) \mid a \in AType \,|\!\}$.

Definition 6. *Let* $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct}}$, $c \in \mathcal{C}(E,T)$, *and* $C \subseteq \mathcal{C}(E,T)$.

- $prob(c) = \begin{cases} 1 & \text{if } c = s \\ \frac{\lambda}{\Lambda} \cdot prob(c') & \text{if } c = s \xrightarrow{a,\lambda} c' \text{ with } \Lambda = Rate(s, AType, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $time(c) = \begin{cases} 0 & \text{if } c = s \\ \frac{1}{\Lambda} + time(c') & \text{if } c = s \xrightarrow{a,\lambda} c' \text{ with } \Lambda = Rate(s, AType, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $prob(C) = \sum_{c \in C} prob(c)$.
- $C_{\leq t} = \{c \in C \mid time(c) \leq t\}$. □

Definition 7. *Let* $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$.

- $E_1 \sqsubseteq_{\mathrm{MT}} E_2$ *iff* $\forall T \in \mathcal{T}_{\mathrm{ct}}. \forall t \in \mathbf{R}_+. prob(\mathcal{S}_{\leq t}(E_1, T)) \leq prob(\mathcal{S}_{\leq t}(E_2, T))$.
- $E_1 \sim_{\mathrm{MT}} E_2$ *iff* $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \wedge E_2 \sqsubseteq_{\mathrm{MT}} E_1$. □

We shall say that $E_1 \sqsubseteq_{\mathrm{MT}} E_2$ ($E_1 \sim_{\mathrm{MT}} E_2$) w.r.t. test T if the related condition in the definition above holds true when $\mathcal{T}_{\mathrm{ct}}$ is restricted to $\{T\}$.

Some remarks are now in order. First of all, note that in the definition of $time(c)$ we consider for each transition of c the average sojourn time of the source state instead of the rate of the transition. In fact, if we consider $E_1 \equiv <a, \lambda>.\underline{0} + <a, \lambda>.\underline{0}$ and $E_2 \equiv <a, 2 \cdot \lambda>.\underline{0}$, then $E_1 \sim_{\mathrm{MT}} E_2$ w.r.t. $T \equiv <a, *>.success$. If we considered rates instead, then we would obtain $prob(\mathcal{S}_{\leq t}(E_1, T)) = 0 < 1 = prob(\mathcal{S}_{\leq t}(E_2, T))$ for $\frac{1}{2 \cdot \lambda} \leq t < \frac{1}{\lambda}$. The reason why the average sojourn times of the traversed states are used instead of the rates of the executed transitions is that durations are described in a stochastic way, not in a deterministic way.

$\sqsubseteq_{\mathrm{MT}}$ and $\sim_{\mathrm{MT}}$ retain the link between the functional part and the performance part of every action, which is a necessary condition to achieve compositionality [1]. If we consider $E_1 \equiv <a, \lambda>.\underline{0} + <b, \mu>.\underline{0}$ and $E_2 \equiv <a, \mu>.\underline{0} + <b, \lambda>.\underline{0}$ where $\lambda < \mu$, then $E_1 \sqsubseteq_{\mathrm{MT}} E_2$ w.r.t. $T \equiv <a, *>.success + <b, *>.\underline{0}$, but $E_2 \not\sqsubseteq_{\mathrm{MT}} E_1$ w.r.t. T. As a consequence, $E_1 \not\sim_{\mathrm{MT}} E_2$ as expected.

Finally, we observe that allowing for internal, exponentially timed actions in tests increases the discriminating power. If we consider $E_1 \equiv <a, \lambda/2>.<b, \mu>.\underline{0} + <a, \lambda/2>.\underline{0}$ and $E_2 \equiv <a, \lambda>.<b, \mu>.\underline{0}$, then $E_1 \sqsubseteq_{\mathrm{MT}} E_2$ w.r.t. $T_1 \equiv <a, *>.<b, *>.success$ while $E_1 \not\sqsubseteq_{\mathrm{MT}} E_2$ w.r.t. $T_2 \equiv <a, *>.(<\tau, \gamma>.success + <b, *>.\underline{0})$.

4.1 Relationship with Classical and Probabilistic Testing

We now show that the Markovian testing preorder is a strict refinement of both the classical testing preorder of [5] and the probabilistic testing preorder of [4]. As far as the classical testing preorder is concerned, this can be reformulated for $\mathrm{EMPA}_{\mathrm{ct}}$ terms by considering their functional semantics.

Definition 8. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$ and $T \in \mathcal{T}_{\mathrm{ct}}$.*

- *An $\mathcal{F}$-configuration is a state of $\mathcal{F}[\![E \parallel_{AType-\{\tau\}} T]\!]$.*
- *An $\mathcal{F}$-configuration is successful iff its test component is success.*
- *An $\mathcal{F}$-computation is a maximal sequence*
$$E \parallel_{AType-\{\tau\}} T \equiv s_0 \xrightarrow{a_1}_{\mathcal{F}} s_1 \xrightarrow{a_2}_{\mathcal{F}} \ldots \xrightarrow{a_n}_{\mathcal{F}} s_n$$
where $\mathcal{F}$-configuration s_i is not successful for any $0 \leq i \leq n-1$.
- *An $\mathcal{F}$-computation is successful iff so is its last configuration.*
- *We denote by $\mathcal{C}_{\mathcal{F}}(E,T)$ and $\mathcal{S}_{\mathcal{F}}(E,T)$ the set of $\mathcal{F}$-computations and successful $\mathcal{F}$-computations, respectively.* □

Definition 9. *Let $E, E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$ and $T \in \mathcal{T}_{\mathrm{ct}}$.*

- *E may T iff $\mathcal{S}_{\mathcal{F}}(E,T) \neq \emptyset$.*
- *E must T iff $\mathcal{S}_{\mathcal{F}}(E,T) = \mathcal{C}_{\mathcal{F}}(E,T)$.*
- *$E_1 \sqsubseteq_{\mathrm{may}} E_2$ iff $\forall T \in \mathcal{T}_{\mathrm{ct}}.\ E_1$ may $T \Longrightarrow E_2$ may T.*
- *$E_1 \sqsubseteq_{\mathrm{must}} E_2$ iff $\forall T \in \mathcal{T}_{\mathrm{ct}}.\ E_1$ must $T \Longrightarrow E_2$ must T.*
- *$E_1 \sqsubseteq_{\mathrm{T}} E_2$ iff $E_1 \sqsubseteq_{\mathrm{may}} E_2 \wedge E_1 \sqsubseteq_{\mathrm{must}} E_2$.*
- *$E_1 \sim_{\mathrm{T}} E_2$ iff $E_1 \sqsubseteq_{\mathrm{T}} E_2 \wedge E_2 \sqsubseteq_{\mathrm{T}} E_1$.* □

Since the classical testing preorder is based on the notion of passing a test, we need an alternative characterization for the Markovian testing preorder which is explicitly based on the quantification of the probability of passing a test within a given amount of time.

Definition 10. *Let $E, E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct}}$, $p \in \mathbb{R}_{[0,1]}$, and $t \in \mathbb{R}_+$.*

- *E pass$_{p,t}$ T iff $prob(\mathcal{S}_{\leq t}(E,T)) \geq p$.*
- *$E_1 \leq_{\mathrm{MT}} E_2$ iff $\forall T \in \mathcal{T}_{\mathrm{ct}}.\ \forall p \in \mathbb{R}_{[0,1]}.\ \forall t \in \mathbb{R}_+.\ E_1$ pass$_{p,t}$ $T \Longrightarrow E_2$ pass$_{p,t}$ T.* □

Lemma 1. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longleftrightarrow E_1 \leq_{\mathrm{MT}} E_2$.* □

Lemma 2. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$ and $T \in \mathcal{T}_{\mathrm{ct}}$. Then*

(i) *E may $T \Longleftrightarrow \exists p \in \mathbb{R}_{[0,1]}.\ \exists t \in \mathbb{R}_+.\ E$ pass$_{p,t}$ T.*
(ii) *E must $T \Longleftrightarrow \exists t \in \mathbb{R}_+.\ E$ pass$_{1,t}$ T.* □

Theorem 1. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longrightarrow E_1 \sqsubseteq_{\mathrm{T}} E_2$.* □

The converse of Thm. 1 does not hold, i.e. classical testing is strictly more abstract than Markovian testing. In fact, if we consider $E_1 \equiv <a,\lambda>.<b,\gamma>.\underline{0} + <a,\mu>.<c,\delta>.\underline{0}$ and $E_2 \equiv <a,\mu>.<b,\gamma>.\underline{0} + <a,\lambda>.<c,\delta>.\underline{0}$ where $\lambda \neq \mu$, then $E_1 \sim_{\mathrm{T}} E_2$ because $\mathcal{F}[\![E_1]\!]$ is isomorphic to $\mathcal{F}[\![E_2]\!]$, but $E_1 \not\sim_{\mathrm{MT}} E_2$ w.r.t. $T \equiv <a,*>.<b,*>.\mathit{success}$.

The probabilistic testing preorder of [4] is defined as follows.

Definition 11. *Let $E, E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct}}$, and $p \in \mathbb{R}_{[0,1]}$.*

- *E pass_p T iff $\mathit{prob}(\mathcal{S}(E,T)) \geq p$.*
- *$E_1 \sqsubseteq_{\mathrm{PT}} E_2$ iff $\forall T \in \mathcal{T}_{\mathrm{ct}}.\, \forall p \in \mathbb{R}_{[0,1]}.\, E_1$ pass_p $T \Longrightarrow E_2$ pass_p T.*
- *$E_1 \sim_{\mathrm{PT}} E_2$ iff $E_1 \sqsubseteq_{\mathrm{PT}} E_2 \wedge E_2 \sqsubseteq_{\mathrm{PT}} E_1$.* □

Theorem 2. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longrightarrow E_1 \sqsubseteq_{\mathrm{PT}} E_2$.* □

The converse of Thm. 2 does not hold, i.e. probabilistic testing is strictly more abstract than Markovian testing. In fact, if we consider $E_1 \equiv <a,\lambda>.\underline{0} + <b,\mu>.\underline{0}$ and $E_2 \equiv <a,2\cdot\lambda>.\underline{0} + <b,2\cdot\mu>.\underline{0}$, then we have $E_1 \sim_{\mathrm{PT}} E_2$ because both transitions with action type a have execution probability $\frac{\lambda}{\lambda+\mu}$ and both transitions with action type b have execution probability $\frac{\mu}{\lambda+\mu}$, but $E_1 \not\sim_{\mathrm{MT}} E_2$ w.r.t. $T \equiv <a,*>.\mathit{success}$.

4.2 Alternative Characterization

In order to ease the task of establishing relationships between Markovian processes, following [13] we define an alternative characterization of the Markovian testing preorder which does not require an analysis of process behavior in response to tests. To simplify the presentation, we consider only the restricted class $\mathcal{T}_{\mathrm{ct},\tau}$ of tests without internal, exponentially timed actions.

The characterization is based on the notion of extended trace, which is a sequence of pairs where the first component can be interpreted as the set of (passive) actions enabled by the environment (i.e., the test) and the second component can be interpreted as the action which is actually executed. Since a process can contain internal, exponentially timed transitions, there may be several different ways in which a process can execute an extended trace.

Definition 12. *An extended trace is an element of set $\mathit{ETrace}_\tau = \{(M_1,a_1)\ldots(M_n,a_n) \in (2^{\mathit{AType}-\{\tau\}} \times (\mathit{AType}-\{\tau\}))^* \mid \forall i = 1,\ldots,n.\, a_i \in M_i\}$.* □

Definition 13. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$ and $\sigma = (M_1,a_1)\ldots(M_n,a_n) \in \mathit{ETrace}_\tau$.*

- *A computation of E compatible with σ is a sequence*
$$E \equiv E_0 \xrightarrow{(\tau,\lambda)^*} E'_0 \xrightarrow{a_1,\lambda_1} E_1 \ldots E_{n-1} \xrightarrow{(\tau,\lambda)^*} E'_{n-1} \xrightarrow{a_n,\lambda_n} E_n$$
where $\xrightarrow{(\tau,\lambda)^*}$ is a shorthand for finitely many (possibly zero) internal, exponentially timed transitions.
- *We denote by $\mathcal{CC}(E,\sigma)$ the set of computations of E compatible with σ.* □

Definition 14. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $\sigma = (M_1, a_1) \ldots (M_n, a_n) \in ETrace_\tau$, $c \in \mathcal{CC}(E, \sigma)$, and $C \subseteq \mathcal{CC}(E, \sigma)$.*

- $prob(c, \sigma) = \begin{cases} 1 & \text{if } c = E_n \\ \frac{\lambda}{\Lambda} \cdot prob(c', \sigma) & \text{if } c = E'_{i-1} \xrightarrow{a_i, \lambda_i} c' \text{ with } \lambda = \lambda_i \wedge \Lambda = Rate(E'_{i-1}, M_i \cup \{\tau\}, 0, \mathcal{E}_{\mathrm{ct}}) \\ & \text{or } c = E_{i-1} \xrightarrow{\tau, \lambda} c' \text{ with } \Lambda = Rate(E_{i-1}, M_i \cup \{\tau\}, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $time(c, \sigma) = \begin{cases} 0 & \text{if } c = E_n \\ \frac{1}{\Lambda} + time(c', \sigma) & \text{if } c = E'_{i-1} \xrightarrow{a_i, \lambda_i} c' \text{ with } \Lambda = Rate(E'_{i-1}, M_i \cup \{\tau\}, 0, \mathcal{E}_{\mathrm{ct}}) \\ & \text{or } c = E_{i-1} \xrightarrow{\tau, \lambda} c' \text{ with } \Lambda = Rate(E_{i-1}, M_i \cup \{\tau\}, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $prob(C) = \sum_{c \in C} prob(c, \sigma)$.
- $C_{\leq t} = \{c \in C \mid time(c, \sigma) \leq t\}$. □

Computing the aggregated rates w.r.t. $M_i \cup \{\tau\}$ instead of M_i in Def. 14 allows the internal transitions of a process to be correctly taken into account.

Definition 15. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$.*

- $E_1 \ll_{\mathrm{MT}} E_2$ *iff* $\forall \sigma \in ETrace_\tau . \forall t \in \mathbb{R}_+ . prob(\mathcal{CC}_{\leq t}(E_1, \sigma)) \leq prob(\mathcal{CC}_{\leq t}(E_2, \sigma))$.
- $E_1 \approx_{\mathrm{MT}} E_2$ *iff* $E_1 \ll_{\mathrm{MT}} E_2 \wedge E_2 \ll_{\mathrm{MT}} E_1$. □

We now prove that $\ll_{\mathrm{MT}}$ coincides with $\sqsubseteq_{\mathrm{MT}}$ in the case of tests without internal, exponentially timed actions. In other words, we prove that $\ll_{\mathrm{MT}}$ is fully abstract w.r.t. $\sqsubseteq_{\mathrm{MT}}$. In order to prove that $\sqsubseteq_{\mathrm{MT}} \subseteq \ll_{\mathrm{MT}}$, we construct a test from a trace in such a way that both of them have the same probabilistic and timing characteristics.

Definition 16. *We define the set of tests $\mathcal{T}_{\mathrm{ct}, ETrace_\tau} = \{T(\sigma) \in \mathcal{T}_{\mathrm{ct},\tau} \mid \sigma = (M_1, a_1) \ldots (M_n, a_n) \in ETrace_\tau\}$ as follows*

$$T(\sigma) \stackrel{\Delta}{=} T_1(\sigma)$$
$$T_i(\sigma) \stackrel{\Delta}{=} <a_i, *>.T_{i+1}(\sigma) + \sum_{b \in M_i - \{a_i\}} <b, *>.\underline{0}, \quad 1 \leq i < n$$
$$T_n(\sigma) \stackrel{\Delta}{=} <a_n, *>.success + \sum_{b \in M_n - \{a_n\}} <b, *>.\underline{0}$$ □

Theorem 3. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longrightarrow E_1 \ll_{\mathrm{MT}} E_2$.* □

In order to prove that $\ll_{\mathrm{MT}} \subseteq \sqsubseteq_{\mathrm{MT}}$, we observe that a test has some number of successful test executions leading from its initial state to one of its successful states. When a process interacts with a test, each successful computation exercises exactly one of the successful test executions. Because of internal, exponentially timed transitions of processes, there may be several successful computations exercising the same successful test execution.

Definition 17. *Let $T \in \mathcal{T}_{\mathrm{ct},\tau}$. An execution ξ of T is a sequence*
$$T \equiv T_0 \xrightarrow{a_1,*} \ldots \xrightarrow{a_n,*} T_n$$
with associated extended trace $etrace(\xi) = (M_1, a_1) \ldots (M_n, a_n)$ where $M_i = \{a \in AType \mid T_i \xrightarrow{a,*}\}$ for all $1 \leq i \leq n$. □

Definition 18. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct},\tau}$, ξ be an execution of T, and $t \in \mathbb{R}_+$.*

– *We denote by $\mathcal{C}(E,T,\xi) = \{c \in \mathcal{C}(E,T) \mid \xi = proj_T(c)\}$ the set of computations exercising ξ, where*
$$proj_T(c) = \begin{cases} T' & \text{if } c = E' \parallel_{AType-\{\tau\}} T' \\ proj_T(c') & \text{if } c = E' \parallel_{AType-\{\tau\}} T' \xrightarrow{a,\lambda} c' \text{ with } E'' \parallel_{AType-\{\tau\}} T' \text{ first config. of } c' \\ T' \xrightarrow{a,*} proj_T(c') & \text{if } c = E' \parallel_{AType-\{\tau\}} T' \xrightarrow{a,\lambda} c' \text{ with } E'' \parallel_{AType-\{\tau\}} T'' \text{ first config. of } c' \text{ and } T'' \neq T' \end{cases}$$

– $\mathcal{C}_{\leq t}(E,T,\xi) = \{c \in \mathcal{C}_{\leq t}(E,T) \mid \xi = proj_T(c)\}$. □

Theorem 4. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \ll_{\mathrm{MT}} E_2 \Longrightarrow E_1 \sqsubseteq_{\mathrm{MT}} E_2$.* □

Corollary 1. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longleftrightarrow E_1 \ll_{\mathrm{MT}} E_2$.* □

Corollary 2. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \Longleftrightarrow E_1 \sqsubseteq_{\mathrm{MT}} E_2$ w.r.t. $\mathcal{T}_{\mathrm{ct},ETrace_\tau}$.* □

The last result means that $\mathcal{T}_{\mathrm{ct},ETrace_\tau}$ is a class of essential tests, i.e. tests that expose all relevant aspects of process behavior.

We now show a proof technique based on the fully abstract characterization above which eases the task of proving $E_1 \sqsubseteq_{\mathrm{MT}} E_2$.

Definition 19. *Let $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$ and $\sigma = (M_1, a_1) \ldots (M_n, a_n) \in ETrace_\tau$.*

– *The functional trace semantics of E is the set $\mathcal{FT}[\![E]\!] = \{a_1 \ldots a_n \in (AType - \{\tau\})^* \mid E \equiv E_0 \xrightarrow{\tau^*}_{\mathcal{F}} E'_0 \xrightarrow{a_1}_{\mathcal{F}} E_1 \ldots E_{n-1} \xrightarrow{\tau^*}_{\mathcal{F}} E'_{n-1} \xrightarrow{a_n}_{\mathcal{F}} E_n\}$.*
– *The trace of σ is the sequence $trace(\sigma) = a_1 \ldots a_n$.*
– *The functional extended trace semantics of E is the set $\mathcal{FET}[\![E]\!] = \{\sigma \in ETrace_\tau \mid trace(\sigma) \in \mathcal{FT}[\![E]\!]\}$.* □

Theorem 5. *Let $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. Then $E_1 \ll_{\mathrm{MT}} E_2 \Longleftrightarrow \mathcal{FT}[\![E_1]\!] \subseteq \mathcal{FT}[\![E_2]\!] \wedge \forall \sigma \in \mathcal{FET}[\![E_2]\!]. \forall t \in \mathbb{R}_+. prob(\mathcal{CC}_{\leq t}(E_1, \sigma)) \leq prob(\mathcal{CC}_{\leq t}(E_2, \sigma))$.* □

On the basis of the theorem above and the full abstraction result of Cor. 1, we have the following proof technique for verifying whether $E_1 \sqsubseteq_{\mathrm{MT}} E_2$:

– Check if $\mathcal{FT}[\![E_1]\!] \subseteq \mathcal{FT}[\![E_2]\!]$.
– Check if $\forall \sigma \in \mathcal{FET}[\![E_2]\!]. \forall t \in \mathbb{R}_+. prob(\mathcal{CC}_{\leq t}(E_1, \sigma)) \leq prob(\mathcal{CC}_{\leq t}(E_2, \sigma))$.

In the case of general tests containing internal, exponentially timed actions, the alternative characterization is given in terms of extended traces which keep track of internal, exponentially timed transitions performed by tests and therefore record not only action types but also action rates. See [2].

4.3 Relationship with Markovian Bisimulation Equivalence

The Markovian testing equivalence is strictly coarser than the Markovian bisimulation equivalence.

Theorem 6. *Let* $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. *Then* $E_1 \sim_{\mathrm{EMB}} E_2 \Longrightarrow E_1 \sim_{\mathrm{MT}} E_2$. □

The converse of Thm. 6 does not hold. In fact, if we consider terms

$$E_1 \equiv <a, \lambda_1>.<b, \mu>.<c, \gamma>.\underline{0} + <a, \lambda_2>.<b, \mu>.<d, \delta>.\underline{0}$$
$$E_2 \equiv <a, \lambda>.(<b, \mu_1>.<c, \gamma>.\underline{0} + <b, \mu_2>.<d, \delta>.\underline{0})$$

where $\lambda_1 = \lambda_2 = \lambda/2$, $\mu_1 = \mu_2 = \mu/2$, and $\gamma \leq \delta$, then it can be proved that $E_1 \sim_{\mathrm{MT}} E_2$ with the proof technique of the previous section, but $E_1 \not\sim_{\mathrm{EMB}} E_2$.

It can be shown that in general $\sim_{\mathrm{EMB}}$ is not a congruence for EMPA; in particular, the replacement of a parallel component by a Markovian bisimilar one will not in general preserve $\sim_{\mathrm{EMB}}$. When certain reasonable restrictions are imposed on the degree of internal nondeterminism among passive actions of the same type, however, this difficulty disappears [1]. It also turns out that essential tests characterizing $\sim_{\mathrm{MT}}$ obey this restriction; consequently, the testing theory of this paper is not affected by the technical shortcomings of $\sim_{\mathrm{EMB}}$ in EMPA.

5 An Alternative Approach to Markovian Testing

The notion of efficiency for Markovian processes developed in the previous sections is based on the probability of executing a successful computation whose average duration is not greater than a given amount of time. This quantification of the probability with which tests are passed within a given amount of time has been chosen because it is relatively easy to work with. On the other hand, it is not the most intuitive one, because it is different from the probability of reaching success within a given amount of time. The problem with this quantification is that it involves linear combinations of hypoexponential distributions [11] instead of numbers, so it is more difficult to deal with.

In this section we formalize the Markovian testing preorder arising from the latter quantification and we show that the Markovian testing equivalences induced by the two different quantifications coincide. This relationship between the two different quantifications, although partial in that it does not relate the two preorders, is important because it justifies the development of the testing theory for the less intuitive quantification and allows us to automatically derive that the properties we have proved for (the easier to work with) $\sim_{\mathrm{MT}}$ hold for the alternative Markovian testing equivalence as well.

Let us preliminarily redefine our Markovian testing preorder in the following equivalent way.

Definition 20. *Let* $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct}}$, $c \in \mathcal{C}(E,T)$, $C \subseteq \mathcal{C}(E,T)$, *and* $t \in \mathbb{R}_+$.

- $prob(c) = \begin{cases} 1 & \text{if } c = s \\ \frac{\lambda}{\Lambda} \cdot prob(c') & \text{if } c = s \xrightarrow{a,\lambda} c' \text{ with } \Lambda = Rate(s, AType, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $time(c) = \begin{cases} 0 & \text{if } c = s \\ \frac{1}{\Lambda} + time(c') & \text{if } c = s \xrightarrow{a,\lambda} c' \text{ with } \Lambda = Rate(s, AType, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $prob(C,t) = \sum_{c \in C} prob(c) \cdot \Pr(time(c) \leq t)$. □

Note that, for all computations c, $\Pr(time(c) \leq t) \in \{0, 1\}$. Thus $prob(C,t)$ represents the probability of executing a computation in C whose average duration is not greater than t.

Definition 21. *Let* $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$.

- $E_1 \sqsubseteq_{\mathrm{MT}} E_2$ *iff* $\forall T \in \mathcal{T}_{\mathrm{ct}}. \forall t \in \mathbf{R}_+. prob(\mathcal{S}(E_1, T), t) \leq prob(\mathcal{S}(E_2, T), t)$.
- $E_1 \sim_{\mathrm{MT}} E_2$ *iff* $E_1 \sqsubseteq_{\mathrm{MT}} E_2 \wedge E_2 \sqsubseteq_{\mathrm{MT}} E_1$. □

The alternative Markovian testing preorder is defined as follows.

Definition 22. *Let us denote by* X_Λ *an exponentially distributed random variable with rate* Λ. *Let* $E \in \mathcal{E}_{\mathrm{ct,ndiv}}$, $T \in \mathcal{T}_{\mathrm{ct}}$, $c \in \mathcal{C}(E,T)$, $C \subseteq \mathcal{C}(E,T)$, *and* $t \in \mathbf{R}_+$.

- $time'(c) = \begin{cases} 0 & \text{if } c = s \\ X_\Lambda + time'(c') & \text{if } c = s \xrightarrow{a,\lambda} c' \text{ with } \Lambda = Rate(s, AType, 0, \mathcal{E}_{\mathrm{ct}}) \end{cases}$
- $prob'(C,t) = \sum_{c \in C} prob(c) \cdot \Pr(time'(c) \leq t)$. □

Observe that $prob'(C,t)$ is the probability of reaching the end of a computation in C within time t. This probability is obtained by summing the products stemming from the multiplication of the probability of performing a given computation in C by the probability of completing that computation within time t, where the latter quantity follows a hypoexponential distribution (see the definition of $time'$).

Definition 23. *Let* $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$.

- $E_1 \preceq_{\mathrm{MT}} E_2$ *iff* $\forall T \in \mathcal{T}_{\mathrm{ct}}. \forall t \in \mathbf{R}_+. prob'(\mathcal{S}(E_1, T), t) \leq prob'(\mathcal{S}(E_2, T), t)$.
- $E_1 \simeq_{\mathrm{MT}} E_2$ *iff* $E_1 \preceq_{\mathrm{MT}} E_2 \wedge E_2 \preceq_{\mathrm{MT}} E_1$. □

Theorem 7. *Let* $E_1, E_2 \in \mathcal{E}_{\mathrm{ct,ndiv}}$. *Then* $E_1 \simeq_{\mathrm{MT}} E_2 \iff E_1 \sim_{\mathrm{MT}} E_2$. □

6 Impossibility Result

Given a Markov chain, performance measures on it are typically defined through rewards [10], so it would be desirable to define a Markovian preorder consistently with reward based performance measures. A performance measure for a Markov chain can be defined (in a simplified form) as a weighted sum $R = \sum_{i \in S} \rho_i \cdot \pi_i$ of the state probabilities $\boldsymbol{\pi}$ of the Markov chain where weights $\boldsymbol{\rho}$ are expressed

through real numbers, called rewards, associated with states. Unfortunately, the following counterexamples show that it is not possible to define a Markovian preorder which is consistent with reward based performance measures. Although we shall concentrate on steady state, instant-of-time performance measures, the impossibility result holds true also for transient state, instant-of-time performance measures as well as transient state, interval-of-time performance measures.

The first counterexample shows that, as one might expect, it is not possible to define a Markovian preorder which is consistent with all the reward based performance measures. Consider the two functionally equivalent terms E and F described below together with the related steady state probabilities:

$$E_1 \stackrel{\Delta}{=} <a, \lambda>.E_2 \quad E_2 \stackrel{\Delta}{=} <b, \mu>.E_1 \quad \pi_{E_1} = \mu/(\lambda+\mu) \quad \pi_{E_2} = \lambda/(\lambda+\mu)$$
$$F_1 \stackrel{\Delta}{=} <a, \gamma>.F_2 \quad F_2 \stackrel{\Delta}{=} <b, \delta>.F_1 \quad \pi_{F_1} = \delta/(\gamma+\delta) \quad \pi_{F_2} = \gamma/(\gamma+\delta)$$

and assume $\lambda \leq \gamma$ and $\mu \leq \delta$. Since the steady state probabilities of a Markov chain sum up to 1, we cannot have that $\pi_{E_1} \leq \pi_{F_1}$ and $\pi_{E_2} \leq \pi_{F_2}$. Suppose then $\pi_{E_1} \leq \pi_{F_1}$ and $\pi_{E_2} \geq \pi_{F_2}$. If we consider a performance measure defined through rewards $\rho_{E_1} = \rho_{F_1} = 1$ and $\rho_{E_2} = \rho_{F_2} = 0$, then $R_E = \pi_{E_1} \leq \pi_{F_1} = R_F$. If we consider instead a performance measure defined through rewards $\rho_{E_1} = \rho_{F_1} = 0$ and $\rho_{E_2} = \rho_{F_2} = 1$, then $R_E = \pi_{E_2} \geq \pi_{F_2} = R_F$.

The second counterexample shows that we do not succeed in defining a Markovian preorder even if we focus on a single reward based performance measure. Consider again terms E and F above and the performance measure defined through rewards $\rho_{E_1} = \rho_{F_1} = 1$ and $\rho_{E_2} = \rho_{F_2} = 0$. If $\lambda = 1$, $\gamma = 2$, $\mu = 3$, and $\delta = 4$, then $R_E = 0.75 > 0.\overline{6} = R_F$. If instead $\lambda = 1$, $\gamma = 1$, $\mu = 3$, and $\delta = 4$, then $R_E = 0.75 < 0.8 = R_F$.

The problem with the definition of a Markovian preorder which takes reward based performance measures into account is essentially that they are based on state probabilities and these are normalized to 1. This somehow causes state probabilities of different processes to be incomparable. This problem is further exacerbated by the fact that state probabilities depend on equations involving the global knowledge of all the rates, so it is not possible to rely on local information such as the relationships between rates of corresponding transitions. The only observation that can be made about E and F above is that, if we relate E_1 with F_1 and E_2 with F_2, then upon executing a the evolution of F_1 into F_2 is faster (on average) than the evolution of E_1 into E_2, because $\lambda \leq \gamma$, and upon executing b the evolution of F_2 into F_1 is faster (on average) than the evolution of E_2 into E_1, because $\mu \leq \delta$. This is exactly what is captured by our Markovian preorder based on a generic notion of efficiency.

7 Conclusion

In this paper we have developed a testing theory for Markovian processes and we have provided a fully abstract characterization of it which simplifies the task of establishing preorder relationships between Markovian processes. We have also justified why we have considered a generic notion of efficiency instead of reward

based performance measures, and why we have characterized the generic notion of efficiency through average durations instead of duration distributions.

As far as future work is concerned, an open problem is to establish whether a result like Thm. 7 holds for preorders as well. Moreover, we would like to investigate compositionality related issues for the Markovian testing preorder as well as find a sound and complete axiomatization. Additionally, a more denotational characterization based entirely on the structure of Markovian processes should be studied. A good starting point may be [12] where the acceptance tree model of [5] is adapted to probabilistic processes. Such a characterization would help devising algorithms for determining preorder relationships between Markovian processes, like in the case of the classical testing theory [3].

References

1. M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna, 1999 (`http://www.di.unito.it/~bernardo/`) 305, 306, 308, 311, 316
2. M. Bernardo, W. R. Cleaveland, *"A Theory of Efficiency for Markovian Processes"*, Tech. Rep. UBLCS-99-02, University of Bologna, 1999 307, 315
3. W. R. Cleaveland, M. C. B. Hennessy, *"Testing Equivalence as a Bisimulation Equivalence"*, in Formal Aspects of Computing 5:1-20, 1993 319
4. W. R. Cleaveland, S. A. Smolka, A. E. Zwarico, *"Testing Preorders for Probabilistic Processes"*, in Proc. of ICALP '92, LNCS 623:708-719, 1992 306, 307, 310, 312, 313
5. R. De Nicola, M. C. B. Hennessy, *"Testing Equivalences for Processes"*, in Theoretical Computer Science 34:83-133, 1983 306, 312, 319
6. H. Hermanns, *"Interactive Markov Chains"*, Ph.D. Thesis, University of Erlangen, 1998 305
7. H. Hermanns, M. Rettelbach, *"Syntax, Semantics, Equivalences, and Axioms for MTIPP"*, in Proc. of PAPM '94, pp. 71-87, Erlangen, 1994 305
8. J. Hillston, *"A Compositional Approach to Performance Modelling"*, Cambridge University Press, 1996 305
9. J. Hillston, *"Exploiting Structure in Solution: Decomposing Composed Models"*, in Proc. of PAPM '98, pp. 1-15, 1998 306
10. R. A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971 317
11. M. F. Neuts, *"Matrix-Geometric Solutions in Stochastic Models - An Algorithmic Approach"*, John Hopkins University Press, 1981 316
12. M. Nuñez, D. de Frutos, L. Llana, *"Acceptance Trees for Probabilistic Processes"*, in Proc. of CONCUR '95, LNCS 962:249-263, 1995 319
13. S. Yuen, W. R. Cleaveland, Z. Dayar, S. A. Smolka, *"Fully Abstract Characterizations of Testing Preorders for Probabilistic Processes"*, in Proc. of CONCUR '94, LNCS 836:497-512, 1994 306, 313

Reasoning about Probabilistic Lossy Channel Systems

Parosh Abdulla[1], Christel Baier[2], Purushothaman Iyer[3⋆], and Bengt Jonsson[1]

[1] Dept of Computer Systems, Uppsala University
Uppsala, Sweden
[2] Dept of Computer Science, University of Bonn
Bonn, Germany
[3] Dept of Computer Science, North Carolina State University
Raleigh, NC 27695, USA

Abstract. We consider the problem of deciding whether an infinite-state system (expressed as a Markov chain) satisfies a correctness property with probability 1. This problem is, of course, undecidable for general infinite-state systems. We focus our attention on the model of *probabilistic lossy channel systems* consisting of finite-state processes that communicating over unbounded lossy FIFO channels. Abdulla and Jonsson have shown that safety properties are decidable while progress properties are not for non-probabilistic lossy channel systems. Under assumptions of "sufficiently high" probability of loss, Baier and Engelen have shown how to check whether a property holds of probabilistic lossy channel system with probability 1.
In this paper we show that the problem of checking whether a progress property holds with probability 1 is undecidable, if the assumption about "sufficiently high" probability of loss is omitted. More surprisingly, we show that checking whether safety properties hold with probability 1 is undecidable too. Our proof depends upon simulating a perfect channel, with a high degree of confidence, using lossy channels.

1 Introduction

Finite state machines which communicate over unbounded FIFO channels have been used as an abstract model of computation for reasoning about communication protocols [4], and they form the backbone of ISO protocol specification languages Estelle and SDL. However, the model is turing-powerful which makes most verification problems of interest undecidable. Ever since the publication of the Alternating bit protocol it has been customary to assume, while modeling a protocol, that the communication channels between processes are free of errors. Possible errors in the communication channels are treated separately, or are completely ignored. However, over the past few years there have been attempts to rectify this situation to allow modeling of imperfections in the communication medium. Abdulla and Jonsson, in [2], considered a model where messages could

⋆ Suppoted in part by ARO under grant DAAG55-98-1-03093 and by STINT.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 320–333, 2000.

be lost from the queue, while waiting to be delivered. They showed that the *reachability* problem (and consequently, the problem of checking for safety properties) is decidable. However, in [1], they also showed that checking for progress properties is undecidable – a consequence of fairness arguments necessary to prove progress properties.

Randomization is an oft-used technique to provide tractable, approximate solutions to intractable problems [5,8]. Given that we are dealing with imperfections in the communication medium it is then natural to consider models of communicating processes where the probability of message loss is taken into account. The reason for considering such an approach is to supplant fairness arguments, necessary for showing progress properties, by probablistic arguments. Furthermore, by considering a probabilistic model we would be making our model more closer to reality. In [6], Iyer and Narasimha considered whether a probabilistic lossy channel systems satisfies an LTL formula with a probability greater than p, with in a tolerance ϵ. In [3] Baier and Engelen considered the following more general problem (which is the topic of this paper):

> given a probabilistic lossy channel system $\mathcal{L}$ and a linear-time temporal logic formula ϕ, does the set of sequences of $\mathcal{L}$ that satisfy ϕ have probability 1.

A partial answer to this question was given by Baier and Engelen who showed that it is decidable if the probability of message loss is sufficiently high. Here "sufficiently high" was taken to mean roughly "at least $\frac{1}{2}$". Under this restriction, it can be shown that (with probability 1) the number of messages in the channel cannot grow unboundedly, and that the general model checking problem can be reduced to checking reachability (which is decidable by [2]).

In this paper we will continue the work of Baier and Engelen by showing that the problem of model checking progress properties with probability 1 is undecidable, when we relax the assumption that messages are lost with "sufficiently high" probability. Our proof is a construction which shows that if the probability of losing each message is lower than $\frac{1}{2}$ then a lossy channel system can (with probability 1) simulate a finite state machine which operates on *perfect* FIFO channels. The main idea is to let each message transmission of the perfect channel system be simulated by a number of retransmissions of the message in the lossy channel system. If the scheme for retransmissions is chosen appropriately, the simulation will be faithful in a certain well-defined sense – a fact that which allows us to establish the undecidability result.

The rest of the paper is organized as follows. In the next two sections we present the necessary definitions for probabilistic lossy channel systems and the probabilistic reachability problem. In Section 4 we provide an overview of our undecidability proof, which we follow in Section 5 with the necessary constructions and in Section 6 with the proofs for the main lemmata.

2 Communicating Finite-State Machines

In this section we introduce communicating finite-state machines (CFSMs) and some of their properties.

For a set M we use M^* to denote the set of finite strings of elements in M. For $x \in M^*$, let $x(i)$ denote the i^{th} element of x. For $x, y \in M^*$ we let $x \bullet y$ denote the concatenation of x and y. The empty string is denoted by ε. For sets C and M, a *string vector from C to M* is a function $C \mapsto M^*$. For a string vector w from C to M we use $w[c := x]$ for the string vector w' such that $w'(c) = x$, and $w'(d) = w(d)$, for $d \neq c$. The string vector which maps all elements in C to the empty string is denoted ε.

Definition 1. *A* Communicating Finite-State Machine (CFSM) $\mathcal{C}$ *is a tuple* $\langle S, s_{init}, C, M, \delta \rangle$*, where*

S *is a finite set of* control states,
$s_{init} \in S$ *is an* initial control state,
C *is a finite set of* channels,
M *is a finite set of* messages,
δ *is a finite set of* transitions*, each of which is a triple of the form* $\langle s_1, op, s_2 \rangle$*, where* s_1 *and* s_2 *are control states, and op is either a label of form* $c!m$*,* $c?m$*, or empty, where* $c \in C$ *and* $m \in M$*.*

Given this finite description of a system, we can now talk about the *global state* γ of the $\mathcal{C}$ as a pair $\langle s, w \rangle$, where $s \in S$ and w is a string vector from C to M. The *initial global state* γ_0 of $\mathcal{C}$ is the pair $\langle s_0, \varepsilon \rangle$. The progression of the system from one global state to another can now be formalized as a transition relation $\longrightarrow \subseteq \gamma \times \delta\gamma$, with a typical member written as a triple of the form $\langle \gamma_1, t, \gamma_2 \rangle$, where γ_1 and γ_2 are global states, and $t \in \delta$. . The relation $\gamma_1 = \langle s_1, w_1 \rangle \xrightarrow{t} \gamma_2 = \langle s_2, w_2 \rangle$ holds iff one of the following conditions is satisfied.

1. $t = \langle s_1, c!m, s_2 \rangle$ and $w_2 = w_1[c := w_1(c) \bullet m]$.
2. $t = \langle s_1, c?m, s_2 \rangle$ and $w_1 = w_2[c := m \bullet w_2(c)]$.
3. $t = \langle s_1, empty, s_2 \rangle$ and $w_2 = w_1$.

For a global state γ, we define the set $en(\gamma)$ of *enabled* transitions at γ such that $t \in en(\gamma)$ iff $\gamma \xrightarrow{t} \gamma'$ for some γ'. We let $\longrightarrow = \cup_{t \in \delta} \xrightarrow{t}$ and let $\xrightarrow{*}$ denote the reflexive transitive closure of $\longrightarrow$. For global states γ_1 and γ_2, we say that γ_2 is *reachable* from γ_1 if $\gamma_1 \xrightarrow{*} \gamma_2$. For a global state γ, we say that γ is *reachable* if γ is reachable from γ_0.

A *computation* π is a sequence (either finite or infinite) of states and transitions: $\gamma_1\ t_1 \gamma_2\ t_2\ \gamma_3\ \cdots\ t_{n-1}\gamma_n \ldots$ such that $\gamma_i \xrightarrow{t_i} \gamma_{i+1}$ for each $i : 1 \leq i$. We will use $\pi(i)$ to denote the i^{th} state γ_i visited in π. If a computation π is finite we say that π *leads* from γ_1 to γ_n[1].

We define the control state reachability problem for CFSMs (Reach-CFSM) as follows

[1] Note that γ' is reachable from γ iff there is a computation leading from γ to γ'.

Instance A CFSM $\mathcal{C}$ and a control state s in $\mathcal{C}$.
Question Is there a w such that $\langle s, w\rangle$ is reachable?

The following result is well-known (e.g. [4]).

Theorem 1. *Reach-CFSM is undecidable.*

The idea of the proof is to use one of the channels to simulate the tape of a Turing machine. In fact, this construction implies that the problem is undecidable even for the class of CFSMs with only one channel – a fact we will use in our proof.

3 Probabilistic Lossy Channel Systems

In this section we consider probabilistic lossy channel systems (PLCSs).

Definition 2. *A* lossy channel system (LCS) $\mathcal{L}$ *is of the same form as a CFSM. However, the transition relation* $\longrightarrow$ *is extended such that* $\langle s_1, w\rangle \xrightarrow{t} \langle s_2, w\rangle$ *if* $t = \langle s_1, c!m, s_2\rangle$. *In other words, we allow a message sent to the channel to be lost without ever being appended to the contents of the channel.*

Note that this semantics of loss is slightly different from the one used in [2,6,3], where a message can be lost at any time from the queue. In contrast in the current paper a message can only be lost as it is being placed in the queue. However, the set of reachable states under both semantics is the same [2].

Definition 3. *A* probabilistic Lossy Channel system (PLCS) *is a tuple*

$$\langle S, s_{init}, C, M, \delta, \mathsf{W}, \lambda\rangle$$

where $\langle S, s_{init}, C, M, \delta\rangle$ *is an LCS,* λ *is real number in the interval* $[0, 1]$ *representing the probability of losing messages, and* W *is a* weight function *which assigns to each transition* $t \in \delta$ *a positive real number* $W(t)$.

Given that a PLCS is also an LCS we will lift the transition relation $\longrightarrow$, from LCS, with probabilities, by customizing the weights assigned to the transitions, to obtain the transition relation for PLCS. More precisely, for a global state γ and a transition t, we define $\mathsf{W}'(\gamma)(t)$ to be equal to $\mathsf{W}(t)$ if $t \in en(\gamma)$ and to be equal to 0 otherwise. We define $P(\gamma_1 \xrightarrow{t} \gamma_2)$, where $\gamma_1 = \langle s_1, w_1\rangle$ and $\gamma_2 = \langle s_2, w_2\rangle$, to be equal to

- $(1 - \lambda) \cdot \big(\mathsf{W}'(\gamma_1)(t)/\sum_{t' \in \delta} \mathsf{W}'(\gamma_1)(t')\big)$, if t is of the form $\langle s_1, c!m, s_2\rangle$ and $w_1 \neq w_2$. This corresponds to performing a non-lossy send operation.
- $\lambda \cdot \big(\mathsf{W}'(\gamma_1)(t)/\sum_{t' \in \delta} \mathsf{W}'(\gamma_1)(t')\big)$, if t is of the form $\langle s_1, c!m, s_2\rangle$ and $w_1 = w_2$. This corresponds to a lossy send operation.
- $\big(\mathsf{W}'(\gamma_1)(t)/\sum_{t' \in \delta} \mathsf{W}'(\gamma_1)(t')\big)$, if t is of the form $\langle s_1, c?m, s_2\rangle$ or of the form $\langle s_1, empty, s_2\rangle$.

For a finite computation $\pi = \gamma_1\ t_1\ \gamma_2\ t_2\ \gamma_3\ \cdots\ t_{n-1}\ \gamma_n$, we define

$$\mathcal{B}_\pi = \{\pi' | \pi \text{ is a prefix of } \pi'\}$$

as the basic cylinder set with probability $P(\mathcal{B}_\pi) = \prod_{0<i<n} P(\gamma_i \xrightarrow{t} \gamma_{i+1})$. We will assume the standard measure space based on the Borel field generated from these basic cylinder sets [7], by taking closure under denumerable unions, denumerable intersection and complementation. For global states γ_1 and γ_2, we define $P(\gamma_1, \gamma_2) = P\left(\bigcup_\pi \text{ leads from } \gamma_1 \text{ to } \gamma_2\ \mathcal{B}_\pi\right)$. For a global state γ and a control state s, we define $P(\gamma, s) = P(\bigcup_\pi \text{ leads from } \gamma \text{ to some } \langle s,w\rangle\ \mathcal{B}_\pi)$. We define $P(\gamma) = P(\gamma_0, \gamma)$ and $P(s) = P(\gamma_0, s)$.

The reachability problem for PLCS (Reach-PLCS) is defined as follows:

Instance A PLCS $\mathcal{L}$ and control state s in $\mathcal{L}$.
Question Is $P(s) = 1$?

Note that checking safety properties can be reduced to checking reachability of a bad control state [2]. Furthermore, checking for liveness properties amounts to checking for repeated reachability of a control state [3]. Consequently, our attention to the probabilistic reachability problem is appropriate.

4 Overview of Our Undecidability Proof

In this section, we give an overview of the undecidability proof for Reach-PLCS. The undecidability result is achieved through a reduction from Reach-CFSM.

Suppose that we are given an instance of Reach-CFSM, i.e., a CFSM $\mathcal{C}$ and a control state s_F. Recall that we assume that $\mathcal{C}$ has a single channel, which we will call c_M. We shall construct an equivalent instance of Reach-PLCS, i.e., a PLCS $\mathcal{L}$ which "simulates" $\mathcal{C}$ such that s_F is reached with probability 1 if and only if s_F is reachable in $\mathcal{C}$. The main idea of the construction is to implement two protocols which allow $\mathcal{L}$ to simulate the computations of $\mathcal{C}$. One of the protocols generates multiple copies of messages, while the other one restarts the system.

Generating multiple copies Due to the risk of losing messages, each send operation $c_M!m$ of $\mathcal{L}$ is simulated by a sequence of retransmissions of the message m on channel c_M. This means that each receive operation must be simulated by a "receive loop" where all copies of a message are received.

Retransmitting a message several times decreases the probability that the message is lost in the channel (i.e., that all copies of the message are lost), but the probability is still nonzero. We will therefore construct a scheme in which the number of retransmissions of a message is increased as the execution proceeds. This means that the probability of losing a message will decrease and converge to zero. To be more precise, let $\mathsf{perfectafter}k$ denote the event that among messages queued, in $\mathcal{L}$, to simulate a transmission of a message in $\mathcal{C}$ all message losses are restricted to the first k message retransmissions. We construct a retransmission

scheme in which $P(\mathsf{perfectafter}k)$ converges to 1 as k goes to infinity. Having constructed this scheme, let us consider the probability that in an execution there is a k such that no messages are lost after the first k messages. This event is denoted $\exists k$ $\mathsf{perfectafter}k$ and satisfies

$$\exists k \ \mathsf{perfectafter}k = \bigvee_{k=0}^{\infty} \mathsf{perfectafter}k$$

Hence, for any k we have

$$P(\exists k \ \mathsf{perfectafter}k) \geq P(\mathsf{perfectafter}k)$$

which, since $P(\mathsf{perfectafter}k)$ can be made arbitrary close to 1, implies that

$$P(\exists k \ \mathsf{perfectafter}k) = 1 \tag{1}$$

In other words, we can deduce that with probability 1 there is a k such that after the first k messages no more messages will get lost. Hence the simulation will be "eventually perfect" in the sense that for each transmitted message in the $\mathcal{C}$ at least one of its simulating retransmissions will not be lost in the channel. The ability to construct an eventually perfect simulation of a CFSM by a PLCS is the central concept in the proof. Once the simulation becomes perfect we can let the PLCS simulate the CFSM, implying that the PLCS will reach s_F if and only if the CFSM does so. We observe that the strings generated by simulating different send operations need to be separated by a special symbol # (say) which we does not belong to the original alphabet. Otherwise, given two consecutive occurences of the same messages m in the channel we will not be able to tell whether they correspond to two send operations in $\mathcal{C}$, or to two retransmissions of the same send operation.

Restarting During the early parts of the simulation we have little control over what happens in the PLCS. Therefore we will devise a mechanism whereby the simulation is "restarted" periodically, in order to recover from the possibility of losing all retransmissions of some message. The periods between restarts should be longer and longer, so that eventually they are sufficiently long to simulate the entire computation from initial state to final state in the CFSM (in case the state s_F is reachable). Notice that this implies that $\mathcal{L}$ should never deadlock, since this would mean that the restarting procedure cannot be continued.

To carry out the construction just outlined, the PLCS $\mathcal{L}$ uses, in addition to c_M, two additional lossy channels, called the *retransmission counter* c_T and the *resetting counter* c_S.

The counter c_T The number of retransmissions (copies) of each message will be controlled by the counter c_T. This counter is implemented by a (lossy) channel with two messages, 0 and 1 (say). When the simulation of a send operation is about to start the content of c_T will be of the form 0^k, denoting a counter value k. The operation of sending a message m in $\mathcal{C}$ is replaced in $\mathcal{L}$ by the

following sequence of operations. First, we send k copies of # to c_M. This is done by receiving the 0s in the head of c_T. Each time a 0 is received, we send two 1s to c_T and send a copy of # to c_M. This means that during sending the #s, the content of the counter will be of the form $0^{k_1}1^{k_2}$, where k_1 is the number of times the symbol # still have to be transmitted. This operation will continue until a 1 is received, indicating that all 0s have already been received. At this point the content of c_T will be of the form 1^ℓ, where ℓ is the number of times the message m will be copied to c_M. The copies of m are sent in a similar manner to transmission of #s, with the difference that the system now receives 1s from c_T and send 0s. Observe that the above procedure produces k copies of # followed by ℓ copies of m. We also observe that in the long run (even taking into account the possibility of losing messages), the value of c_T increases, and therefore the number of copies produced will also increase. Receiving a message m is simulated by a loop which first removes all #s in front of the queue and then removes all copies of m.

The counter c_S The mechanism for restarting the simulation must be devised so that the system is restarted infinitely often, with increasing periods between the restarts. This is controlled by the counter c_S. The counter c_S should satisfy three properties

- Its value should be increasing at the successive restarting points.
- Its value should be decreasing between two restarting points.
- the value of c_S should grow more quickly than the value of c_T.

The first two conditions are not contradictory (although they might seem to be). The trick is to store both the current value and the next value of c_S. During the simulation, we continuously decrease the current value and increase the next value of the counter. When the restarting procedure is initiated the contents of c_T is either of the form 0^k or of the form 1^k. The current value k of c_S represents the number of steps for which the system is simulated, before the next restart is performed. Suppose that the content of c_S is of the form 0^k. When simulating a step of $\mathcal{C}$ the current value of c_S is decreased. In a manner similar to c_T, this is done by receiving a 0 and sending two 1s. This means that, during the simulation procedure, the content of c_S will be of the form $0^{k_1}1^{k_2}$, where k_1 is the current value of the counter (the number of steps of $\mathcal{C}$ that still have to be simulated) and k_2 is the next value of the counter. This implies that the current value of the counter will decrease, while the next value will increase during the simulation. The main difficulty in implementing c_S is to guarantee that its value will decrease even if the rest of the system has deadlocked. This can happen if we try to receive a message m from c_M which has been lost. To avoid this, we add to each receive loop an additional loop decreasing the value c_M. This guarantees that the system is reset infinitely often with increasing intervals according to the following

- If the message we attempt to receive is not inside c_M then the loop decreasing c_S will eventually make the current value of c_S equal to 0 and the system is restarted.

– If the message we attempt to receive is available then the loop decreasing c_S will still be running. However, since c_S grows more quickly than c_T, the number of copies of the message (which is bound by the value of c_T) will be all received before the c_S has decreased to 0. Therefore, more steps of the simulation can be performed.

Observe that the contents of c_T is always either of the form $0^{k_1}1^{k_2}$ in which case we say that the system is in mode 0, or of the form $1^{k_1}0^{k_2}$ in which case we say that the system is in mode 1.

5 Implementation of Operations

In this section we show in more detail how to implement $\mathcal{L}$ such that its behaviour satisfies the properties described in Section 4. Each control state s in $\mathcal{C}$ has two copies s_0 and s_1 in $\mathcal{L}$ corresponding to the two modes 0 and 1. There are also two special states $exit_0$ and $exit_1$, from which the restarting procedure starts. Furthermore, there are a number of "intermediate" states which do not correspond to any states in $\mathcal{C}$. Below we describe how we implement the restarting procedure and the operations of sending and receiving messages. We assume mode 0. The behaviour during mode 1 can be derived by interchanging the roles of the 0s and the 1s. We assume that each transition of $\mathcal{L}$ has a positive weight, and that the probability of losing messages is less than 0.5. Furthermore, we will assume that if there is a self-loop containing only a send transition then its probability is also less than 0.5.

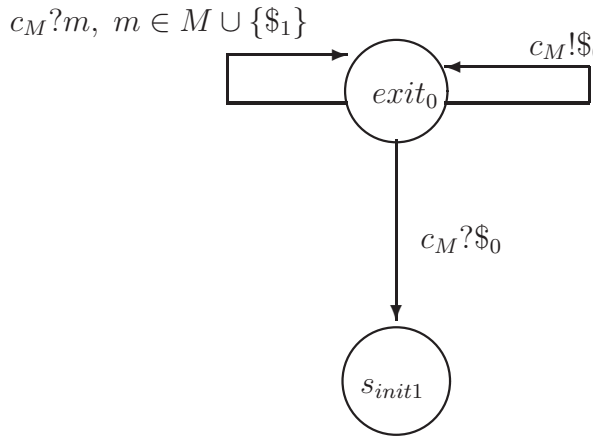
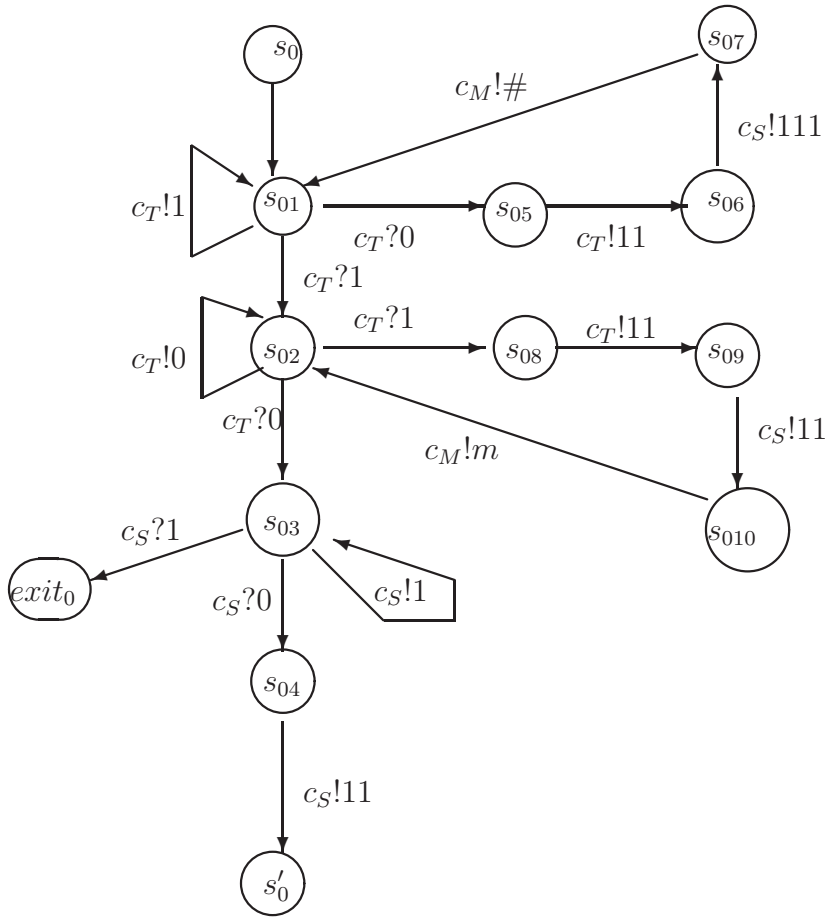


Fig. 1. The restarting operation

Restarting: To implement this operation, We use two new symbols $\$_0$ and $\$_1$ not in Σ as end markers. This operation (Figure 1) resets the content of c_M. It starts from one of the exit states and empties c_M of all messages except $\$_0$ or $\$_1$. Assuming mode 0, we start from $exit_0$ and send $\$_0$ to c_M, and then start receiving all messages until we receive an $\$_0$. In such a case we know that the content of c_M is a string in $\$_0^*$. The reason why we need two symbols (rather than only one), is the fact that some end markers may be left inside the channel from previous time we performed the resetting procedure. By alternating the use of the end markers (using $\$_0$ in mode 0 and $\$_1$ in mode 1), we can distinguish between the current end marker and the previous one.



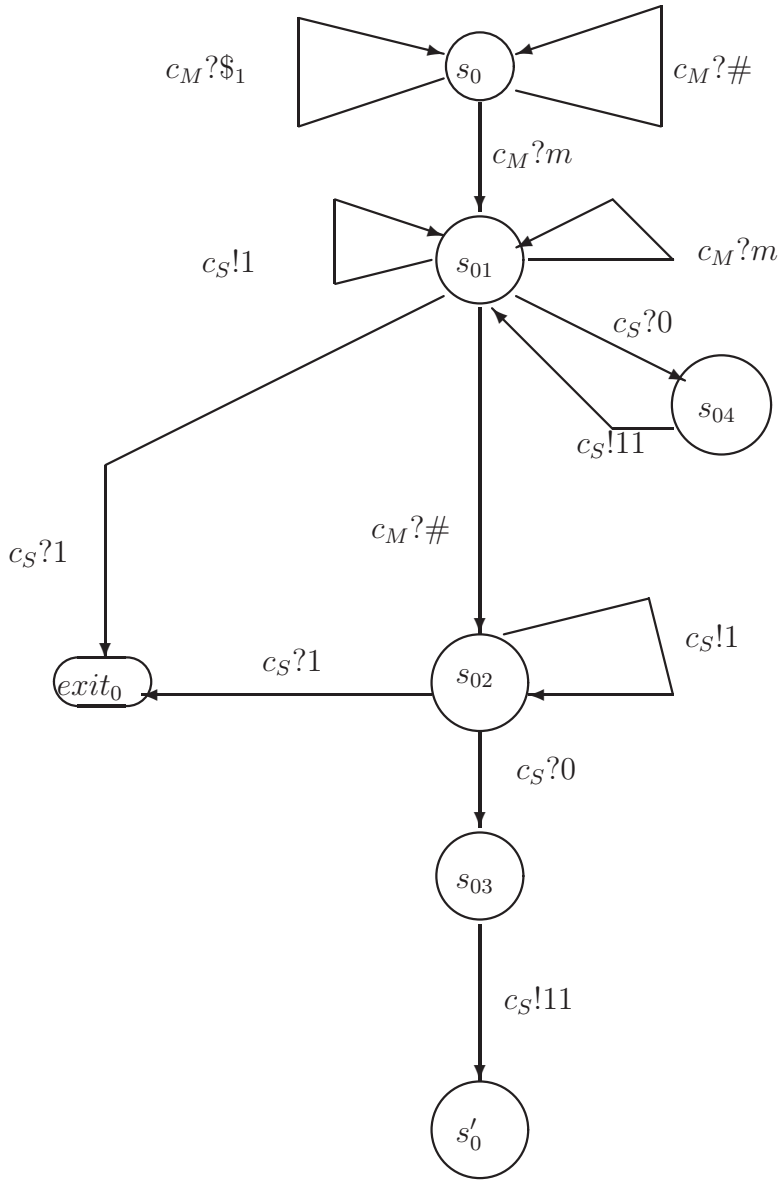
Note: $c_T!11$ should be read as two send command, each of them sending an 1 to c_T.

Fig. 2. Translation of send transition $\langle s, c_M!m, s' \rangle$ in mode 0

Send: Suppose that we have a transition of the form $(s, c_M!m, s')$ in $\mathcal{C}$. This transition is simulated in mode 0 by the following sequence of transitions in $\mathcal{L}$ (Figure 2). First we generate as many copies of # as the value of c_T. This is implemented as follows. We receive a 0 from c_T. To avoid decreasing the value of c_T we replace the 0 by two 1s. Furthermore, to avoid decreasing the ratio between the values of the counters we send three 1s to c_S. Finally, we send a copy of # to c_M. This procedure is repeated until we receive a 1 from c_T indicating that we have consumed all the 0s in the front of c_T and thus produced all the necessary copies of #. In such a case we move to the state s_{02} and start sending copies of m in a similar manner. The aim of the self-loop, as is the case else where, from states s_{02} and s_{03} is to prevent deadlocks when trying to receive from c_T and c_S, respectively. After we have produced the required number of copies of m we move to state s_{03}. Observe that c_T will now contain a string of 0s. At state s_{03}, we reduce the value of c_S. If c_S is empty, i.e., if there is a 1 at the head of c_T, we go to state $exit_0$ to change the mode to 1, and to restart the system. Otherwise, we receive a 0 from c_S replace it by two 1s, and proceed to state s'_0, where we continue with the simulation. The reason we send two 1s is to guarantee that the next value of the counter (represented by the number 1s) will be increasing. We also need the self-loop from state s_{01} to prevent deadlock if all messages inside c_M have been lost.

Receive: A transition $\langle s, c_M?m, s' \rangle$ is simulated as in Figure 3. We have two loops from s_0 which receive # and $\$_1$. This procedure is needed since copies of $\$_1$ may have been left from the restarting procedure (see above), and since the sending procedure produces multiple copies of #. We receive m from c_M and move to s_{01}. Observe that the system cannot deadlock at state s_0, since s_0 has at least one outgoing transition which is not a receive transition (by the assumption mentioned in Section 2). From s_{01} there are two loops. In one of the loops we receive copies of m from c_M, until we receive a # indicating that all copies of m have been consumed. The other loop decreases the value of c_S in a similar manner to that described above. This prevents deadlock in case no copies of # reside at the head of c_M. Notice that this self-loop is activated even if # is available. However, we know that the value of c_S is much larger than the number of copies of m (bounded by the value of c_T), and hence all copies of these messages will be received before c_S has become empty. In the long run, it will always be the case that the system will eventually receive # and move to state s_{02}. In s_{02} we decrease the value of c_S, move to s'_0, and continue with the simulation.

Empty transition: The translation for an empty transitions is similar to the translation of a send transition, except that the only task of the translation is to receive a 0 from c_S (followed by an output of two 1s onto the channel c_S). Clearly, if the head of c_S is a 1 then the execution should be restarted.



Note: $c_S!1^k$ means k sends of 1 on channel c_S.

Fig. 3. Translation of receive transition $\langle s, c_M?m, s' \rangle$ in mode 0

6 Correctness

We now show the correctness of the reduction described in the previous sections. Assume that s_{init0} and s_{init1} are the start states of the two modes. Our first observation is the following: every time the system is at a restart point the probability of reaching the next restart point is 1. Formally,

Lemma 1. *For every* $\gamma_{i0} = (s_{init0}, w_M, w_T, w_S)$ *and* $\gamma_{i1} = (s_{init1}, w_M, w_T, w_S)$ *we have* $P(\gamma_{i0}, s_{init1}) = 1$ *and* $P(\gamma_{i1}, s_{init0}) = 1$.

Proof sketch: By inspection of the translation we can see that any path starting from s_{init0} will reach s_{init1} in at most k steps, where k is encoded in the channel c_S. Similar argument holds for paths starting from s_{init1}. We thus have: $P(s_{init0}) = 1$ and $P(s_{init0}) = 1$. □

We could now extend this lemma to computations. To that end define $visit_\infty$ as follows:

Definition 4. *Given a control state* s *define* $visit_\infty(s)$ *to be the set of computations* π *such that the control state* s *appears infinitely often in* π.

From our previous lemma we now have:

Corollary 1. $P(visit_\infty(s_{init0})) = P(visit_\infty(s_{init1})) = 1$.

We will now concentrate on the behavior of the counters c_T and c_S. Recall from our discussions about c_T (on Page 325) that we wish it grow unboundedly. We will now prove that our implementation indeed does satisfy that requirement. Clearly, from the construction of our PLCS system we see that whenever we remove a 0 from c_T we attempt to place two 1s in c_T. It is possible that a subset of these two messages could be lost (by our semantics). A simple calculation shows us for every 0 received the probability that the size of the chanel will increase by 1 is given by the formula $(1-\lambda)^2$. Given that we are allowed to pick a λ to make our proof work, if we assume that $\lambda < \frac{1}{2}$ then the probability of increasing the size in every cycle of action on c_T is at least $\frac{1}{2}$. Now, consider the subprocess of the entire system concentrating on the counter c_T (this amounts to taking a projection of the computation sequences onto the actions on the counter c_T); we are justified in taking such a projection because the actions on c_T are independent of actions on other queues (note that the system never deadlocks). The advantage of this view is that it can be viewed as a random walk in a single dimension. By classic results [7] on random walk we can infer that when the probability of increasing the size is greater than $\frac{1}{2}$ the probability of the system drifting away with larger sizes is 1. This observation provides us with a justification for the following:

Lemma 2. *For every natural number* n *we have*

$$P(\{\pi | \exists \ell : \pi(\ell) = \langle s_{init0}, w_M, w_S, w_T \rangle \wedge w_T = 0^k \wedge k > n\}) = 1$$

.

We now provide justification for the requirement on counter c_S that we expressed in Page 326, viz., that it grows faster than the counter c_T. By inspection of the construction we see that we always add more to the counter c_S than we do to c_T. To keep the fraction $\frac{w_S}{w_T}$ bounded we will have to lose increasing number of messages, whose probability, of course, tends towards zero. Thus, we have:

Lemma 3. *For every natural number n we have*

$$P(\{\pi | \exists \ell : \pi(\ell) = \langle s_{init0}, w_M, w_S, w_T \rangle \wedge \frac{w_S}{w_T} > n\}) = 1$$

.

□

Note that both of the lemmata above also hold for s_{init1}. It follows from Corollary 1 that if a control state s is reachable in a CFSM then it should be reachable with probability 1 in the corresponding PLCS. To see this let s be reachable control state in the CFSM. Then the set of execution sequences that neither visit s_0 nor s_1 can be expressed as an intersection of an infinite sequence of sets, B_i, where the i^{th} set does not take the first i opportunities to visit s_0 and s_1. Given that the probability of the intersection of the infinite sequence of sets, B_i, is the limit of an ever decreasing sequence of probabilities $P(B_i)$, we can infer that both s_0 and s_1 can only be avoided with probability 0. Formally, we have:

Lemma 4. *If a control state s is reachable in a CFSM then $P(\gamma_0, s_0) = 1$ and $P(\gamma_0, s_1) = 1$.*

We will now consider the other direction of our simulation. Let s be a control state which is unreachable in the $\mathcal{C}$. Now it is possible, due to the lossy nature of the buffer, that the control state s is visited. We will show that even though s might be visited in the $\mathcal{L}$ it can not be visited with probability one. To that end consider $\Pi(k) = \{\pi | \pi \mathsf{perfectafter} k\}$. Clearly, we know that these sets are not empty. We now claim that the there exists a k and a non-empty $\Pi'(k) \subseteq \Pi(k)$ such that the sequences in $\Pi'(k)$ do not visit the control state s and that $P(\Pi'(k)) > 0$. If there were no such k we would be contradicting Equation 1, which holds because the counters c_S and c_T grow unboundedly (from Lemma 2 and Lemma 3). Given that there is a set of sequences with non-zero measure that do not visit state s_0 or s_1 we can infer the following:

Lemma 5. *If a control state s is unreachable in a CFSM then in its corresponding PLCS the probability of visiting s_0 or s_1 is less than 1.*

This leads to the main result of the paper:

Theorem 2. *The problem Reach-PLCS is undecidable.*

7 Conclusion

In this paper we have shown that reachability with probability 1 is undecidable for probabilistic lossy channel systems. This in turn implies that model checking

properties of lossy channel systems with probability 1 is also undecidable. The proof uses a model of loss where the probability to lose a message is bounded both from above and from below. It would be interesting to consider what the impact of removing the upper and/or the lower bound of losing a particular message would be on the verification problems.

Acknowledgments

We are grateful to Marta Kwiatkowska for prompting this work by an invitation to PROBMIV 1999.

References

1. Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1):71–90, 1996. A Preliminary Version appeared in Proc. ICALP '94, LNCS 820. 321
2. Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996. A Preliminary Version appeared in Proc. 8^{st} IEEE Int. Symp. on Logic in Computer Science. 320, 321, 323, 324
3. C. Baier and B. Engelen. Establishing qualitative properties for probabilistic lossy channel systems. In Katoen, editor, *ARTS'99, Formal Methods for Real-Time and Probabilistic Systems, 5th Int. AMAST Workshop*, volume 1601 of *Lecture Notes in Computer Science*, pages 34–52. Springer Verlag, 1999. 321, 323, 324
4. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983. 320, 323
5. Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On randomizatioon in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, March 1994. 321
6. Purush Iyer and Murali Narasimha. Probabilistic lossy channel systems. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 667–681. Springer-Verlag, 1997. 321, 323
7. J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov Chains*. Van Nostrand, New Jersey, 1966. 324, 331
8. M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976. This classic paper on probabilistic algorithms features algorithms for primality testing and nearest neighbors. 321

Weak Bisimulation for Probabilistic Systems*

Anna Philippou[1], Insup Lee[2], and Oleg Sokolsky[2]

[1] Department of Computer Science, University of Cyprus, Cyprus
annap@cs.ucy.ac.cy
[2] Department of Computer and Information Science, University of Pennsylvania, USA
lee@central.cis.upenn.edu
sokolsky@saul.cis.upenn.edu

Abstract. In this paper, we introduce weak bisimulation in the framework of Labeled Concurrent Markov Chains, that is, probabilistic transition systems which exhibit both probabilistic and nondeterministic behavior. By resolving the nondeterminism present, these models can be decomposed into a possibly infinite number of computation trees. We show that in order to compute weak bisimulation it is sufficient to restrict attention to only a finite number of these computations. Finally, we present an algorithm for deciding weak bisimulation which has polynomial-time complexity in the number of states of the transition system.

1 Introduction

In recent years, the need for reasoning about probabilistic behavior, as exhibited for instance in randomized, distributed and fault-tolerant systems, has triggered much interest in the area of formal methods for the specification and analysis of probabilistic systems [6,11,13,14,15,27,28,30]. The general approach taken has been to extend existing models and techniques which have proved successful in the nonprobabilistic setting with probability.

Thus, much work in the area of formal models for probabilistic systems has been based on labeled transition systems [23]. In order to extend labeled transition systems to the probabilistic setting, various mechanisms for capturing probabilistic behavior have been proposed and investigated. On one end of the spectrum, several approaches have replaced nondeterministic branching in labeled transition systems with probabilistic branching [13] by assigning probabilities to each transition, while others explored the possibility of integrating nondeterministic and probabilistic behavior [30,21,13,14,28]. For example, in the reactive model of [13] as well as in the simple probabilistic automata of [28], probability distributions are dependent on the occurrence of actions, whereas in the stratified model, levelwise probabilistic branching is also possible [13]. A more general model for probabilistic computation is captured in the probabilistic transition

* This research was supported in part by NSF CCR-9619910, NSF CISE-9703220, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, and ONR N00014-97-1-0505 (MURI).

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 334–349, 2000.

systems of [30] and the probabilistic automata of [28], which extend the stratified model with nondeterminism among actions.

Verification techniques for these models have been inspired by successful approaches in the nonprobabilistic case. On one hand, temporal logics have been enriched with probability and on the other hand, probabilistic notions of equivalence and preorder relations have been explored. Among these, bisimulations [21], simulations [17,29] and testing preorders [8,10,16], have been defined and algorithms given for their automatic verification [2,9]. The majority of this work has focused on fully probabilistic systems, that is, systems where only probabilistic branching is involved.

In the nonprobabilistic setting, weak bisimulations have proved fundamental for the compositional verification of systems where abstraction from internal computation is essential. However, such notions have been rare in the setting of probabilistic systems and, as noted, although desirable their formalization has been problematic [14,18,3]. One paper that addresses this issue is [29], where notions of weak and branching bisimulations are introduced for a certain class of probabilistic transition systems defined by *simple probabilistic automata*. This model captures nondeterministic and probabilistic behavior by allowing from each state the nondeterministic choice of a number of probability distributions, each of which involves probabilistic transitions associated with a distinct action. The definition presented replaces the weak transition in the weak bisimulation definition of Milner [23], by assigning a (possibly infinite) set of distributions to each state, representing the (non-deterministic) alternatives of probabilistic distributions for states that are reachable by weak transitions. However, no method for computing the notion is considered in [29].

More recently, [3] has introduced a notion of weak bisimulation for fully probabilistic systems and presented a polynomial-time algorithm for deciding it. In this definition, weak transitions are replaced with the probability of making a transition to reach a certain state and the requirement of weakly bisimilar states is that the probability of a step by each process can be matched by the other. Another algorithmic approach for deciding weak equivalences for the model of [29] is proposed in [5]. The equivalence hereby obtained lies between strong and weak bisimulation.

In this paper, we propose a notion of weak bisimulation for probabilistic systems that allows for both nondeterministic and probabilistic branching. Such systems arise as formal models of randomized distributed systems, as well as real-life reactive systems that exhibit uncertainty. While probabilistic choice in these systems becomes relevant due to faults or random assignments, nondeterministic branching is also present due to the asynchronicity of a system's subprocesses or external intervention, as for instance an action taken by the environment.

Our definition of weak bisimulation in this model extends the definition of [3] by treating nondeterministic in addition to probabilistic behavior. It achieves this by incorporating the notion of a *scheduler*, an entity that resolves nondeterminism in a system by choosing the next step to take place, out of a set of nondeterministic alternatives. Indeed, due to the presence of nondeterminism it

is not possible in general to determine the probability with which a weak transition may take place. Instead, we associate such a probability with each of the possible schedulers and in order to establish weak bisimulation we compare the set of possible probabilities for each of two states. Our first main result is that according to our definition, weak bisimulation can be characterized in terms of maximum probabilities of transitions over all schedulers. We then turn to tackle the problem of computing such probability bounds. Although the set of schedulers for a system is in general infinite, our second main result shows that in order to compute maximum probabilities it is sufficient to consider only a finite number of them. In particular, we introduce the notion of *determinate* schedulers and we isolate a finite set of schedulers in which maximal probabilities arise. On the basis of the above, we present an algorithm for deciding weak bisimulation equivalence classes, in time polynomial in the number of states of the underlying probabilistic system. Thus the main contribution of this paper is the definition of weak bisimulation in a general framework of probabilistic systems and the corresponding algorithm for computing weak bisimulation equivalence classes.

The remainder of the paper is organized as follows: the following section contains an account of the Labeled Concurrent Markov Chain model and some background material, section 3 introduces and studies the notion of weak bisimulation, section 4 presents determinate schedulers and their relevance to weak bisimulation, while section 5 describes an algorithm for deciding weak bisimulation classes. We conclude with a comparison of our proposal with that of [29] and a discussion of further work. Due to the limitation of space, proofs of results are only briefly sketched, the complete proofs can be found in [25].

2 The Model

In this section we introduce Labeled Concurrent Markov Chains and some background definitions and notations we will be using.

Definition 1. A *Labeled Concurrent Markov Chain* (LCMC) is a tuple $\langle S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0\rangle$, where S_n is the set of nondeterministic states, S_p is the set of probabilistic states, $Act = L \cup \{\tau\}$ is the set of labels, (where τ is the internal action), $\longrightarrow_n \subset S_n \times Act \times (S_n \cup S_p)$ is the nondeterministic transition relation, $\longrightarrow_p \subset S_p \times (0,1] \times S_n$ is the probabilistic transition relation, satisfying $\Sigma_{(s,\pi,t)\in\longrightarrow_p} \pi = 1$ for all $s \in S_p$, and $s_0 \in S_n \cup S_p$ is the initial state. □

Thus the set of states of an LCMC is partitioned into two sets, S_n and S_p. States in S_n are capable of performing nondeterministic transitions while states in S_p may perform probabilistic transitions. We assume both of these sets to be finite. Note that the nonprobabilistic model is derivable from the LCMC model by setting $S_p = \emptyset$. In what follows we will write S for $S_n \cup S_p$ and we will let s, s' range over S, α, β over Act and ℓ over $Act \cup (0,1]$. In addition, when it is clear within a context, we will refer to a LCMC $\langle S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0\rangle$ by s_0.

Computations of LCMC's arise by resolving the nondeterministic and probabilistic choices:

Definition 2. A *computation* in $\Xi = \langle S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0\rangle$ is either a finite sequence $s_0\,\ell_1\,s_1\ldots\ell_k\,s_k$, where s_k has no transitions, or an infinite sequence $s_0\,\ell_1\,s_1\ldots\ell_k\,s_k\ldots$, such that $(s_i, \ell_{i+1}, s_{i+1}) \in \longrightarrow_p \cup \longrightarrow_n$, for all $0 \leq i$.

We denote by $\mathsf{Comp}(\Xi)$ the set of all computations of Ξ and by $\mathsf{Comp}_{fin}(\Xi)$ the set of all partial computations of Ξ, i.e. $\mathsf{Comp}_{fin}(\Xi) = \{s_0\ell_1\ldots\ell_k s_k \mid \exists c \in \mathsf{Comp}(\Xi)\cdot\ c = s_0\ell_1\ldots\ell_k s_k\ldots\}$. Given $c = s_0\ell_1\ldots\ell_k s_k \in \mathsf{Comp}_{fin}(\Xi)$, we define $\mathsf{trace}\,c = \ell_1\ldots\ell_k\restriction L$, $\mathsf{inter}\,c = \{s_0,\ldots,s_{k-1}\}$, $\mathsf{first}\,c = s_0$ and $\mathsf{last}\,c = s_k$.

To define probability measures on computations, it is necessary to resolve the nondeterminism present. To achieve this, the notion of a scheduler has been employed [30,14,29]. A scheduler is an entity that given a partial computation (ending in a nondeterministic state) chooses the next transition to be scheduled:

Definition 3. A *scheduler* of a LCMC Ξ is a function $\sigma : \mathsf{Comp}_{fin}(\Xi) \mapsto (\longrightarrow_n \cup\bot)$, such that, if $\sigma(c) = tr \in \longrightarrow_n$ then $tr = (\mathsf{last}\,c, \alpha, s)$ for some α and s. □

Here we use $\sigma(c) = \bot$ to express that a scheduler may schedule nothing at some point during computation. In the rest of the paper we will use $\mathsf{Sched}(\Xi)$ to denote the set of schedulers of Ξ, and we will let σ range over all schedulers. For a LCMC Ξ and a scheduler $\sigma \in \mathsf{Sched}(\Xi)$ we define the set of *scheduled computations* $\mathsf{Scomp}(\Xi,\sigma) \subseteq \mathsf{Comp}(\Xi)$, to be the finite computations $c = s_0\,\ell_1\ldots\ell_k\,s_k$ where for all $i < k$, $s_i \in \mathsf{S}_n$ $\sigma(s_0\,\ell_1\ldots\ell_i s_i) = (s_i, \ell_{i+1}, s_{i+1})$, and $\sigma(s_0\,\ell_1\ldots\ell_k\,s_k) = \bot$, and the infinite computations $c = s_0\,\ell_1\ldots\ell_k\,s_k\ldots$ where for all i, $s_i \in \mathsf{S}_n$, $\sigma(s_0\,\ell_1\ldots\ell_i s_i) = (s_i, \ell_{i+1}, s_{i+1})$. Each scheduler σ induces a probability space on $\mathsf{Scomp}(\Xi,\sigma)$ in the usual way [29].

We conclude with the definition of strong bisimulation for the model. First we have a definition.

Definition 4. Given $s, s' \in \mathsf{S}$, and $\mathcal{M} \subseteq \mathsf{S}$, we define

1. $$\mathsf{pr}(s,s') = \begin{cases} \pi, & \text{if } s \xrightarrow{\pi}_p s' \\ 1, & \text{if } s = s', s \in S_n \\ 0, & \text{otherwise} \end{cases}$$
2. $\mu(s,\mathcal{M}) = \Sigma_{s'\in\mathcal{M}}\mathsf{pr}(s,s')$. □

Thus, $\mathsf{pr}(s,s')$ denotes the probability that s may perform at most one probabilistic transition to become s', and $\mu(s,\mathcal{M})$ denotes the cumulative probability that s may perform a probabilistic action to a state in $\mathcal{M}$.

Definition 5. An equivalence relation $\mathcal{R} \subseteq \mathsf{S}\times\mathsf{S}$ is a *strong bisimulation* if, whenever $s\mathcal{R}t$

1. for all $\alpha \in Act$, if $s,t \in S_n$ and $s \xrightarrow{\alpha}_n s'$ then $t \xrightarrow{\alpha}_n t'$ and $s'\mathcal{R}t'$;
2. for all $\mathcal{M} \in \mathsf{S}/\mathcal{R}$, $\mu(s,\mathcal{M}) = \mu(t,\mathcal{M})$.

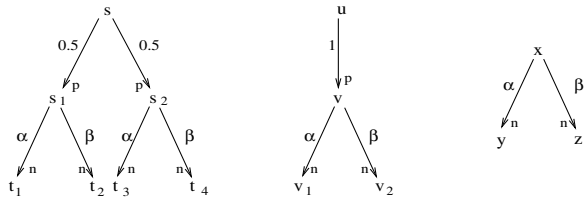


Fig. 1. $s \sim u \sim x$

Two states s and t are *strong bisimulation equivalent*, written $s \sim t$, if there exists a strong bisimulation $\mathcal{R}$ such that $s\mathcal{R}t$. □

An example of strong bisimulation equivalent systems is shown in Figure 1.

The above definition is almost identical to the one proposed in [14], where an *alternating* model is considered, However, with a slight reformulation of the definition of $\mathsf{pr}(s, s')$, Definition 5 allows for pairs of probabilistic and nondeterministic systems, such as (s, x), to be considered as bisimulation equivalent.

3 Weak Bisimulation

In this section we define weak bisimulation for probabilistic transition systems. Weak bisimulation was introduced in the context of nonprobabilistic transition systems in [23]. It abstracts away from internal computation by focusing on *weak* transitions, that is transitions of the form $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ (where $\Longrightarrow$ is the transitive, reflexive closure of $\xrightarrow{\tau}$) and requires that weakly bisimilar systems can match each other's observable behavior.

However, in the probabilistic setting, while considering a transition with observable content $\alpha \in Act$ ($\mathsf{trace}\, c = \alpha$), it is necessary to take account of the probability of the transition taking place, and to ensure that weakly bisimilar systems may not only match one another's transitions but also perform the transitions with matching probabilities. To achieve this, given a state s, an action α, and an equivalence class of the weak bisimulation relation, $\mathcal{M}$, we are interested in computing the probability of s reaching a state in $\mathcal{M}$ via a transition with observable content α. This probability depends on how the nondeterminism of s is resolved. So, given a LCMC Ξ, $\Phi \subset Act^*$, $\mathcal{M} \subseteq \mathsf{S}$ and $\sigma \in \mathsf{Sched}(\Xi)$ we define

$$\mathsf{Paths}(\Xi, \Phi, \mathcal{M}, \sigma) \;=\; \{c \in \mathsf{Scomp}(\Xi, \sigma) \mid \mathsf{last}\, c \in \mathcal{M},\ \sigma(c) = \bot,\ \mathsf{trace}\, c \in \Phi\}.$$

Thus, $\mathsf{Paths}(\Xi, \Phi, \mathcal{M}, \sigma)$ denotes the set of computations of Ξ, scheduled by σ, leading to a state in $\mathcal{M}$ via a sequence of actions in Φ. In what follows, we use ε to denote the empty word. We also often abbreviate the singleton set $\{x\}$ by x.

The probability $\mathsf{Pr}(s, \Phi, \mathcal{M}, \sigma) \stackrel{\text{def}}{=} \mathcal{P}(\mathsf{Paths}(\Xi, \Phi, \mathcal{M}, \sigma))$, s is the initial state of Ξ, is given by the smallest solution to $X(s, \Phi, \mathcal{M}, \sigma, s)$ defined by the following

set of equations:

$$X(s,\Phi,\mathcal{M},\sigma,c) = \begin{cases} 1, & \text{if } \varepsilon \in \Phi, s \in \mathcal{M},\ \sigma(c) = \bot \\ 0, & \text{if } (\varepsilon \notin \Phi, s \notin \mathcal{M},\ \sigma(c) = \bot) \text{ or } \Phi = \emptyset \\ \Sigma_t\ \mathsf{pr}(s,t) \cdot X(t,\Phi,\mathcal{M},\sigma,c\,\mathsf{pr}(s,t)\,t), & \text{if } s \in S_p \\ X(t,\Phi-\alpha,\mathcal{M},\sigma,c\,\alpha\,t), & \text{if } s \in S_n, \sigma(c) = (s,\alpha,t) \end{cases}$$

where $\Phi - \alpha = \{\phi \mid \alpha\phi \in \Phi\}$. Note that the last argument of $X(s,\Phi,\mathcal{M},\sigma,c)$, c, records the history of reaching s. This is needed for performing subsequent scheduling of s under scheduler σ. Moreover, we use regular expressions (such as $\tau^*\alpha\tau^*\beta\tau^*$) to represent sets of traces. So, for example, $\mathsf{Pr}(s,\tau^*\alpha\tau^*,\mathcal{M},\sigma)$ denotes the probability to reach some state in $\mathcal{M}$ from state s, at the endpoints of scheduler σ, by performing a weak transition with observable content α.

The definition of weak bisimulation follows. As usual we write $\widehat{\alpha}$ for α if $\alpha \in L$ and ε otherwise. Furthermore, given a relation $\mathcal{R}$, we write $\mu_{\mathcal{R}}(s,\mathcal{M})$ for the probability of reaching $\mathcal{M} \subseteq \mathsf{S}$ from state s weighted by the probability of exiting the equivalence class $[s]_{\mathcal{R}}$:

$$\mu_{\mathcal{R}}(s,\mathcal{M}) = \begin{cases} \mu(s,\mathcal{M})/(1-\mu(s,[s]_{\mathcal{R}})), & \text{if } \mu(s,[s]_{\mathcal{R}}) \neq 1, \\ \mu(s,\mathcal{M}), & \text{otherwise} \end{cases}$$

Definition 6. An equivalence relation $\mathcal{R} \subseteq \mathsf{S} \times \mathsf{S}$ is a *weak bisimulation* if whenever $s\mathcal{R}t$,

1. for all $\alpha \in Act$, if $s,t \in S_n$ and $s \xrightarrow{\alpha}_n s'$, then there exists $\sigma \in \mathsf{Sched}(t)$ such that $\mathsf{Pr}(t,\tau^*\widehat{\alpha}\tau^*,[s']_{\mathcal{R}},\sigma) = 1$;
2. there exists $\sigma \in \mathsf{Sched}(t)$ such that for all $\mathcal{M} \in \mathsf{S}/\mathcal{R} - [s]_{\mathcal{R}}$, $\mu_{\mathcal{R}}(s,\mathcal{M}) = \mathsf{Pr}(t,\tau^*,\mathcal{M},\sigma)$.

We say that s and t are weakly bisimilar, written $s \approx t$, if $(s,t) \in \mathcal{R}$ for some weak bisimulation $\mathcal{R}$. □

Thus the definition specifies how deterministic and probabilistic transitions are matched by weakly bisimilar states. Nondeterministic behavior is matched as follows: if one of the system can engage in a transition involving an action there exists a scheduler of the other which weakly performs the same transition with probability one. On the other hand, if $s \approx t$ then, given a probabilistic branching of s, that is a set of transitions $s \xrightarrow{\pi_i}_p s_i$, there exists a *single* scheduler of t that weakly matches the branching, in the sense that the weighted probabilities of reaching any equivalence class $\mathcal{M}$ are the same for both systems. The purpose for considering weighted probabilities is highlighted in the example of Figure 2(a).

We observe that states s and t can both engage in actions α and β equally likely. Furthermore, although, $\mathsf{pr}(t,t_2) = \mathsf{pr}(t,t_3) = 0.2$, t may also probabilistically reach t_1 where clearly $t \approx t_1$, in this way contributing to the probability of t eventually performing either α and β. Thus, t can weakly perform both α and β with probability 0.5 which suggests s and t should be considered weakly bisimilar. To allow this, on matching probabilistic transitions of weakly bisimilar states, we

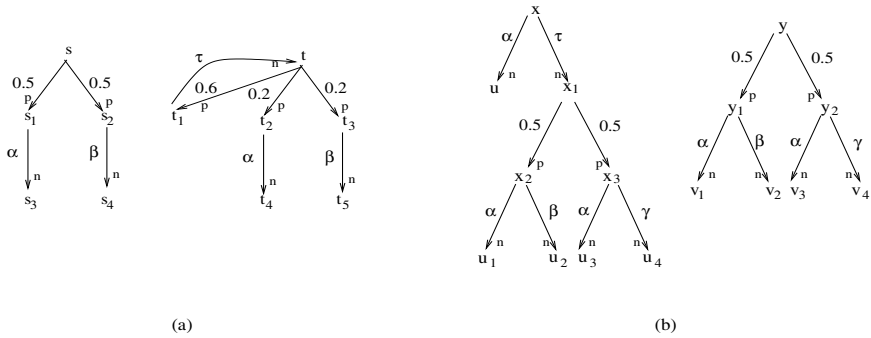


Fig. 2. (a)$s \approx t$ and (b) $x \approx y$

consider the probability of reaching an equivalence class weighted by the probability of exiting the equivalence class of the initial state. Thus, the weak bisimulation that connects s and t is $\mathcal{R} = \{\{s, t, t_1\}, \{s_1, t_2\}, \{s_2, t_3\}, \{s_3, s_4, t_4, t_5\}\}$. To establish satisfaction of the relation for the pair (s, t), we observe that clause 2 of Definition 6 is satisfied as $\mu_{\mathcal{R}}(s, [s_1]_{\mathcal{R}}) = 0.5$ and since $\mu(t, [t]_{\mathcal{R}}) = 0.6$, we have that $\mu_{\mathcal{R}}(t, [s_1]_{\mathcal{R}}) = \mu_{\mathcal{R}}(t, [t_2]_{\mathcal{R}}) = \frac{0.2}{1-0.6} = 0.5$ as required. Similarly it can be shown that $\mu_{\mathcal{R}}(s, [s_2]_{\mathcal{R}}) = \mu_{\mathcal{R}}(t, [s_2]_{\mathcal{R}}) = 0.5$.

Another example of weakly bisimilar systems is exhibited in Figure 2(b). We have that states x and y are related by the weak bisimulation $\mathcal{R} = \{\{x, y, x_1\}, \{x_2, y_1\}, \{x_3, y_2\}, \{u, u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4\}\}$. We point out that while $x \xrightarrow{\alpha}_n u$, there is $\sigma \in \mathsf{Sched}(y)$ such that $\mathsf{Pr}(y, \tau^*\alpha\tau^*, [u]_{\mathcal{R}}, \sigma) = 1$, as required.

Weak bisimilation satisfies the following properties:

Lemma 1. $\approx$ is the largest weak bisimulation and $\sim \subset \approx$. □

Given $\alpha \in Act$ and $\mathcal{M} \subseteq \mathsf{S}$, we introduce the following notation for the largest probabilities of reaching $\mathcal{M}$ via a path with observable content α, over all schedulers: $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) \stackrel{\text{def}}{=} \max_{\sigma \in \mathsf{Sched}(s)} \mathsf{Pr}(s, \tau^*\widehat{\alpha}\tau^*, \mathcal{M}, \sigma)$. Our first important result states that weak bisimulation preserves maximum probabilities:

Theorem 1. If $s \approx t$ then, for all $\alpha \in Act$ and $\mathcal{M} \in \mathsf{S}/\approx$, $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) = \mathsf{Pr}_{max}(t, \alpha, \mathcal{M})$. □

Proof: The proof involves showing that if $s \approx t$, for any $\sigma \in \mathsf{Sched}(s)$, $\alpha \in Act$ and $\mathcal{M} \in \mathsf{S}/\mathcal{R}$ there exists $\sigma' \in \mathsf{Sched}(t)$ such that $\mathsf{Pr}(s, \tau^*\alpha\tau^*, \mathcal{M}, \sigma) \leq \mathsf{Pr}(t, \tau^*\widehat{\alpha}\tau^*, \mathcal{M}, \sigma')$. To do this we must match every move made by s via σ, by a scheduler of t, using the weak bisimulation definition, concatenating schedulers on the way. It then remains to show that the sets of equations that define the two probabilities have the same least solutions. □

We continue with a result which is central to the understanding of the interplay between probabilistic behavior and the definition of weak bisimulation.

Suppose $s \xrightarrow{\pi_i}_p s_i$. The result states that on matching such a transition in t, $t \approx s$, a scheduler only passes via states that are weakly bisimilar to s.

Lemma 2. Suppose $s \approx t$. Then, if $\sigma \in \mathsf{Sched}(t)$ is such that for all $\mathcal{M} \in \mathsf{S}/\approx$, $\mu_\approx(s, \mathcal{M}) = \mathsf{Pr}(t, \tau^*, \mathcal{M}, \sigma)$, then, for all partial computations $c \in \mathsf{Scomp}_{fin}(t, \sigma)$, if $\sigma(c) \neq \bot$, and $t' = \mathsf{last}\, c$, either $t' \approx t$ or $\mathsf{Pr}(t', \tau^*, [t']_\approx, \sigma) = 1$. □

Proof: Suppose $s \approx t$ and pick $\sigma \in \mathsf{Sched}(t)$ such that for all $\mathcal{M} \in \mathsf{S}/\approx$, $\mu_\approx(s, \mathcal{M}) = \mathsf{Pr}(t, \tau^*, \mathcal{M}, \sigma)$. We assume for the sake of contradiction that there exists computation $c \in \mathsf{Scomp}(t, \sigma)$, such that $\sigma(c) \neq \bot$, $t' = \mathsf{last}\, c$ with $t' \not\approx t$ and $\mathsf{Pr}(t', \tau^*, [t']_\approx, \sigma) \neq 1$. (Note that $\mathsf{Pr}(t', \tau^*, [t']_\approx, \sigma) = 1$ implies that $\mu_\approx(s, [t']_\approx) > 0$.) Computation of the probabilities $\mathsf{Pr}_{max}(s, \tau, [t']_\approx)$, $\mathsf{Pr}_{max}(t, \tau, [t']_\approx)$ results in violation of Theorem 1 which completes the proof. □

This result implies that on matching a probabilistic branching of weakly bisimilar states the branching structure of the states is preserved. It comes in agreement with the senario of [3] where it is shown that for fully probabilistic systems weak and branching bisimulations coincide. Of course this does not hold in our model due to the presence of nondeterminism.

Let $\mathsf{Sched}'(t, M)$ be the subset of schedulers of t containing all schedulers that schedule within the set of states M and consider the case $M = [t]_\approx$. We may prove by structural induction of t that $\mathsf{Sched}'(t, [t]_\approx) = \emptyset$, and that each $\sigma_t \in \mathsf{Sched}'(t, [t]_\approx)$ satisfies the requirement of Lemma 2: If $s \approx t$ and $\sigma_t \in \mathsf{Sched}'(t, [t]_\approx)$, then for all $\mathcal{M} \in \mathsf{S}/\approx$, $\mu_\approx(s, \mathcal{M}) = \mathsf{Pr}(t, \tau^*, \mathcal{M}, \sigma_t)$. Thus letting $\mathsf{Pr}'_{max}(s, \alpha, \mathcal{M}, M) \stackrel{\text{def}}{=} \max_{\sigma \in \mathsf{Sched}'(s,M)} \mathsf{Pr}(s, \tau^* \widehat{\alpha} \tau^*, \mathcal{M}, \sigma)$, we have that for all $\mathcal{M} \in \mathsf{S}/\approx$, $Pr'_{max}(t, \tau, \mathcal{M}, [t]_\approx) = \mu_\approx(s, \mathcal{M})$.

As a consequence, we have an alternative definition of weak bisimulation in terms of maximum probabilities.

Theorem 2. An equivalence relation $\mathcal{R} \subseteq \mathsf{S} \times \mathsf{S}$ is a *weak bisimulation iff* whenever $s\mathcal{R}t$, then

1. if $s, t \in S_n$, $\alpha \in Act$ and $\mathcal{M} \in \mathsf{S}/\mathcal{R}$, then $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) = \mathsf{Pr}_{max}(t, \alpha, \mathcal{M})$;
2. for all $\mathcal{M} \in \mathsf{S}/\mathcal{R} - [s]_\mathcal{R}$, $\mu_\mathcal{R}(s, \mathcal{M}) = \mathsf{Pr}'_{max}(t, \tau, \mathcal{M}, [t]_\mathcal{R})$. □

Proof: Let $\mathcal{R}$ be a weak bisimulation and suppose $s\mathcal{R}t$. Then, by Theorem 1, $\mathcal{R}$ satisfies condition 1 above. In addition, by the observation above, $\mu_\mathcal{R}(s, \mathcal{M}) = \mathsf{Pr}'_{max}(t, \tau^*, \mathcal{M}, [t]_\mathcal{R})$, thus condition 2 is also satisfied.

To prove the converse, suppose $\mathcal{R}$ satisfies the conditions of the theorem. By condition 1, $\mathcal{R}$ satisfies Definition 6(1). To establish the second condition it is sufficient to note that whenever $t \xrightarrow{\tau}_n t_1$, $t \xrightarrow{\tau}_n t_2$ with $t\mathcal{R}t_1\mathcal{R}t_2$ and $(s, t) \in \mathcal{R}$, for all $\mathcal{M} \in \mathsf{S}/\mathcal{R}$, $\mu_\mathcal{R}(s, \mathcal{M}) = \mathsf{Pr}'_{max}(t_1, \tau^*, \mathcal{M}, [t_1]_\approx)$, and $\mu_\mathcal{R}(s, \mathcal{M}) = \mathsf{Pr}'_{max}(t_2, \tau^*, \mathcal{M}, [t_2]_\approx)$. This implies that for any $\sigma \in \mathsf{Sched}'(t, [t]_\mathcal{R})$, $\mu_\mathcal{R}(s, \mathcal{M}) = \mathsf{Pr}(t, \tau^*, \mathcal{M}, \sigma)$, for all $\mathcal{M} \in \mathsf{S}/\mathcal{R}$. Thus, Definition 6(2) holds. □

4 Determinate Schedulers

In this section we turn to the issue of deciding weak bisimulation for probabilistic systems. According to Theorem 2, establishing weak bisimilarity of two

systems amounts to computing certain maximum probabilities. These probabilities are quantified over the set of all schedulers, and as noted earlier, such sets are in general infinite. The question then arises whether it is possible to compute maximum probabilities by looking only at a finite subset of schedulers.

So let $s \in \mathsf{S}$, $\alpha \in Act$ and $\mathcal{M} \in \mathsf{S}/\approx$ and consider $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M})$. First we point out that the only schedulers relevant to computing this probability are such that $\sigma(c) = \perp$ unless $\mathsf{trace}\, c \in \{\varepsilon, \alpha\}$. Given a set of schedulers D, let D^α be the subset of D which only contains such schedulers. We may see that in general D^α is an infinite set. For example, consider agent s in Figure 3. The family of schedulers $\{\sigma_i\}_{i\geq 0}$ defined below is such that $\sigma_k \in \mathsf{Sched}(s)^\alpha$.

$$\sigma_k(s\, 0.5\, (u\, \tau s\, 0.5)^i\, u) = (u, \tau, s), \text{ if } i < k$$
$$\sigma_k(s\, 0.5\, (u\, \tau s\, 0.5)^k\, u) = (u, \beta, y)$$

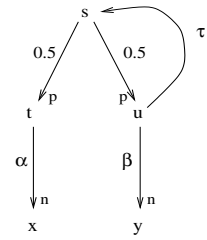


Fig. 3. Determinate schedulers

We define the following set of schedulers:

Definition 7. Let Ξ be a LCMC and $\sigma \in \mathsf{Sched}(\Xi)$. We say that scheduler σ is *determinate* if for all $c, c' \in \mathsf{Comp}_{fin}(\Xi)$ with $\mathsf{last}\, c = \mathsf{last}\, c'$ and $\mathsf{trace}\, c = \mathsf{trace}\, c'$, $\sigma(c) = \sigma(c')$. We write $\mathsf{DSched}(\Xi)$ for the set of determinate schedulers of Ξ. □

It is not difficult to see that $\mathsf{DSched}(\Xi)^\alpha$ is a finite set. Furthermore, it turns out that in order to compute $\mathsf{Pr}_{max}(\Xi, \alpha, \mathcal{M})$ it is sufficient to restrict our attention to $\mathsf{DSched}(\Xi)$ (and consequently to $\mathsf{DSched}(\Xi)^\alpha$).

Theorem 3. Suppose $s \in \mathsf{S}$. Then if $\alpha \in Act$, $\mathcal{M} \in \mathsf{S}/\approx$, (1) if $\sigma \in \mathsf{Sched}(s)$ there exists $\sigma' \in \mathsf{DSched}(s)$ such that $\mathsf{Pr}(s, \tau^*\alpha\tau^*, \mathcal{M}, \sigma) \leq \mathsf{Pr}(s, \tau^*\widehat{\alpha}\tau^*, \mathcal{M}, \sigma')$, and (2) if $\sigma \in \mathsf{Sched}(s, [s]_\approx)$, there exists $\sigma' \in \mathsf{DSched}(s) \cap \mathsf{Sched}'(s, [s]_\approx)$ such that $\mathsf{Pr}(s, \tau^*, \mathcal{M}, \sigma) \leq \mathsf{Pr}(s, \tau^*, \mathcal{M}, \sigma')$. □

Proof: The proof involves transforming an arbitrary scheduler into a determinate one without decreasing the probability of interest, by scheduling from each state the transition that maximizes the desired probability. □

The problem of computing maximum (and minimum) probabilities of properties was also tackled in the context of model checking for probabilistic extensions

of the CTL temporal logic [7,12,4]. In these works, the challenge had been to compute the probability bounds that certain logical properties are satisfied by probabilistic systems. As was shown in the papers just cited, to compute such probabilities it is sufficient to consider only the (finite) set of *simple* schedulers, where a scheduler σ is *simple* if for all c, with $\mathsf{last}\, c = \mathsf{last}\, c'$, $\sigma(c) = \sigma(c')$. Thus, a simple scheduler is also determinate. However, simple schedulers are insufficient for computing $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M})$. For instance consider the LCMC s in Figure 4. None of the two simple schedulers of the system achieves probability $\mathsf{Pr}_{max}(s, \alpha, [u]_{\approx})$. On the other hand, $\mathsf{DSched}(s)$ contains σ, where $\sigma(s) = (s, \tau, t)$, $\sigma(s\tau t) = (t, \alpha, t)$, $\sigma(s\tau t\alpha t) = (t, \tau, u)$, and indeed, $\mathsf{Pr}(s, \alpha, [u]_{\approx}, \sigma) = \mathsf{Pr}_{max}(s, \alpha, [u]_{\approx}) = 1$.

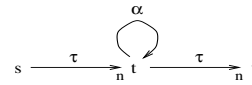


Fig. 4. Simple vs. determinate schedulers

5 The Algorithm

In this section we develop an algorithm for deciding weak bisimulation for labeled concurrent Markov chains. The basic idea originates from the bisimulation algorithms of [24,19], where a partitioning technique is employed to compute strong bisimulation equivalence classes for nonprobabilistic systems. Briefly, this technique involves considering the set of states of the system in question, S, and beginning with the trivial partition of this set $\mathcal{W} = \{S\}$, successively refining the partition until the set of weak bisimulation equivalence classes is reached. The refinement method is based on the notion of 'splitters'.

Definition 8. Let $\mathcal{W}$ be a partition of $S \subset \mathsf{S}$. Then a *splitter* of $\mathcal{W}$ is a triple $(\mathcal{C}, \alpha, \mathcal{M})$, where $\mathcal{C} \in \mathcal{W}$, $\alpha \in Act$ and $\mathcal{M} \in \mathcal{W}$, such that there are $s, s' \in \mathcal{C}$ and $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) \neq \mathsf{Pr}_{max}(s', \alpha, \mathcal{M})$ or $\mu_{\mathcal{W}}(s, \mathcal{M}) > \mathsf{Pr}'_{max}(s', \tau, \mathcal{M}, \mathcal{C})$, or $\mu_{\mathcal{W}}(s', \mathcal{M}) > \mathsf{Pr}'_{max}(s, \tau, \mathcal{M}, \mathcal{C})$.

Thus a splitter $(\mathcal{C}, \alpha, \mathcal{M})$ of a partition $\mathcal{W}$ isolates a class of $\mathcal{W}$ which contains states that prevent $\mathcal{W}$ from being a weak bisimulation. So suppose $(\mathcal{C}, \alpha, \mathcal{M})$ a splitter of a partition $\mathcal{W}$. The purpose of function Split is to partition $\mathcal{C}$ into classes $\mathcal{C}_1 \ldots \mathcal{C}_k$, none of which can be split by the pair $(\alpha, \mathcal{M})$. Formally, $\mathsf{Split}(\mathcal{C}, \alpha, \mathcal{M}) = \mathcal{C}/\equiv$, where $s \equiv t$ *iff* $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) = \mathsf{Pr}_{max}(t, \alpha, \mathcal{M})$, $\mu_{\mathcal{W}}(s, \mathcal{M}) \leq \mathsf{Pr}'_{max}(s', \tau, \mathcal{M}, [s]_{\equiv})$, and $\mu_{\mathcal{W}}(s', \mathcal{M}) \leq \mathsf{Pr}'_{max}(s, \tau, \mathcal{M}, [s']_{\equiv})$. Taking this a step further, given a splitter $(\mathcal{C}, \alpha, \mathcal{M})$ of a partition $\mathcal{W}$, it is possible to refine $\mathcal{W}$ as follows: $\mathsf{Refine}(\mathcal{W}, \mathcal{C}, \alpha, \mathcal{M}) = \mathsf{Split}(\mathcal{C}, \alpha, \mathcal{M}) \cup (\mathcal{W} - \mathcal{C})$. The correctness of the procedure is given by a generalization of Theorem 2.

Theorem 4. Suppose $\mathcal{M} = \bigcup_{i \in I} \mathcal{M}_i$ and $\mathcal{M}' = \bigcup_{i \in I'} \mathcal{M}'_i$, where each $\mathcal{M}_i, \mathcal{M}'_i$ is in $\mathsf{S}/\approx$ and further let $s \approx t$, $s \in \mathcal{M}'$. The following hold:

1. for all $\alpha \in Act$, $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M}) = \mathsf{Pr}_{max}(t, \alpha, \mathcal{M})$, and
2. $\mu_{\approx}(s, \mathcal{M}) \leq \mathsf{Pr}'_{max}(t, \tau, \mathcal{M}, \mathcal{M}')$.

It is easy to see that given a partition $\mathcal{W}$ of a set S, if $\mathcal{W}$ is coarser than $S/\approx$ then there exists a splitter $(\mathcal{C}, \alpha, \mathcal{M})$ of $\mathcal{W}$. Furthermore, given the previous result, $\mathsf{Refine}(\mathcal{W}, \mathcal{C}, \alpha, \mathcal{M})$ is strictly finer than $\mathcal{W}$ and coarser than $S/\approx$. Finally, if no splitter exists for partition $\mathcal{W}$ we may conclude that $\mathcal{W} = S/\approx$. The algorithm for weak bisimulation follows.

Algorithm for Computing Weak Bisimulation Equivalence Classes

Input: $(S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0)$
Output: $(S_n \cup S_p)/\approx$
Method:
$\mathcal{W} := \{S\}$;
$(\mathcal{C}, \alpha, \mathcal{M}) := \mathsf{FindSplit}(\mathcal{W})$;
`while` $\mathcal{C} \neq \emptyset$ `do`
$\mathcal{W} := \mathsf{Refine}(\mathcal{W}, \mathcal{C}, \alpha, \mathcal{M})$;
$(\mathcal{C}, \alpha, \mathcal{M}) := \mathsf{FindSplit}(\mathcal{W})$
`od`
`return` $\mathcal{W}$

It is based on the procedure $\mathsf{FindSplit}(\mathcal{W})$ which, given a partition $\mathcal{W}$, it isolates and returns a splitter $(\mathcal{C}, \alpha, \mathcal{M})$ of $\mathcal{W}$, if one exists, and a triple $(\emptyset, \alpha, \mathcal{M})$, otherwise. It achieves this by considering each $\alpha \in Act$ and each $\mathcal{M} \in \mathcal{W}$ and computing $\mu_{\mathcal{W}}(s, \mathcal{M})$, $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M})$ and $\mathsf{Pr}'_{max}(s, \tau, \mathcal{M}, [s]_{\mathcal{W}})$ for all $s \in S$, thus determining a splitter if one exists. To compute maximum probabilities, $\mathsf{FindSplit}(\mathcal{W})$ makes use of the functions $\mathsf{FindMax}(s, \alpha, \mathcal{M})$ and $\mathsf{FindMax}'(s, \tau, \mathcal{M}, [s]_{\mathcal{W}})$, responsible for computing probabilities $\mathsf{Pr}_{max}(s_0, \alpha, \mathcal{M})$ and $\mathsf{Pr}'_{max}(s_0, \tau, \mathcal{M}, [s_0]_{\mathcal{W}})$ respectively. We describe the former, computation of the latter is similar. To do this, we associate with each state s the variables X^{τ}_s and, if $\alpha \neq \tau$, X^{α}_s. The relationship between the variables is given by the following equations:

$$X^{\alpha}_s = \begin{cases} \sum\limits_{s \xrightarrow{\pi}_p s'} \pi \cdot X^{\alpha}_{s'}, & s \in S_p \\ \max(\{X^{\tau}_{s'} \mid s \longrightarrow_n s'\} \cup \{X^{\alpha}_{s'} \mid s \xrightarrow{\tau}_n s'\}), & s \in S_n \end{cases}$$

$$X^{\tau}_s = \begin{cases} 1, & s \in \mathcal{M} \\ \sum\limits_{s \xrightarrow{\pi}_p s'} \pi \cdot X^{\tau}_{s'}, & s \in S_p - \mathcal{M} \\ \max\limits_{s \xrightarrow{\tau}_n s'} X^{\tau}_{s'}, & s \in S_n - \mathcal{M} \end{cases}$$

We can find a solution for this set of equations by solving a linear programming problem. More precisely, for all equations of the form $X = \max\{X_1, \ldots X_n\}$,

we introduce the set of inequations $X \geq X_i$ and we minimize the function $\sum_{s \in \mathsf{S}} X_s^\alpha + X_s^\tau$. Using algorithms based on the ellipsoid method, this problem can be solved in time polynomial to the number of variables (see, e.g. [20]).

Given a solution to this problem we let $\mathsf{FindMax}(s_0, \alpha, \mathcal{M}) = X_{s_0}^\alpha$ and claim that $X_{s_0}^\alpha = \mathsf{Pr}_{max}(s_0, \alpha, \mathcal{M})$. The correctness of this claim can be proved by appealing to Theorem 3, according to which, $\mathsf{Pr}_{max}(s_0, \alpha, \mathcal{M})$ can be achieved by a determinate scheduler. In particular, we make use of the following observation: since a determinate scheduler $\sigma \in \mathsf{DSched}(S)^\alpha$ schedules partial computations with trace either τ or α, it can be viewed as either (1) a simple scheduler (if $\alpha = \tau$) or (2) the concatenation of two simple schedulers (if $\alpha \in L$), the first responsible for scheduling a transition ending with $\xrightarrow{\alpha}_n$, and the second responsible for scheduling invisible transitions to reach $\mathcal{M}$. Thus, according to the equations above, X_s^α and X_s^τ correspond to $\mathsf{Pr}_{max}(s, \alpha, \mathcal{M})$ and $\mathsf{Pr}_{max}(s, \tau, \mathcal{M})$, respectively.

Therefore, assuming that the size of Act is constant and the size of $S_n \cup S_p$ is N, we have the following theorem.

Theorem 5. The above algorithm for computing weak bisimulation equivalence classes can be computed in time polynomial in N.

We point out that a more efficient formulation of the algorithm is possible, which avoids unnecessary recomputation of probabilities and in the searching of splitters. Such concerns will be relevant in the implementation of the algorithm.

6 Concluding Remarks

In this paper we have defined the notion of weak bisimulation for Labeled Concurrent Markov Chains. We have developed a method for deciding weak bisimulation and presented an algorithm which computes weak bisimulation equivalence classes with polynomial-time complexity in the size of the transition system. Due to the generality of our framework, our results can be adopted to other models in which nondeterminism and probabilistic behavior co-exist.

Although not described in the paper, we have also investigated the relationship of the proposed definition with existing proposals of weak bisimulation [25]. In particular, we have shown, that when restricted to a fully probabilistic model the definition presented here coincides with that of [3]. Furthermore, we have compared the definition we propose to that defined by [29] for probabilistic automata. A probabilistic automaton is an automaton whose states allow the nondeterministic choice among a number of probability distributions, each of which involves probabilistic transitions associated with a single action. The weak bisimulation definition of the model, here denoted as $\approx_A$, requires that if automata A_1 and A_2 are weakly bisimilar and A_1 can engage in a transition involving a probability distribution f, then A_2 can engage in a weak transition which combines probability distributions (in a serial manner) to one that is equivalent to f, in the sense that both distributions assign the same probability to the same equivalence classes. For example, automata A and M of Figure 5(a) are bisimilar to

each other: A can match every transition of M. In addition, A's initial a-labeled transition can be weakly matched by M by combining the initial τ-labeled distribution and consequently the two a labeled ones.

To compare the two bisimulations we have considered translations between LCMC's and probabilistic automata, which describe how LCMC's can be captured by probabilistic automata and vice versa. This is done with the aid of two functions $[\cdot] : \mathsf{Aut} \longrightarrow \mathsf{S}$ and $[[\cdot]] : \mathsf{S} \longrightarrow \mathsf{Aut}$, illustrated in Figure 6, where S and Aut are the sets of LCMC's and probabilistic automata respectively. We may then prove that if two LCMC's are bisimilar to each other then their translations are also bisimilar with respect to the definition of [29], that is if $s \approx s'$ then $[[s]] \approx_A [[s']]$, except in the case when some LCMC state s performs a probabilistic transition to $s_{[\approx]}$ (see Figure 2). This is due to the fact that [29] does not consider weighted probabilities in the manner of Definition 6(2). However, a similar result does not hold in the opposite direction. That is, it is not the

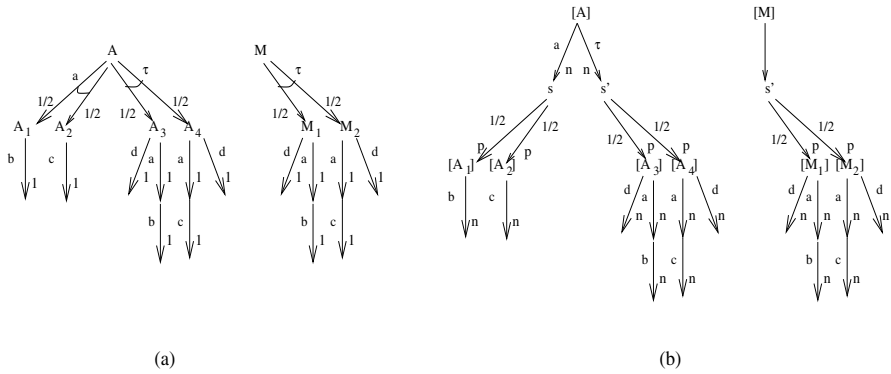


Fig. 5. (a) $A \approx_A M$, (b) $[A] \not\approx [M]$

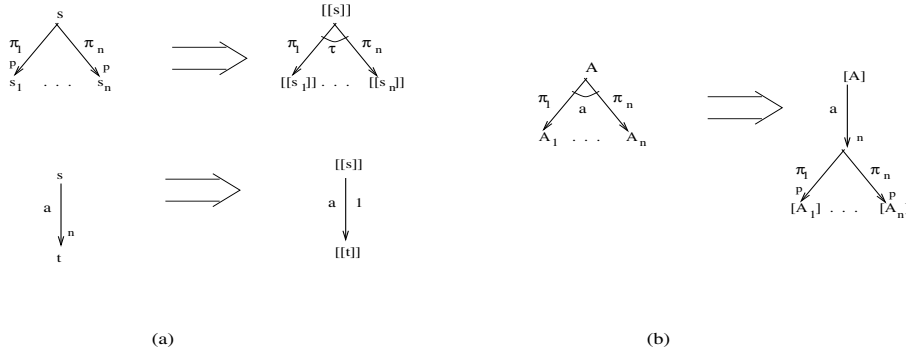


Fig. 6. Translation functions $[[\cdot]]$ (a) and $[\cdot]$ (b)

case that if $A \approx_A A'$ then $[A] \approx [A']$. A counter-example is provided by the automata of Figure 5(a). As illustrated Figure 5(b) the LCMC $[A]$ can probabilistically reach state s. However there is no matching transition of $[M]$. This fact is not surprising considering the separation made between nondeterministic and probabilistic states in the LCMC-model.

An additional notion defined in [29] is that of *probabilistic* weak bisimulation, which extends $\approx_A$ by allowing schedulers to range over the set of randomized schedulers. One of the motivations behind this extension was to define a notion that preserves properties expressed in PCTL. We believe that such an approach will not be necessary for the weak bisimulation we propose. The logical characterization of this definition is an issue we are currently investigating.

In related work, we are studying the notion of weak bisimulation within the context of the PACSR process algebra [26], a probabilistic extension of ACSR [22] which is a real-time process algebra that captures the notions of priorities and resources. In this area work is being carried out with aims the axiomatization of the congruence induced by weak bisimulation, and the extension of existing tools with automated weak-bisimulation checking and state-space minimization.

Acknowledgments

We are grateful to Rance Cleaveland and Scott Smolka for enlightening initial discussions as well as Marta Kwiatkowska and Roberto Segala for interesting comments and suggestions.

References

1. J. Baeten and J. Klop, editors. *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. 348
2. C. Baier. Polynomial time algorithms for testing probabilistic bisimulation and simulation. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, volume 1102 of *Lecture Notes in Computer Science*, pages 38–49. Springer-Verlag, July/August 1996. 335
3. C. Baier and H. Hermanns. Weak bisimulation for fully probabilistic processes. In *Proceedings of the 9th International Conference on Computer Aided Verification,* Haifa, volume 1254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997. 335, 341, 345
4. C. Baier and M. Kwiatkowska. Automatic verification of liveness properties of randomized systems. In *Proceedings of the* 14^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Santa Barbara, California, August 1997. 343
5. C. Baier and M. Stoelinga. Norm functions for bisimulations with delays. In *Proc. FOSSACS'00*. Springer-Verlag, 2000. 335
6. M. Bernardo and R. Gorrieri. Extended markovian process algebra. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR 96,* Pisa, Italy, volume 1119 of *Lecture Notes in Computer Science*, pages 315–330. Springer-Verlag, 1996. 334

7. A. Bianco and R. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proceedings Foundations of Software Techonology ans Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1995. 343
8. I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In Baeten and Klop [1], pages 126–140. 335
9. L. Christoff and I. Christoff. Efficient algorithms for verification of equivalences of probabilistic processes. In K. Larsen and A. Skou, editors, *Proceedings of the 3rd International Workshop on Computer Aided Verification,* Aalborg, Denmark, volume 575 of *Lecture Notes in Computer Science*, pages 310–321. Springer-Verlag, 1991. 335
10. R. Cleaveland, S. Smolka, and A. Zwarico. Testing preorders for probabilistic processes. In W. Kuich, editor, *Proceedings* 19^{th} *ICALP,* Vienna, volume 623 of *Lecture Notes in Computer Science*, pages 708–719. Springer-Verlag, 1992. 335
11. C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proceedings* 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 338–345. IEEE, 1988. 334
12. C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In M. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 336–349. Springer-Verlag, July 1990. 343
13. R. van Glabbeek, S. Smolka, B. Steffen, and C. Tofts. Reactive, generative, and stratified models of probabilistic processes. In *Proceedings* 5^{th} *Annual Symposium on Logic in Computer Science,* Philadelphia, USA, pages 130–141. IEEE Computer Society Press, 1990. 334
14. H. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1991. DoCS 91/27. 334, 335, 337, 338
15. J. Hillston. PEPA: Performance enhanced process algebra. Technical Report CSR-24-93, University of Edinburgh, UK, 1993. 334
16. B. Jonsson, C. Ho-Stuart, and W. Yi. Testing and refinement for nondeterministic and probabilistic processes. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings of the Third International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'94),* Lübeck, Germany, September 1994, volume 863 of *Lecture Notes in Computer Science*, pages 418–430. Springer-Verlag, 1994. 335
17. B. Jonsson and K. Larsen. Specification and refinement of probabilistic processes. In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, pages 266–277. IEEE Computer Society Press, 1991. 335
18. C. Jou and S. Smolka. Equivalences, congruences and complete axiomatizations for probabilistic processes. In Baeten and Klop [1]. 335
19. P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990. 343
20. H. Karloff. *Linear Programming.* Progress in Theoretical Computer Science. Birkhauser, 1991. 345
21. K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94, 1991. 334, 335
22. I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994. 347

23. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989. 334, 335, 338
24. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. 343
25. A. Philippou, I. Lee, and O. Sokolsky. Weak bisimulation for probabilistic systems. Technical report, University of Cyprus, 1999. 336, 345
26. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In D. Sangiorgi and R. de Simone, editors, *Proceedings CONCUR 98,* Nice, France, volume 1446 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1998. 347
27. A. Pnueli and L. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993. 334
28. R. Segala. *Modelling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995. 334, 335
29. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*, pages 481–496. Springer-Verlag, 1994. 335, 336, 337, 345, 346, 347
30. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings 26th Annual Symposium on Foundations of Computer Science*, pages 327–338. IEEE, 1985. 334, 335, 337

Nondeterminism and Probabilistic Choice: Obeying the Laws

Michael Mislove*

Department of Mathematics, Tulane University
New Orleans, LA 70118
mwm@math.tulane.edu

Abstract. In this paper we describe how to build semantic models that support both nondeterministic choice and probabilistic choice. Several models exist that support both of these constructs, but none that we know of satisfies all the laws one would like. Using domain-theoretic techniques, we show how models can be devised using the "standard model" for probabilistic choice, and then applying modified domain-theoretic models for nondeterministic choice. These models are distinguished by the fact that the expected laws for nondeterministic choice and probabilistic choice remain valid. We also describe a potential application of our model to aspects of security.

1 Introduction

The most widely employed method for modeling concurrent computation is to take sequential composition as a primitive operator, and then to use nondeterministic choice to generate an interleaving semantics for parallel composition. This approach is well-supported by the models of computation, including both the standard domain-theoretic models (cf. [8]), and the metric space approach (cf. [2]). These and similar approaches to modeling nondeterminism satisfy the basic assumption that nondeterministic choice is a commutative, associative and idempotent operation. In fact, the results from [8] characterize the three fundamental *power domains* in terms of their universal properties as ordered semilattices – i.e., that they each are the object-level of a left adjoint to a forgetful functor from an appropriate category.

More recently, probabilistic choice has been added as a family of operators in the syntax of the language under study. One can trace this research in domain theory to the work of Saheb-Dharjomi [21]. While this was the first to consider modeling probabilistic choice using a domain, the most influential work along this line is without question the PhD thesis of Jones [9], where it was shown that Saheb-Dharjommi's construction could be extended to "measures" having total variation less than 1, and that the probabilistic power domain of a continuous

* Partial support provided by the National Science Foundation and the US Office of Naval Research

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 350–365, 2000.

domain is again continuous. More importantly, Jones provided a finitary characterization of the probabilistic power domain in terms of equations the operators should satisfy. If these equations hold, then her model is initial.

One issue that causes problems is that the difference between nondeterministic choice and probabilistic choice is not clearly understood. Indeed, the title of [9] reveals an identification of probabilistic choice as a form of nondeterminism. Yet probabilistic choice operators are not associative. For example,[1]

$$(p \,{}_{.5}\!+ q) \,{}_{.5}\!+ r = p \,{}_{.25}\!+ (q \,{}_{1/3}\!+ r).$$

Still, several authors have attempted to incorporate *both* nondeterministic choice and probabilistic choice within one model. None that we know of has accomplished that goal completely satisfactorily. For example, in [17] a model incorporating probabilistic choice is built by simply applying Jones' probabilistic power domain to the standard failures-divergences model for CSP. But, the natural extension of nondeterministic choice to this model is not idempotent, so a fundamental law of nondeterminism fails in the model. To explain this anomaly, it is argued in [18] that the probability that the process $(p \,{}_{.5}\!+ q) \sqcap (p \,{}_{.5}\!+ q)$ actually acts like p is .25, since each branch resolves the probabilistic choice independently. On the other hand, one might argue that the process in question is supposed to act like one branch, not like both – ie, the probabilistic choice should be resolved *after* the nondeterministic choice is resolved. But models for CSP typically do not discriminate closely enough to keep track of the order in which choices are resolved, something that is reflected by the fact that the nondeterministic choice operator distributes through the probabilistic choice operators.

This brings us to the issue we are interested in confronting: how to build denotational models for general process algebras which support both nondeterministic choice and probabilistic choice, so that the laws for nondeterministic choice and for probabilistic choice that one expects to hold actually are valid. Our construction relies heavily on domain theory and some of the constructs it provides. The work here is closely related to the emerging area of devising semantic models using coalgebraic techniques (cf., e.g., [20] for an introduction).

There are several approaches that have been put forward for modeling probabilistic choice, including

- approaches such as [14,16] that focus on state-based models and use probabilistic transition systems to reflect the operational behavior of the system under study. In these approaches, discrete probabilistic models are considered, and the focus is on the probability of a process being in a given state *after* it has executed a given action.
- approaches such as [5] and [15] that use a process algebra in which probabilistic choice is substituted for nondeterministic choice. Here, and in the next case, the focus is on the probability that the process in question acts like one branch or the other from the choices listed.

[1] We will use the notation $p \,{}_{\lambda}\!+ q$ to denote a probabilistic choice in which the process has probability λ of acting like p, and probability $1 - \lambda$ of acting like q, where $0 \leq \lambda \leq 1$.

– approaches such as [17,6] that extend a process algebra by adding probabilistic choice operators. If the branches have distinct initial actions, then the focus is on the probability of the process executing a given action, rather than on the state after a given action is executed.

The approach nearest our own is the last, but, as just remarked, none of these approaches provides a semantic model in which the laws we are interested in hold. Moreover, there are close links between all these approaches, so they should be viewed as variants of one another. Our goal in this paper is to show how a model supporting both nondeterministic choice and probabilistic choice can be devised, so that the expected laws for nondeterministic choice and for probabilistic choice all hold.

As stated above, we use domain theory as the basis for the constructions we devise. There is a long history of modeling probabilistic choice in this area, dating back to the seminal work of Saheb-Djarhomi [21] in which a now standard construction of a cpo supporting probabilistic choice was given, beginning with an underlying cpo. This work led to the results in [9,10] that clarified and expanded the nature of Saheb-Djarhomi's construction, and also showed that this construction, when applied to a continuous domain, yields a continuous domain. This is the construction used in [17] for *probabilistic CSP*, which is simply the probabilistic power domain $\mathcal{P}_{Pr}(\mathbb{FD})$ of the failures-divergences model $\mathbb{FD}$ for untimed CSP. As we noted above, the extension of the nondeterministic choice operation from $\mathbb{FD}$ to $\mathcal{P}_{Pr}(\mathbb{FD})$ is not idempotent. In fact, there is no *convex* idempotent operation on $\mathcal{P}_{Pr}(\mathbb{FD})$ that extends the nondeterministic choice operator on (the image of) $\mathbb{FD}$ (in $\mathcal{P}_{Pr}(\mathbb{FD})$), since the extension used in [17] is one such, and the *Splitting Lemma* (cf. [9] or Lemma 1) implies there is only one such. There is no obvious candidate for a nondeterministic choice operator on this model, even if one drops the convexity hypothesis.

Our approach to remedying this problem is to take the construction one step further: we apply a power domain operator to $\mathcal{P}_{Pr}(\mathbb{FD})$. In fact, there are three possible power domains to apply - the lower, the upper and the convex power domains. While these produce new nondeterministic choice operators, one soon discovers that the probabilistic choice operators on $\mathcal{P}_{Pr}(\mathbb{FD})$, when lifted to any of these power domains do not satisfy the expected laws. However, we are not far from our desired model. We simply consider the family of *(probability) convex* sets in each of the respective power domains, and we find that in each case, these do provide models where all the laws – both those of nondeterminism and of probabilistic choice – are valid. (This is an idea suggested in [17], but considered there only in the case of the upper power domain, and for which no details are presented.) What is more, in the case of the lower or upper power domains, the models we construct yield a bounded complete domain,[2] when applied to a Scott domain. In addition, the compositions $\mathcal{P}_{PL} \circ \mathcal{P}_{Pr}$ and $\mathcal{P}_{PU} \circ \mathcal{P}_{Pr}$ are

[2] A domain is *bounded complete* if each non-empty subset has a greatest lower bound. Equivalently, each subset with an upper bound has a least upper bound. These objects, when it also is assumed that the domain is ω-algebraic, form what are called *Scott domains*.

continuous endofunctors of the category BCD of continuous, bounded complete domains – the continuous analogues of Scott domains. Since this category is cartesian closed, one can in principle construct models for the lambda calculus extended to include both probabilistic choice operators and nondeterministic choice operators. Unfortunately, as far as we know, this result does not extend to the case of the convex power domain: while $\mathcal{P}_{PC} \circ \mathcal{P}_{Pr}$ is continuous, we are unable to show it lands back in RB, the category of retracts of bifinite domains. Even so, our models have the added bonus that the probabilistic choice operators do not distribute through the nondeterministic choice operators, which has an important implication for the application of these models to the area of security. We outline this application in the last section of the paper.

The rest of the paper is organized as follows. In the next section we review some background in domain theory and we review the principal construction of a model for probabilistic choice in domains – the probabilistic power domain, followed by a description of PCSP from [17]. This serves to present a motivating example for our work; it demonstrates how the failure of an expected law in a semantic model can lead to unexpected results in the behavior of a process. Actually, such results are inevitable if one uses the model constructed in [17] because of the way in which the CSP operators are defined on their model. The next section gives our construction, showing how to build a model supporting both nondeterministic choice and probabilistic choice over any bounded complete domain. We next describe some potential applications of our models, and some of the problems that still need to be resolved to provide meaningful answers to these questions.

2 Domains and the Probabilistic Power Domain

In this section, we review some of the basics we need to describe our results. A good reference for most of this can be found in [1]. To begin, a *partial order* is a non-empty set endowed with a reflexive, antisymmetric and transitive relation. If P is a partial order, then a subset $D \subseteq P$ is *directed* if every finite subset of D has an upper bound in D. We say P is *directed complete* if every directed subset of D has a least upper bound, denoted $\sqcup D$, in P. Such partial orders we call *dcpos*, and we use the term *cpo* for a dcpo that also has a least element, usually denoted $\bot$.

Dcpos can be endowed with a topology that plays a fundamental role in the theory. A subset $U \subseteq P$ is *Scott open* if $U = \uparrow U = \{x \in P \mid (\exists u \in U)\, u \leq x\}$ is an upper set, and, for every directed set D, if $\sqcup D \in U$, then $D \cap U \neq \emptyset$. The Scott continuous functions $f\colon P \to Q$ between dcpos are easy to characterize order-theoretically: they are exactly the maps that preserve the order and also preserve suprema of directed sets – $f(\sqcup D) = \sqcup f(D)$ for all $D \subseteq P$ directed.

The category DCPO of (d)cpos and Scott continuous maps is a cartesian closed category. More precisely, the product of (d)cpos is another such, there is a terminal object among dcpos – the one point dcpo – and there is an *internal hom*: for dcpos P and Q, the family $[P \to Q]$ of continuous maps between them

is a dcpo in the pointwise order, and $[P \times Q \to R] \simeq [P \to [Q \to R]]$ for dcpos P, Q and R. What is just as important is that we can find minimal solutions to *domain equations* within these categories, assuming the equations are defined by continuous endofunctors defined on DCPO (cf. [1]).

Continuous domains: If P is a dcpo and $x \leq y \in P$, then we write $x \ll y$ if and only if $(\forall D \subseteq P \text{ directed})\, y \leq \sqcup D \;\Rightarrow\; (\exists d \in D)\, x \leq d$. P is *continuous* if $\Downarrow y = \{x \in P \mid x \ll y\}$ is directed and $y = \sqcup \Downarrow y$ for all $y \in P$. Unfortunately, the category CON of continuous domains and Scott continuous maps is not cartesian closed. In fact, a classification of the maximal cartesian closed subcategories of CON is given in [1].

Coherent domains: Of particular interest to us is the category COH of *coherent domains* and Scott continuous maps (cf. [1]). These domains are most easily described in topological terms. The Scott topology on a domain P satisfies only weak separation conditions: it is *sober* (which for continuous domains can be seen as a topological statement that P is directed complete), and all open sets are upper sets. In order to refine the topology to obtain a Hausdorff topology, one has only to add certain open lower sets. These are the family $\{P \setminus \uparrow F \mid F \subseteq P \text{ finite}\}$. The common refinement of this topology and the Scott topology is called the *Lawson topology.*

Definition 1. *A domain P is* coherent *if its Lawson topology is compact. Equivalently, the intersection of Scott compact upper sets is again Scott compact.*

Power domains: Continuous domains admit standard models for nondeterminism, each of which is the object level of a left adjoint to an appropriate forgetful functor. In the case of coherent domains, these *power domains* can be defined as:

The Lower Power Domain is defined as $\mathcal{P}_L(P) = \{X \subseteq D \mid \emptyset \neq X = \downarrow X \text{ is Scott closed}\}$, ordered by inclusion.

The Upper Power Domain is defined as $\mathcal{P}_U(D) = \{X \subseteq P \mid \emptyset \neq X = \uparrow X \text{ is Scott compact}\}$ ordered by reverse inclusion.

The Convex Power Domain can then be defined as $\mathcal{P}_C(P) = \{X \subseteq P \mid X = \downarrow X \cap \uparrow X \;\wedge\; \downarrow X \in \mathcal{P}_L(P) \;\wedge\; \uparrow X \in \mathcal{P}_U(P)\}$, ordered by $X \sqsubseteq Y$ iff $\downarrow X \subseteq \downarrow Y$ and $\uparrow X \supseteq \uparrow Y$.

Each of these constructs is a *continuous semilattice:* each admits an associative, commutative and idempotent operation that preserves directed suprema (in the first two cases, the operation is simply union, while in the last it is obtained by taking the *(order) convex hull* of the union of the components). Moreover, each is the object level of a left adjoint to a forgetful functor from an appropriate category of ordered semilattice domains and Scott continuous maps to the category of coherent domains and Scott continuous maps (cf. [8]).

The probabilistic power domain: We now describe the construction that allows probabilistic choice operators to be added to a domain. This construction was first investigated by Saheb-Djarhomi [21], who showed that the family he defined yields a cpo. The construction later was refined by Jones [9,10] where it also was shown that the probabilistic power domain of a continuous domain is again continuous. The definition of the more general construction goes as follows.

Definition 2. *If P is a dcpo, then a* continuous valuation *on P is a mapping $\mu\colon \Sigma D \to [0,1]$ defined on the Scott open subsets of P that satisfies:*

1. $\mu(\emptyset) = 0$.
2. $\mu(U \cup V) = \mu(U) + \mu(V) + \mu(U \cap V)$,
3. μ *is monotone, and*
4. $\mu(\cup_i U_i) = \sup_i \mu(U_i)$, *if* $\{U_i \mid i \in I\}$ *is an increasing family of Scott open sets.*

We order this family pointwise: $\mu \leq \nu \;\Leftrightarrow\; \mu(U) \leq \nu(U)$ $(\forall U \in \Sigma P)$, *and we denote the family of continuous valuations on P by $\mathcal{P}_{Pr}(P)$.*

It was Lawson [13] who first showed the connection between continuous valuations and measures on the cpo P: he showed that, in the case P has a countable basis, there is a one-to-one correspondence between regular Borel measures on P and continuous valuations on ΣP. This result has recently been generalized to a much larger category of topological spaces.

The probabilistic power domain construction has been fraught with problems almost from its inception. An excellent discussion of this can be found in [12]. One of the key properties of domain theory has been the ample supply of cartesian closed categories that are closed under each of the constructs the theory has to offer. For example, the constructions needed to build Scott's D_∞ model all leave the category of continuous, bounded complete domains invariant. It was the fact that the convex power domain does not leave this category invariant that led to the discovery of the cartesian closed category of bifinite domains and Scott continuous maps. *Bifinite domains* are those that can be expressed as the limit of a directed family of finite posets under embedding-projection pairs; they all are algebraic, while the category RB of *retracts of bifinite domains* is a cartesian closed category containing continuous domains such as the unit interval. Since $\mathcal{P}_{Pr}(P)$ is continuous if P is, but never algebraic, the natural question is whether the cartesian closed category RB is closed under this construction. The answer remains unknown. More generally, there is no known cartesian closed category of continuous domains that is closed under the probabilistic power domain operator. This means, in particular, that the only cartesian closed categories which are known to be closed under this construct are CPO and DCPO, the categories of cpos (dcpos) and Scott continuous maps, respectively. This is unsatisfactory, since so little is known about the structure of the objects in these categories.

Among the continuous valuations on a dcpo, the *simple valuations* are particularly easy to describe. They are of the form $\mu = \Sigma_{x \in F} r_x \cdot \delta_x$, where $F \subseteq P$ is a finite subset, δ_x represents point mass at x (the mapping sending an open set

to 1 if it contains x, and to 0 otherwise), and $r_x \in [0,1]$ satisfy $\Sigma_{x\in F} r_x \leq 1$. In this case, the *support* of μ is just the family F. The so-called *Splitting Lemma* of [9] is a fundamental result about the order on simple measures:

Lemma 1 (Splitting Lemma [9]). *If $\mu = \Sigma_{x\in F} r_x \cdot \delta_x$ and $\nu = \Sigma_{y\in G} s_y \cdot \delta_y$ are simple valuations, then $\mu \leq \nu$ if and only if there is a family of non-negative real numbers $\{t_{x,y} \mid x \in F, y \in G\}$ satisfying*

1. *For all $x \in F$, $\Sigma_{y\in G} t_{x,y} = r_x$.*
2. *For all $y \in G$, $\Sigma_{x\in F} t_{x,y} \leq s_y$, and*
3. *If $t_{x,y} \neq 0$, then $x \leq y$.*

Moreover, $\mu \ll \nu$ if and only if $\leq$ is replaced by $<$ in 2), and by $\ll$ in 3). □

It follows from this result that the probabilistic power domain of a continuous domain is again continuous. But nothing much more is known about the structure of $\mathcal{P}_{Pr}(P)$; in particular, a simple example is given in [9] of a bounded complete domain P for which $\mathcal{P}_{Pr}(P)$ is not bounded complete.

One fact about the probabilistic power domain that has been established is that it leads to a continuous endofunctor on CON. That is, each continuous map $f\colon P \to Q$ between (continuous) domains can be lifted to a continuous map $\mathcal{P}_{Pr}(f)\colon \mathcal{P}_{Pr}(P) \to \mathcal{P}_{Pr}(Q)$ by $\mathcal{P}_{Pr}(f)(\mu)(U) = \mu(f^{-1}(U))$. In fact, [9] shows that the resulting functor is a left adjoint, which means that $\mathcal{P}_{Pr}(P)$ is a free object over P in an appropriate category. The category in question can be described in terms of probabilistic choice operators satisfying the following laws (cf. [9]):

Definition 3. *A* probabilistic algebra *is a dcpo A endowed with a continuous mapping $(\lambda, a, b) \mapsto a\ {}_\lambda{+}\ b\colon [0,1] \times A \times A \to A$ so that the following laws hold for all $a, b, c \in A$, $\lambda \in [0,1]$:*

- $a\ {}_\lambda{+}\ b = b\ {}_{1-\lambda}{+}\ a$,
- $(a\ {}_\lambda{+}\ b)\ {}_\rho{+}\ c = a\ {}_{\lambda\rho}{+}\ (b\ {}_{\frac{\rho(1-\lambda)}{1-\lambda\rho}}{+}\ c)$ (if $\lambda\rho < 1$).
- $a\ {}_\lambda{+}\ a = a$, *and*
- $a\ {}_1{+}\ b = a$.

The operations ${}_\lambda{+}$ are defined on $\mathcal{P}_{Pr}(P)$ in a pointwise fashion, so for instance, $\mu\ {}_\lambda{+}\ \nu = \lambda\mu + (1-\lambda)\nu$. It then is routine to verify that $\mathcal{P}_{Pr}(P)$ is a probabilistic algebra over P for each dcpo P. Moreover, Jones [9] shows that the probabilistic power domain forms the object level of a left adjoint to the category PROB of continuous probabilistic algebra domains and continuous mappings which also preserve the probabilistic choice operators.

2.1 Probabilistic CSP

The model for probabilistic CSP – PCSP as it is denoted – that was devised in [17] is now easy to describe. It is built by simply applying the probabilistic power domain operator to the failures-divergences model for CSP. But some

extra information is provided to allow a better understanding of the structure of the model.

First, it is shown in [17] that $\mathbb{FD}$ is an algebraic cpo: indeed, the compact elements are the "truncated processes" $\{p{\downarrow_n} \mid p \in \mathbb{FD} \;\&\; n \geq 0\}$, where $p{\downarrow_n}$ is the process that acts like p for at most n steps, and then diverges (recall that DIV is the least element of $\mathbb{FD}$). In fact, in [17] it is shown that the n-step truncations of any process form an increasing sequence whose supremum is the original process, and it is easy to show that $p{\downarrow_n}$ is compact for every n. Moreover, the very definition of $\mathbb{FD}$ allows one to conclude that the union of any non-empty family of processes in $\mathbb{FD}$ is another such, which combined with the result just cited shows that $\mathbb{FD}$ is a Scott domain. Applying the probabilistic power domain operator to $\mathbb{FD}$ then results in a continuous probabilistic algebra, and this is the model for probabilistic CSP used in [17].

The syntax of PCSP is not much different from that of CSP. Indeed, PCSP simply adds the family of operators $_\lambda{+}$ for $0 \leq \lambda \leq 1$ to the usual family of operators of untimed CSP. So, for example, we can reason about processes such as $(a \rightarrow STOP)\,{}_\lambda{+}\,(b \rightarrow STOP \;\Box\; c \rightarrow STOP)$, which will act like $a \rightarrow STOP$ with probability λ, and offer the external choice of doing a b or a c with probability $1 - \lambda$. The approach provided in [17] to reasoning about such processes is via weakest precondition semantics, where weakest preconditions for probabilistic processes are represented as random variables.

The extension of the operators of CSP to $\mathcal{P}_{Pr}(\mathbb{FD})$ can be understood in terms of the construction. Namely, $\mathcal{P}_{Pr}(\mathbb{FD})$ is a set of continuous mappings from the set of Scott open sets of $\mathbb{FD}$ to the unit interval. So, for example, a unary operator $f\colon \mathbb{FD} \rightarrow \mathbb{FD}$ can be extended to $\mathcal{P}_{Pr}(f)\colon \mathcal{P}_{Pr}(\mathbb{FD}) \rightarrow \mathcal{P}_{Pr}(\mathbb{FD})$ by $\mathcal{P}_{Pr}(f)(\mu)(U) = \mu(f^{-1}(U))$. Similar reasoning shows how to extend operators of higher arity (this relies on the fact that the product of Scott open sets is again Scott open). Two facts emerge from this method:

- If we embed $\mathbb{FD}$ into $\mathcal{P}_{Pr}(\mathbb{FD})$ via the mapping $p \mapsto \delta_p$, then the interpretation of each CSP operator on $\mathbb{FD}$ *extends* to a continuous operator on $\mathcal{P}_{Pr}(\mathbb{FD})$: this means that the mapping from $\mathbb{FD}$ into $\mathcal{P}_{Pr}(\mathbb{FD})$ is compositional for all the operators of CSP. This has the consequence that any laws that the interpretation of CSP operators satisfy on $\mathbb{FD}$ still hold *on the image of* $\mathbb{FD}$ *in* $\mathcal{P}_{Pr}(\mathbb{FD})$.
- The way in which the operators of CSP are extended to the model of PCSP forces all the CSP operators to distribute through the probabilistic choice operators. For example, we have

$$a \rightarrow (p\,{}_\lambda{+}\,q) = (a \rightarrow p)\,{}_\lambda{+}\,(a \rightarrow q),$$

  for any event a and any processes p and q. This has the result that some of the laws of CSP fail to hold *on* $\mathcal{P}_{Pr}(\mathbb{FD})$ *as a whole.*

Here is an example illustrating the second point:

Example 1. Consider the process

$$(p\,{}_{.5}{+}\,q) \sqcap (p\,{}_{.5}{+}\,q).$$

The internal choice operator $\sqcap$ is supposed to be idempotent, but using the fact that, when lifted to PCSP, the CSP operators distribute through the probabilistic choice operators, we find that

$$(p \,{}_{.5}{+}\, q) \sqcap (p \,{}_{.5}{+}\, q) = p \,{}_{.25}{+}\, ((p \sqcap q) \,{}_{1/3}{+}\, q),$$

which means that the probability that the process acts like p is somewhere between .25 and .75, depending on how the choice $p \sqcap q$ is resolved. This unexpected behavior can be traced to the fact that $\sqcap$ distributed through ${}_{.5}{+}$ (and through ${}_{\lambda}{+}$ for all λ). One way to view this is that the resolution of the probabilistic choice in $p \,{}_{.5}{+}\, q$ is an internal event, and using the CSP paradigm of *maximal progress* under which internal events are always on offer and happen as soon as possible, the probabilistic choice then is resolved at the same time as the *internal* nondeterministic one. Then the processes on either side of $\sqcap$ represent distinct instances of the same processes, but because they are distinct, the probabilistic choice is resolved independently in each branch. In [17], the term *duplication* is used for this phenomenon. We are unable to assign a precise probability to this process acting like p, since we have no way to assign a probability to how $\sqcap$ resolves its choices, precisely since it is *not* a probabilistic choice operator. Further work in [18] addresses the question of duplication arising where it is not desired, and two possible solutions are presented there.

Our interest is in studying how to overcome duplication at the nondeterministic choice level. Since it is the fact that $\sqcap$ distributes through ${}_{\lambda}{+}$ that causes $\sqcap$ not to be idempotent, one way to avoid this issue would be to craft a model which forces us to resolve $\sqcap$ first, *before* the probabilistic choices are resolved.

3 Constructing New Models

In this section we show how, given an continuous cpo P, we can construct a domain Q which supports nondeterministic choice and probabilistic choice, so that the choice operator is idempotent. In fact, we can construct three such domains Q, each of which is an analog of one of the power domains. Moreover, if P is bounded complete, then in the first two cases, so is the associated Q, and in each case, Q is the image of either a closure operator or a kernel operator.

We start with an arbitrary coherent cpo P. We want to construct a coherent domain which contains a copy of P, that admits a projection onto P, and that simultaneously supports both an idempotent nondeterministic choice operation $+$ and probabilistic choice operations ${}_{\lambda}{+}$ satisfying the laws of a probabilistic algebra.

The failures-divergences model $\mathbb{FD}$ is a Scott domain, and the operators from CSP are extended to the model $\mathcal{P}_{Pr}(\mathbb{FD})$ for PCSP constructed in [17] using the categorical results, which can be traced through the construction of $\mathcal{P}_{Pr}(\mathbb{FD}) \subseteq [\Sigma(\mathbb{FD}) \to [0,1]]$. This method of construction forces the lifting of the operations from $\mathbb{FD}$ to this family all to distribute through the probabilistic choice operators. And, since the simple measures are Scott dense in $\mathcal{P}_{Pr}(\mathbb{FD})$ (a result of the

Splitting Lemma 1), it follows that there is no extension of $\sqcap$ to $\mathcal{P}_{Pr}(\mathbb{FD})$ if we require the extension to preserve convex combinations of processes such as $p \,{}_\lambda\!+ q$.

A somewhat more esoteric question revolves around the structure of the model $\mathcal{P}_{Pr}(\mathbb{FD})$. Indeed, all that one can confidently assert about the probabilistic power domain of a continuous domain is that it is again continuous, and that the probabilistic power domain of a coherent continuous domain is again coherent (cf. [11]). In particular, it remains an open question whether this functor leaves any cartesian closed category of continuous domains invariant. In the case of the lower and upper power domains, our approach is to avoid this issue entirely by "dragging" $\mathcal{P}_{Pr}(\mathbb{FD})$ back into the category of bounded complete domains by applying another functor:

Theorem 1. *If P is a continuous domain, then $\mathcal{P}_L(D), \mathcal{P}_U(D) \in \mathsf{BCD}$, and if P is coherent, then so is $\mathcal{P}_C(P)$. In particular, for any continuous domain P, $\mathcal{P}_L(\mathcal{P}_{Pr}(P))$ and $\mathcal{P}_U(\mathcal{P}_{Pr}(P))$ are both bounded complete and continuous, and $\mathcal{P}_C(\mathcal{P}_{Pr}(P))$ is coherent.*

Proof. One can find a proof that $\mathcal{P}_L(P)$ and $\mathcal{P}_U(P)$ are both bounded complete and continuous if P is continuous, and a proof that $\mathcal{P}_C(P)$ is coherent if P can be found in [1]. The last part then follows from [11]. □

Jones [9] showed that the probabilistic power domain functor is continuous, and it is well known that the power domain functors $\mathcal{P}_L$, $\mathcal{P}_U$ and $\mathcal{P}_C$ are continuous, so the compositions $\mathcal{P}_L \circ \mathcal{P}_{Pr}$, $\mathcal{P}_U \circ \mathcal{P}_{Pr}$ and $\mathcal{P}_C \circ \mathcal{P}_{Pr}$ are all continuous. Moreover, the theorem above yields:

Corollary 1. *The compositions $\mathcal{P}_L \circ \mathcal{P}_{Pr}$ and $\mathcal{P}_U \circ \mathcal{P}_{Pr}$ are continuous endofunctors of BCD, and $\mathcal{P}_C \circ \mathcal{P}_{Pr}$ is a continuous endofunctor of COH.* □

However, this is not what we want. The reason is that, if we use the standard approach to extending the operations from P to $\mathcal{P}_L(P)$, $\mathcal{P}_U(P)$ or $\mathcal{P}_C(P)$ in the case P is a probabilistic algebra, we find that the laws we want no longer are valid. For example, for $X, Y \in \mathcal{P}_U(P)$

$$X \,{}_\lambda\!+ Y = \{x \,{}_\lambda\!+ y \mid x \in X, y \in Y\}, \text{ so } X \,{}_\lambda\!+ X = \{x \,{}_\lambda\!+ y \mid x, y \in X\},$$

and this is not equal to X – in general, $X \,{}_\lambda\!+ X$ will be larger than X. To remedy this, we proceed as follows.

Definition 4. *Let P be a probabilistic algebra, and let $X \subseteq P$. We define*

$$\langle X \rangle = \{x \,{}_\lambda\!+ y \mid x, y \in X \ \wedge \ 0 \leq \lambda \leq 1\}.$$

We say that X is pconvex *if $X = \langle X \rangle$, and we let $\mathcal{P}_{LP}(P) = \{X \in \mathcal{P}_L(P) \mid X = \langle X \rangle\}$, $\mathcal{P}_{UP}(P) = \{X \in \mathcal{P}_U(P) \mid X = \langle X \rangle\}$, and $\mathcal{P}_{CP}(P) = \{X \in \mathcal{P}_C(P) \mid X = \langle X \rangle\}$. We call these* nondeterministic probability domains*, and we denote by PCOH the category of coherent probabilistic algebras and continuous maps preserving probabilistic choice.*

Theorem 2. *Let P be a probabilistic algebra which is also a coherent domain. Then*

1. $\kappa_L : (\mathcal{P}_L(P), \subseteq) \to (\mathcal{P}_{LP}(P), \subseteq)$ *given by* $\kappa_L(X) = \downarrow\langle X\rangle$ *is a (continuous) closure operator.*
2. $\kappa_U : (\mathcal{P}_U(P), \supseteq) \to (\mathcal{P}_{UP}(P), \supseteq)$ *given by* $\kappa_U(X) = \uparrow\langle X\rangle$ *is a continuous kernel operator.*
3. $\kappa_C : (\mathcal{P}_C(P), \sqsubseteq) \to (\mathcal{P}_{CP}(P), \sqsubseteq)$ *given by* $\kappa_C(X) = \kappa_L(X) \cap \kappa_U(X)$ *is a continuous idempotent operator.*

Furthermore, $\mathcal{P}_{LP}(P)$ and $\mathcal{P}_{UP}(P)$ are bounded complete domains, and $\mathcal{P}_{CP}(P)$ is a coherent domain, and all three are probabilistic algebras. Finally, the first two extend to continuous functors $\mathcal{P}_{LP}, \mathcal{P}_{UP} : \mathsf{PCOH} \to \mathsf{BDC}$, and $\mathcal{P}_{CP}$ extends to a continuous functor $\mathcal{P}_{CP} : \mathsf{PCOH} \to \mathsf{PCOH}$.

Proof. Because P is coherent and the mapping $(\lambda, x, y) \mapsto x \mathbin{_\lambda\!+\!} y : [0,1] \times P \times P \to P$ is continuous, it follows routinely that $\downarrow\langle X\rangle$ is Scott closed if X is, and that $\uparrow\langle X\rangle$ is Scott compact if X is. This implies that

- $\downarrow\langle X\rangle = \bigcap\{Y \in \mathcal{P}_L(P) \mid X \subseteq Y\}$ for all $X \in \mathcal{P}_L(P)$,
- $\uparrow\langle X\rangle = \bigcap\{Y \in \mathcal{P}_U(P) \mid X \subseteq Y\}$ for all $X \in \mathcal{P}_U(P)$, and
- $\downarrow\langle X\rangle \cap \uparrow\langle X\rangle = \bigcap\{Y \in \mathcal{P}_C(P) \mid X \subseteq Y\}$ for all $X \in \mathcal{P}_C(P)$.

The first result just listed implies that $\downarrow X$ is pconvex if X is, and so $\downarrow\langle X\rangle$ is both Scott closed and pconvex. Moreover, it is clear that $X \subseteq \kappa_L(X) = \kappa_L(X)^2$, so that $\kappa_L : \mathcal{P}_L(P) \to \mathcal{P}_L(P)$ is a closure operator. It is routine to show that $\mathcal{P}_{LP}(P)$ is closed in $\mathcal{P}_L(P)$ under directed suprema, so that κ_L is continuous, from which it follows that $\mathcal{P}_{LP}(P)$ is a continuous domain.

Similarly, $\kappa_U(X) = \kappa_U(X)^2$ is the smallest Scott compact upper set containing X, for each $X \in \mathcal{P}_U(P)$. Since $X \subseteq \kappa_U(X)$, we conclude that κ_U is a kernel operator. It is easy to show that $\mathcal{P}_{UP}(P)$ is closed in $\mathcal{P}_U(P)$ under all filtered intersections, and that κ_U is continuous with respect to $\supseteq$. It follows that $\kappa_U : \mathcal{P}_U(P) \to \mathcal{P}_{UP}(P)$ is a continuous kernel operator and then that $\mathcal{P}_{UP}(P)$ is continuous as well.

Finally, using the coherence of P, it is straightforward to show that $\{X \in \mathcal{P}_C(P) \mid X = \langle X\rangle\}$ is closed under directed sups in the Egli-Milner order. In fact, the coherence of P implies that the mapping

$$X \mapsto (\downarrow X, \uparrow X) : \mathcal{P}_{CP}(P) \to \{(X, Y) \in \mathcal{P}_{LP}(P) \times \mathcal{P}_{UP}(P) \mid X \cap Y \neq \emptyset\}$$

is an order isomorphism, whose inverse is $(X, Y) \mapsto X \cap Y$. This implies that $\kappa_C(X) = \kappa_L(\downarrow X) \cap \kappa_U(\uparrow X)$ is continuous, since it is the composition of continuous mappings. This shows $\mathcal{P}_{CP}(P)$ is the image of a continuous selection-retraction pair (cf. [1]), and so $\mathcal{P}_{CP}(P)$ is continuous.

The probabilistic choice operators on each of these domains can be defined by $X \mathbin{_\lambda\!+\!} Y = \{x \mathbin{_\lambda\!+\!} y \mid x \in X, y \in Y\}$, and it follows from the rectangle law (cf. [7]) and the coherence of P that $X \mathbin{_\lambda\!+\!} Y$ is again in the appropriate nondeterministic probabilistic domain. One can argue that each such domain with these operations

satisfies the laws of Mean Values (cf. [7]), which are equivalent to the probabilistic algebra laws of [9]. Since the operations are easily seen to be continuous, it follows that each domain is a probabilistic algebra.

Finally, to show that each of these constructs leads to a continuous functor, it is sufficient to show that each is locally continuous (cf. [1]). For example, in the case of $\mathcal{P}_{LP}$, we need to show that, for P, Q coherent domains which are probabilistic algebras, the mapping $f \mapsto \mathcal{P}_{LP}(f) : \mathcal{P}_{LP}(P) \to \mathcal{P}_{LP}(Q)$ is continuous. If we restrict attention to PCOH, then this is just the restriction of the mapping $\widehat{f} : \Gamma(P) \to \Gamma(Q)$ by $\widehat{f}(X) = \downarrow f(X)$. □

We might use the domain $\mathcal{P}_{UP}(\mathcal{P}_{Pr}(\mathbb{FD}))$ to provide a model for probabilistic CSP, because the upper power domain is the power domain of demonic choice, which reflects how internal nondeterminism is modeled in CSP. The following reveals some of the structure of this object.

Theorem 3.

1. *If P is any bounded complete, continuous domain, then there is an e-p pair from P to $\mathcal{P}_{Pr}(P)$.*
2. *If P is a coherent domain that also is a probabilistic algebra, then the there is an injection of P into $\mathcal{P}_{UP}(P)$ that is a morphism of probabilistic algebras.*
3. *If P is a bounded complete continuous domain, then there is an e-p pair from $\mathcal{P}_{Pr}(P)$ to $\mathcal{P}_{UP}(\mathcal{P}_{Pr}(P))$.*

Proof. Since $\mathcal{P}_{Pr}$ is a left adjoint, we can use the unit of the adjunction for the embedding. This is simply the mapping $x \mapsto \delta_x$, which assigns the point mass at x to each point $x \in P$. For the projection mapping, we use the support function: $\mu \mapsto \operatorname{supp} \mu$. For simple measures $\Sigma_{x \in F} r_x \delta_x$, this is simply F. Since each measure is the directed supremum of simple measures, for general μ we can form the "limit" of the family F_i, where $\mu = \sqcup_i \Sigma_{x \in F_i} r_x \delta_x$. The projection mapping then send μ to $\wedge \operatorname{supp} \mu$, for which it is routine to verify the required equations for an e-p pair.

For the second claim, we note that $x \mapsto \uparrow x : P \to \mathcal{P}_{UP}(P)$ is a morphism of probabilistic algebras by the definition of the operations on $\mathcal{P}_{UP}(P)$.

Finally, if P is bounded complete, we can derive an e-p pair from $\mathcal{P}_{Pr}(P)$ to $\mathcal{P}_{UP} \circ \mathcal{P}_{Pr}(P)$, whose embedding is the composition of the units: $x \mapsto \uparrow \delta_x$, and whose projection is $X \mapsto \wedge \{\operatorname{supp} \mu \mid \mu \in X\}$. It is once again routine to validate the required equations for an e-p pair. □

4 Two Potential Applications:

Probabilistic CSP: We have motivated our work by considering probabilistic CSP and the fact that internal nondeterminism is not idempotent on PCSP. In fact, Morgan, et al outline the approach we have taken in [17], where they call the nondeterministic operation *indifferent nondeterminism*, because it is indifferent to how probabilistic choices are resolved. It would now be natural to develop a

model for CSP that uses this construction, and that also includes the other main operators for CSP in a way that the usual laws of CSP are satisfied. However, this is not such a simple task. For example, the external choice operator $\Box$ is idempotent in CSP, but one can argue that it should *not* be idempotent in a model also supporting probabilistic choice.

Indeed, consider the process $(p \,{}_{.5}\!+\! q)\Box(p' \,{}_{1/3}\!+\! q')$. In order for $\Box$ to be external nondeterminism, the environment should be able to select from the available actions. But this cannot be determined until the probabilistic choices are resolved – ie, ${}_{.5}\!+$ and ${}_{1/3}\!+$ must be resolved *before* $\Box$ for this to make computational sense. It is not hard to show that this implies that $\Box$ cannot be idempotent in the model, which means it must satisfy some other laws. Most appealing is that $\Box$ distribute through the probabilistic choice operators, as in PCSP, since, as we have seen, this implies $\Box$ is not idempotent.

Going beyond this discussion, one can also ask what other laws should hold in a model for CSP that also supports probabilistic choice operators. One law we believe should hold is the familiar law that $\Box$ reverts to $\sqcap$ if both branches begin with the same action: $(a \to p)\Box(a \to q) = a \to (p \sqcap q)$. Unfortunately, it is not clear how to build a model for CSP in which this law holds, in which $\Box$ distributes through the probabilistic operators, and in which $\sqcap$ is idempotent. This is the focus of ongoing, collaborative research with Gavin Lowe.

Security and information flow: The area which motivated the work reported in this paper has to do with security and information flow. Imagine a system in which there are users of varying security levels, and in which it is required that information about what the High level users are doing is not supposed to flow to the Low level users in the system. A particular concern in this setting is the potential for a covert channel by which information could pass from High to Low. An approach to modeling this situation generally assumes that all participants – both High level users and Low level users, know the process that represents what the system S may do, and it is also assumed that the system can be modeled as $P = H_{High} \|_{High \cup Low} S_{High \cup Low} \|_{Low} L$, where H represents High and L represents Low. The intention is that H interacts using High's alphabet of actions, and L interacts using Low's alphabet of actions, and the system S is required to interact on both alphabets.

Results in this area have been developed by Roscoe and his colleagues [19], and a brief summary of the results can be stated as follows. One way to ensure that no information flows from High to Low is to be sure that the system always looks deterministic to Low, no matter what High does. One way to express this is that, after two traces t and t' which have the property that $t|_{Low} = t'|_{Low}$, then $(P/t) \setminus H = (P/t') \setminus H$. That is, what Low sees (H's actions are hidden from view) is the same after two traces which have the same Low events in them. Unfortunately, this is not quite right, since hiding H's actions allows H to block actions, thus passing information to Low. So Roscoe generalizes this by obfuscating High's actions appropriately. And, the result he obtains is that, if such a system looks deterministic to Low, then it is secure.

This approach has some shortcomings. Here are some examples:

1. Consider $S = (h_1 \to l_1 \to S)\Box(h_2 \to l_2 \to S)$. This is obviously insecure, since L will know which action H has done according to which he can do.
2. On the other hand, $S = (h_1 \to (l_1 \sqcap l_2) \to S)\Box(h_2 \to (l_1 \sqcap l_2) \to S)$ avoids the problem in the first example, but it is still insecure, since L can deduce that H has executed some action if he can do an action.
3. Finally, the process $S = ((h_1 \to (l_1 \sqcap l_2) \to S)\Box(h_2 \to (l_1 \sqcap l_2) \to S)) \sqcap ((l_1 \sqcap l_2) \to S)$ avoids the problems of both examples. While this process should be considered secure, it falls outside the scope of Roscoe's analysis because L's view is $l_1 \sqcap l_2$, which is nondeterministic.

From these examples it should be clear that there is considerable latitude for secure processes S for which Low's view is nondeterministic. But the *refinement paradox*, under which a process that is secure can have a more deterministic refinement which is insecure means that security can be quite difficult to reason about in the presence of nondeterminism.

One motivation for the present work is to devise a model for CSP which also supports probabilistic choice operators. Such a model could then be a setting to test processes such as S above to reveal insecure behavior. The idea would be to replace the nondeterminism in S with probabilistic choice, and to see if the resulting system passed the test for secure behavior. For example, the last example above could be rewritten

$$S = ((h_1 \to (l_1 \,{}_{.5}{+}\, l_2) \to S)\Box(h_2 \to (l_1 \,{}_{.5}{+}\, l_2) \to S)) \sqcap ((l_1 \,{}_{.5}{+}\, l_2) \to S),$$

which now is deterministic (because the probabilistic choice of deterministic processes is deterministic), and so security is not compromised. Since using any probabilities in this process results in a secure one, we would be safe in deducing that the associated process

$$S = ((h_1 \to (l_1 \sqcap l_2) \to S)\Box(h_2 \to (l_1 \sqcap l_2) \to S)) \sqcap ((l_1 \sqcap l_2) \to S)$$

also is secure, even though it is nondeterministic from Low's viewpoint.

5 Summary

The approach we have taken essentially builds on an idea proposed in [17] to use the pconvex subsets of a power domain to model probabilistic choice and nondeterminism simultaneously. This approach actually provides some interesting new models in which all the laws one might expect hold. But there is much left to be done here. In particular, using these models for a fully abstract denotational semantics of a language for which there is an operational semantics is an obvious goal. In addition, finding a model for CSP that supports all the operators would shed light on the model of PCSP from [17]. And, similarly, such a model could be used for reasoning about the security concerns we just described.

References

1. S. Abramsky and A. Jung, *Domain Theory*, In: S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic and Computer Science,* **3**, Clarendon Press (1994), pp. 1–168. 353, 354, 359, 360, 361
2. P. America and J. J. R. R. Rutten, *Solving reflexive domains equations in a category of complete metric spaces, Journal of Computer Systems and Sciences* **39** (1989), pp. 343–375. 350
3. S. D. Brookes and A. W. Roscoe, *An improved failures model for communicating processes*, *Lecture Notes in Computer Science* **197** (1985), pp. 281 - 305.
4. E. P. de Vink and J. J. M. M. Rutten, *Bisimulation for probabilistic transition systems: a coalgebraic approach,* CWI preprint, October, 1998.
5. H. Hansson and B. Jonsson, *A calculus for communicating systems with time and probability, Proceedings* of the 11th Symposium on Real Time Systems, 1990. 351
6. J. I. den Hartog and E. P. de Vink, *Mixing up nondeterminism and probability: A preliminary report*, *Electronic Notes in Theoretical Computer Science* **22** (1997), URL: http://www.elsevier.nl/locate/entcs/volume22.html. 352
7. R. Heckmann, *Probabilistic domains, Proceedings* of CAAP '94, *Lecture Notes in Computer Science* **787** (1994), pp. 142–156. 360, 361
8. M. Hennessy and G. Plotkin, *Full abstraction for a simple parallel programming language,* Lecture Notes in Computer Science **74** (1979), Springer-Verlag. 350, 354
9. C. Jones, "Probabilistic Non-determinism," PhD Thesis, University of Edinburgh, 1990. Also published as Technical Report No. CST-63-90. 350, 351, 352, 355, 356, 359, 361
10. C. Jones and G. Plotkin, *A probabilistic powerdomain of evaluations, Proceedings* of 1989 Symposium on Logic in Computer Science, IEEE Computer Society Press, 1989, pp. 186–195. 352, 355
11. A. Jung, *Lawson-compactness for the probabilistic powerdomain,* Preprint, 1997. 359
12. A. Jung and R. Tix, *The troublesome probabilistic powerdomain, Electronic Notes in Theoretical Computer Science* **13** (1998), URL: http://www.elsevier.nl/locate/entcs/volume13.html. 355
13. J. D. Lawson, *Valuations on continuous lattices,* In: Rudolf-Eberhard Hoffmann, editor, *Continuous Lattices and Related Topics, Mathematik Arbeitspapiere* **27** (1982), Universität Bremen, pp. 204–225. 355
14. K. Larsen and A. Skou, *Bisimulation through probabilistic testing, Information and Computation* **94** (1991), pp. 456–471. 351
15. G. Lowe, "Probabilities and Priorities in Timed CSP," DPhil Thesis, University of Oxford, 1991. 351
16. N. Lynch and R. Segala, *Probabilistic simulations for probabilistic processes,Proceedings* of CONCUR'94, *Lecture Notes in Computer Science* **836** (1994), pp. 481–496. 351
17. C. Morgan, A. McIver, K. Seidel and J. Sanders, *Refinement-oriented probability for CSP,* University of Oxford Technical Report, 1994. 351, 352, 353, 356, 357, 358, 361, 363
18. C. Morgan, A. McIver, K. Seidel and J. Sanders, *Argument duplication in probabilistic CSP,* University of Oxford Technical Report, 1995. 351, 358
19. A. W. Roscoe. CSP and determinism in security modeling. In *IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 1995. 362

20. J. J. M. M. Rutten and D. Turi, *On the foundations of final semantics: Non-standard sets, metric spaces and partial orders, Proceedings* of the REX'92 Workshop, *Lecture Notes in Computer Science* **666** (1993), pp. 477–530. 351
21. N. Saheb-Djahromi, *CPOs of measures for nondeterminism, Theoretical Computer Science* **12** (1980), pp. 19–37. 350, 352, 355

Secrecy and Group Creation

Luca Cardelli[1], Giorgio Ghelli[2], and Andrew D. Gordon[1]

[1] Microsoft Research
[2] Pisa University

Abstract. We add an operation of group creation to the typed π-calculus, where a group is a type for channels. Creation of fresh groups has the effect of statically preventing certain communications, and can block the accidental or malicious leakage of secrets. Intuitively, no channel belonging to a fresh group can be received by processes outside the initial scope of the group, even if those processes are untyped. We formalize this intuition by adapting a notion of secrecy introduced by Abadi, and proving a preservation of secrecy property.

1 Introduction

Group creation is a natural extension of the sort-based type systems developed for the π-calculus. However, group creation has an interesting and subtle connection with secrecy. We start from the untyped π-calculus, where an operation to create fresh communication channels can be interpreted as creating fresh secrets. Under this interpretation, though, secrets can be leaked. We then introduce the notion of groups, which are types for channels, together with an operation for creating fresh groups. We explain how a fresh secret belonging to a fresh group can never be communicated to anybody who does not know the group in the first place. In other words, our type system prevents secrets from being leaked. Crucially, groups are not values, and cannot be communicated; otherwise, this secrecy property would fail.

1.1 Leaking Secrets

Consider the following configuration, where P is a private subsystem (a player) running in parallel with a potentially hostile adversary O (an opponent).

$$O \mid P$$

Suppose that the player P wants to create a fresh secret x. For example, x could be a private communication channel to be used only between subsystems of P. In the π-calculus this can be done by letting P evolve into a configuration $(\nu x)P'$, which means: create a new channel x to be used in the scope of P'.

$$O \mid (\nu x)P'$$

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 365–379, 2000.

The channel x is intended to remain private to P'. This privacy policy is going to be violated if the system then evolves into a situation such as the following, where p is a public channel known to the opponent ($p(y)$ is input of y on p, and $\overline{p}\langle x\rangle$ is output of x on p):

$$p(y).O' \mid (\nu x)(\overline{p}\langle x\rangle \mid P)$$

In this situation, the name x is about to be sent by the player over the public channel p and received by the opponent. In order for this communication to happen, the rules of the π-calculus, described in Section 2, require first an enlargement (extrusion) of the scope of x (otherwise x would escape its lexical scope). We assume that x is different from p, y, and any other name in O', so that the enlargement of the scope of x does not cause name conflicts. After extrusion, we have:

$$(\nu x)(p(y).O' \mid \overline{p}\langle x\rangle \mid P)$$

Now, x can be communicated over p into the variable y, while keeping x entirely within the scope of (νx). This results in:

$$(\nu x)(O'\{y{\leftarrow}x\} \mid P)$$

where the opponent has acquired the secret.

1.2 Preventing Leakage

The private name x has been leaked to the opponent by a combination of two mechanisms: the output instruction $\overline{p}\langle x\rangle$, and the extrusion of (νx). Can we prevent this kind of leakage of information? We have to consider that such a leakage may arise simply because of a mistake in the code of the player P, or because P decides to violate the privacy policy of x, or because a subsystem of P acts as a spy for the opponent.

It seems that we need to restrict either communication or extrusion. Since names are dynamic data in the π-calculus, it is not easy to say that a situation such as $\overline{p}\langle x\rangle$ (sending x on a channel known to the opponent) should not arise, because p may be dynamically obtained from some other channel, and may not occur at all in the code of P.

The other possibility is to try to prevent extrusion, which is a necessary step when leaking names outside their initial scope. However, extrusion is a fundamental mechanism in the π-calculus: blocking it completely would also block innocent communications over p. In general, attempts to limit extrusion are problematic, unless we abandon the notion of "fresh channel" altogether.

A natural question is whether one could somehow declare x to be private, and have this assertion statically checked so that the privacy policy of x cannot be violated. To this end, we may consider typed versions of the π-calculus. In these systems, we can classify channels into different groups (usually called sorts in the literature). We could have a group G for our private channels and write $(\nu x{:}G)P'$ to declare x to be of sort G. Unfortunately, in standard π-calculus type

systems all the groups are global, so the opponent could very well mention G in an input instruction. Global groups do not offer any protection, because leakage to the opponent can be made to type-check:

$$p(y{:}G).O' \mid (\nu x{:}G)(\overline{p}\langle x\rangle \mid P'')$$

In order to guarantee secrecy, we would want the group G itself to be secret, so that no opponent can input names of group G, and that no part of the player can output G information on public channels. A first idea is to partition groups into public ones and secret ones, with the static constraints that members of secret groups cannot be communicated over channels of public groups [7]. But this would work only for systems made of two (or a fixed number of) distrustful components; we aim to find a more general solution.

1.3 Group Creation

In general, we want the ability to create fresh groups on demand, and then to create fresh elements of those groups. To this end, we extend the π-calculus with an operator, $(\nu G)P$, to dynamically create a new group G in a scope P. This is a dynamic operator beacause, for example, it can be used to create a fresh group after an input:

$$q(y{:}T).(\nu G)P$$

Although group creation is dynamic, the group information can be tracked statically to ensure that names of different groups are not confused. Moreover, dynamic group creation can be very useful: we can dynamically spawn subsystems that have their own pool of shared resources that cannot interfere with other subsystems (compare with applet sandboxing).

Our troublesome example can now be represented as follows, where G is a new group, $G[\,]$ is the type of channels of group G, and a fresh x is declared to be a channel of group G (the type structure will be explained in more detail later):

$$p(y{:}T).O' \mid (\nu G)(\nu x{:}G[\,])\overline{p}\langle x\rangle$$

Here an attempt is made again to send the channel x over the public channel p. Fortunately, this process cannot be typed: the type T would have to mention G, in order to receive a channel of group G, but this is impossible because G is not known in the global scope where p would have to have been declared. The construct (νG) has extrusion properties similar to (νx), which are needed to permit legal communications over channels unrelated to G channels, but these extrusion rules prevent G from being confused with any group mentioned in T.

1.4 Untyped Opponents

Let us now consider the case where the opponent is untyped or, equivalently, not well-typed. This is intended to cover the situation where an opponent can execute any instruction available in the computational model without being restricted

by static checks such as type-checking or bytecode verification. For example, the opponent could be running on a separate, untrusted, machine.

We first make explicit the type declaration of the public channel, $p{:}U$, which has so far been omitted. The public channel must have a proper type, because that type is used in checking the type correctness of the player, at least. This type declaration could take the form of a channel declaration $(\nu p{:}U)$ whose scope encloses both the player and the opponent, or it could be part of some declaration environment shared by the player and the opponent and provided by a third entity in the system (for example, a name server).

Moreover, we remove the typing information from the code of the opponent, since an opponent does not necessarily play by the rules. The opponent now attempts to read any message transmitted over the public channel, no matter what its type is.

$$(\nu p{:}U)(p(y).O' \mid (\nu G)(\nu x{:}G[\,])\overline{p}\langle x\rangle)$$

Will an untyped opponent, by cheating on the type of the public channel, be able to acquire secret information? Fortunately, the answer is still no. The fact that the player is well-typed is sufficient to ensure secrecy, even in the presence of untyped opponents. This is because, in order for the player to leak information over a public channel p, the output operation $\overline{p}\langle x\rangle$ must be well-typed. The name x can be communicated only on channels whose type mentions G. So the output $\overline{p}\langle x\rangle$ cannot be well-typed, because then the type U of p would have to mention the group G, but U is not in the scope of G.

The final option to consider is whether one can trust the source of the declaration $p{:}U$. This declaration could come from a trusted source distinct from the opponent, but in general one has to mistrust this information as well. In any case, we can assume that the player will be type-checked with respect to this questionable information, $p{:}U$, within a trusted context. Even if U tries to cheat by mentioning G, the typing rules will not confuse that G with the one occurring in the player as (νG), and the output operation $\overline{p}\langle x\rangle$ will still fail to type-check. The only important requirement is that the player must be type-checked with respect to a global environment within a trusted context, which seems reasonable. This is all our secrecy theorem (Section 3) needs to assume.

1.5 Secrecy

We have thus established, informally, that a player creating a fresh group G can never communicate channels of group G to an opponent outside the initial scope of G, either because a (well-typed) opponent cannot name G to receive the message, or, in any case, because a well-typed player cannot use public channels to communicate G information to an (untyped) opponent.

Channels of group G are forever secret outside the initial scope of (νG).

So, secrecy is reduced in a certain sense to scoping and typing restrictions. But the situation is fairly subtle because of the extrusion rules associated with scoping, the fact that scoping restrictions in the ordinary π-calculus do not

prevent leakage, and the possibility of untyped opponents. As we have seen, the scope of channels can be extruded too far, perhaps inadvertently, and cause leakage, while the scope of groups offers protection against accidental or malicious leakage, even though it can be extruded as well.

We organise the remainder of the paper as follows. Section 2 defines the syntax, reduction semantics, and type system of our typed π-calculus with groups. In Section 3 we adapt a notion of secrecy due to Abadi to the untyped π-calculus. We also state the main technical result of the paper, Theorem 1, that a well-typed process preserves the secrecy of a fresh name of a fresh group, even from an untyped opponent. We outline the proof of Theorem 1 in Section 4; the main idea of the proof is to separate trusted data (from the typed process) and untrusted data (from the untyped opponent) using an auxiliary type system defined on untyped processes. Finally, Section 5 concludes.

2 A Typed π-Calculus with Groups

We present here a typed π-calculus with groups and group creation. The earliest type system for the π-calculus, reported in Milner's book [10] but first published in 1991, is based on sorts; sorts are like groups in that each name belongs to a sort, but Milner's system has no construct for sort creation. Moreover, his system allows recursive definitions of sorts; we would need to add recursive types to our system to mimic such definitions. Subsequent type systems introduced a variety of channel type constructors and subtyping [11,12].

2.1 Syntax and Operational Semantics

Types specify, for each channel, its group and the type of the values that can be exchanged on that channel.

Types:

$T ::= G[T_1, \ldots, T_n]$	polyadic channel in group G

We study an asynchronous, choice-free, polyadic typed π-calculus. The calculus is defined as follows. We identify processes up to capture-avoiding renaming of bound variables; we write $P = Q$ to mean that P and Q are the same up to capture-avoiding renaming of bound variables.

Expressions and Processes:

x, y, p, q	names, variables
$P, Q, R ::=$	process
$x(y_1{:}T_1, \ldots, y_k{:}T_k).P$	polyadic input
$\overline{x}\langle y_1, \ldots, y_k\rangle$	polyadic output
$(\nu G)P$	group creation
$(\nu x{:}T)P$	restriction

$P \mid Q$	composition
$!P$	replication
$\mathbf{0}$	inactivity

In a restriction, $(\nu x{:}T)P$, the name x is bound in P, and in an input, $x(y_1{:}T_1, \ldots, y_k{:}T_k).P$, the names $y_1, \ldots, y_k$ are bound in P. In a group creation $(\nu G)P$, the group G is bound with scope P. Let $fn(P)$ be the set the names free in a process P, and let $fg(P)$ and $fg(T)$ be the sets of groups free in a process P and a type T, respectively.

In the next two tables, we define a reduction relation, $P \to Q$, in terms of an auxiliary notion of structural congruence, $P \equiv Q$. Structural congruence allows a process to be re-arranged so that reduction rules may be applied. Each reduction derives from an exchange of a tuple on a named communication channel.

Our rules for reduction and structural congruence are standard [10] apart from the inclusion of new rules for group creation, and the exclusion of garbage collection rules such as $\mathbf{0} \equiv (\nu x{:}T)\mathbf{0}$ and $x \notin fn(P) \Rightarrow (\nu x{:}T)P \equiv P$. Such rules are unnecessary for calculating reduction steps. In their presence, reduction can enlarge the set of free groups of a process. Hence, their inclusion would slightly complicate the statement of subject reduction.

Structural Congruence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu x{:}T)P \equiv (\nu x{:}T)Q$	(Struct Res)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct GRes)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow x(y_1{:}T_1, \ldots, y_n{:}T_n).P \equiv x(y_1{:}T_1, \ldots, y_n{:}T_n).Q$	(Struct Input)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$x_1 \neq x_2 \Rightarrow (\nu x_1{:}T_1)(\nu x_2{:}T_2)P \equiv (\nu x_2{:}T_2)(\nu x_1{:}T_1)P$	(Struct Res Res)
$x \notin fn(P) \Rightarrow (\nu x{:}T)(P \mid Q) \equiv P \mid (\nu x{:}T)Q$	(Struct Res Par)
$(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P$	(Struct GRes GRes)
$G \notin fg(T) \Rightarrow (\nu G)(\nu x{:}T)P \equiv (\nu x{:}T)(\nu G)P$	(Struct GRes Res)
$G \notin fg(P) \Rightarrow (\nu G)(P \mid Q) \equiv P \mid (\nu G)Q$	(Struct GRes Par)

Reduction: $P \to Q$

$\overline{x}\langle y_1, \ldots, y_n\rangle \mid x(z_1{:}T_1, \ldots, z_n{:}T_n).P \to P\{z_1 \leftarrow y_1\} \cdots \{z_n \leftarrow y_n\}$	(Red I/O)
$P \to Q \Rightarrow P \mid R \to Q \mid R$	(Red Par)

$$P \to Q \Rightarrow (\nu G)P \to (\nu G)Q \qquad \text{(Red GRes)}$$
$$P \to Q \Rightarrow (\nu x{:}T)P \to (\nu x{:}T)Q \qquad \text{(Red Res)}$$
$$P' \equiv P, P \to Q, Q \equiv Q' \Rightarrow P' \to Q' \qquad \text{(Red } \equiv\text{)}$$

The new rules for group creation are the congruence rules (Struct GRes) and (Red GRes), and the scope mobility rules (Struct GRes GRes), (Struct GRes Res), and (Struct GRes Par). The latter rules are akin to the standard scope mobility rules for restriction (Struct Res Res) and (Struct Res Par).

2.2 The Type System

Environments declare the names and groups in scope during type-checking; we define environments, E, by $E ::= \varnothing \mid E, G \mid E, x{:}T$. We define $dom(E)$ by $dom(\varnothing) = \varnothing$, $dom(E, G) = dom(E) \cup \{G\}$, and $dom(E, x{:}T) = dom(E) \cup \{x\}$.

We define four typing judgments: first, $E \vdash \diamond$ means that E is well-formed; second, $E \vdash T$ means that T is well-formed in E; third, $E \vdash x : T$ means that $x{:}T$ is in E, and that E is well-formed; and, fourth, $E \vdash P$ means that P is well-formed in the environment E.

Typing Rules:

$$\text{(Env } \varnothing\text{)} \quad \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env } x\text{)} \quad \frac{E \vdash T \quad x \notin dom(E)}{E, x{:}T \vdash \diamond} \qquad \text{(Env } G\text{)} \quad \frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond}$$

$$\text{(Type Chan)} \quad \frac{G \in dom(E) \quad E \vdash T_1 \quad \cdots \quad E \vdash T_n}{E \vdash G[T_1, \ldots, T_n]} \qquad \text{(Exp } x\text{)} \quad \frac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}$$

$$\text{(Proc GRes)} \quad \frac{E, G \vdash P}{E \vdash (\nu G)P} \qquad \text{(Proc Res)} \quad \frac{E, x{:}T \vdash P}{E \vdash (\nu x{:}T)P} \qquad \text{(Proc Zero)} \quad \frac{E \vdash \diamond}{E \vdash \mathbf{0}}$$

$$\text{(Proc Par)} \quad \frac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q} \qquad \text{(Proc Repl)} \quad \frac{E \vdash P}{E \vdash !P}$$

$$\text{(Proc Input)} \quad \frac{E \vdash x : G[T_1, \ldots, T_n] \qquad E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P}{E \vdash x(y_1{:}T_1, \ldots, y_n{:}T_n).P}$$

$$\text{(Proc Output)} \quad \frac{E \vdash x : G[T_1, \ldots, T_n] \qquad E \vdash y_1 : T_1 \quad \cdots \quad E \vdash y_n : T_n}{E \vdash \overline{x}\langle y_1, \ldots, y_n \rangle}$$

The rules for good environments ensure that the names and groups declared in an environment are distinct, and that all the types mentioned in an environment are good. The rule for a good type ensures that all the groups free in a type are declared. The rule for a good name looks up the type of a name in the

environment. The rules (Proc Input) and (Proc Output) for well-typed processes ensure that names occurring in inputs and outputs are used according to their declared types. The rules (Proc GRes) and (Proc Res) allow fresh groups and names, respectively, to be used inside their scope but not outside. The other rules (Proc Zero), (Proc Par), and (Proc Repl) define a composite process to be well-typed provided its components, if any, are themselves well-typed.

2.3 Subject Reduction

Subject reduction is a property stating that well-typed processes reduce necessarily to well-typed processes, thus implying that "type errors" are not generated during reduction. As part of establishing this property, we need to establish a subject congruence property, stating that well-typing is preserved by congruence. Subject congruence is essential for a type system based on the π-calculus: two congruent processes are meant to represent the same computation so they should have the same typing properties.

As we shall see shortly, a consequence of our typing discipline is the ability to preserve secrets. In particular, the subject reduction property, together with the proper application of extrusion rules, has the effect of preventing certain communications that would leak secrets. For example, consider the discussion in Section 1.3, regarding a process of the form:

$$p(y{:}T).O' \mid (\nu G)(\nu x{:}G[\,])P$$

In order to communicate the name x (the secret) on the public channel p, we would need to reduce the initial process to a configuration containing the following:

$$p(y{:}T).O'' \mid \overline{p}\langle x\rangle$$

If subject reduction holds then this reduced term has to be well-typed, which is true only if $p : H[T]$ for some H, and $T = G[\,]$. However, in order to get to the point of bringing the input operation of the opponent next to an output operation of the player, we must have extruded the (νG) and $(\nu x{:}G[\,])$ binders outward. The rule (Struct GRes Par), used to extrude (νG) past $p(y{:}T).O''$, requires that $G \notin fg(T)$. This contradicts the requirement that $T = G[\,]$. If that extrusion were allowed, that is, if we failed to prevent name clashes on group names, then the player could communicate with the opponent in a well-typed way, and secrecy would fail.

Lemma 1 (Subject Congruence). *If $E \vdash P$ and $P \equiv Q$ then $E \vdash Q$.*

Proposition 1 (Subject Reduction). *If $E \vdash P$ and $P \to Q$ then $E \vdash Q$.*

Subject reduction allows us to prove secrecy properties like the following one.

Proposition 2. *Let the process $P = p(y{:}T).O' \mid (\nu G)(\nu x{:}G[T_1, \ldots, T_n])P'$. If $E \vdash P$, for some E, then no process deriving from P includes a communication of x along p. Formally, there are no processes P'' and P''' and a context $C[\,]$ such that $P \equiv (\nu G)(\nu x{:}G[T_1, \ldots, T_n])P''$, $P'' \to P'''$, $P''' \equiv C[\overline{p}\langle x\rangle]$, where p and x are not bound by $C[]$.*

Proof. Assume that P'' and P''' exist. Subject reduction implies the judgment $E, G, x{:}G[T_1, \ldots, T_n] \vdash P'''$, which implies that $E, G, x{:}G[T_1, \ldots, T_n], E' \vdash \overline{p}\langle x \rangle$ for some E'. Hence, p has a type $H[G[T_1, \ldots, T_n]]$. But this is impossible, since p is defined in E, hence out of the scope of G. □

In the following section we generalize this result, and extend it to a situation where the opponent is not necessarily well-typed.

3 Secrecy in the Context of an Untyped Opponent

We formalize the idea that in the process $(\nu G)(\nu x{:}G[T_1, \ldots, T_n])P$, the name x of the new group G is known only within P (the scope of G) and hence is kept secret from any opponent able to communicate with the process (whether or not the opponent respects our type system). We give a precise definition of when an untyped process $(\nu x)P$ preserves the secrecy of a restricted name x from an opponent (the external process with which it interacts). Then we show that the untyped process obtained by erasing type annotations and group restrictions from a well-typed process $(\nu G)(\nu x{:}G[T_1, \ldots, T_n])P$ preserves the secrecy of the name x.

3.1 Review: The Untyped π-Calculus

In this section, we describe the syntax and semantics of an untyped calculus that corresponds to the typed calculus of Section 2. The process syntax is the same as for the typed calculus, except that we drop type annotations and the new-group construct.

Processes:

x, y, p, q	names, variables
$P, Q, R ::=$	process
$x(y_1, \ldots, y_n).P$	polyadic input
$\overline{x}\langle y_1, \ldots, y_n \rangle$	polyadic output
$(\nu x)P$	restriction
$P \mid Q$	composition
$!P$	replication
$\mathbf{0}$	inactivity

As in the typed calculus, the names $y_1, \ldots, y_n$ are bound in an input $x(y_1, \ldots, y_n).P$ with scope P, and the name x is bound in $(\nu x)P$ with scope P. We identify processes up to capture-avoiding renaming of bound names. We let $fn(P)$ be the set of names free in P.

Every typed process has a corresponding untyped process obtained by erasing type annotations and group creation operators. We confer reduction, $P \rightarrow Q$, and structural congruence, $P \equiv Q$, relations on untyped processes corresponding to the typed reduction and structural congruence relations. We omit the standard

rules, which are obtained from the rules of the typed calculus by erasing type annotations and deleting rules that mention the new-group construct.

Erasures of type annotations and group restrictions:

$erase((\nu G)P) \triangleq erase(P)$ $\quad$ $erase((\nu x{:}T)P) \triangleq (\nu x)erase(P)$

$erase(\mathbf{0}) \triangleq \mathbf{0}$ $\quad$ $erase(P \mid Q) \triangleq erase(P) \mid erase(Q)$

$erase(!P) \triangleq !erase(P)$ $\quad$ $erase(\overline{x}\langle y_1, \ldots, y_n\rangle) \triangleq \overline{x}\langle y_1, \ldots, y_n\rangle$

$erase(x(y_1{:}T_1, \ldots, y_n{:}T_n).P) \triangleq x(y_1, \ldots, y_n).erase(P)$

Proposition 3 (Erasure). *For all typed processes P and Q, and untyped processes R, $P \to Q$ implies $erase(P) \to erase(Q)$ and $erase(P) \to R$ implies there is a typed process Q such that $P \to Q$ and $R \equiv erase(Q)$.*

Finally, we define input and output transitions to describe the interactions between an untyped process and an untyped opponent running alongside in parallel. An input transition $P \xrightarrow{x} (y_1, \ldots, y_n)Q$ means that P is ready to receive an input tuple on the channel x in the variables $y_1, \ldots, y_n$, and then continue as Q. The variables $y_1, \ldots, y_n$ are bound with scope Q. An output transition $P \xrightarrow{\overline{x}} (\nu z_1, \ldots, z_m)\langle y_1, \ldots, y_n\rangle Q$ means that P is ready to transmit an output tuple $\langle y_1, \ldots, y_n\rangle$ on the channel x, and then continue as Q. The set $\{z_1, \ldots, z_m\} \subseteq \{y_1, \ldots, y_n\}$ consists of freshly generated names whose scope includes both the tuple $\langle y_1, \ldots, y_n\rangle$ and the process Q. The names $z_1, \ldots, z_n$ are unknown to the opponent beforehand, but are revealed by the interaction.

Labelled transitions such as these are most commonly defined inductively by a structural operational semantics; for the sake of brevity, the following definitions are in terms of structural congruence.

- Let $P \xrightarrow{x} (y_1, \ldots, y_n)Q$ if and only if the names $y_1, \ldots, y_n$ are pairwise distinct, and there are processes P_1 and P_2 and pairwise distinct names $z_1, \ldots, z_m$ such that $P \equiv (\nu z_1, \ldots, z_m)(x(y_1, \ldots, y_n).P_1 \mid P_2)$ and $Q \equiv (\nu z_1, \ldots, z_m)(P_1 \mid P_2)$ where $x \notin \{z_1, \ldots, z_m\}$, and $\{y_1, \ldots, y_n\} \cap (\{z_1, \ldots, z_m\} \cup fn(P_2)) = \varnothing$.
- Let $P \xrightarrow{\overline{x}} (\nu z_1, \ldots, z_m)\langle y_1, \ldots, y_n\rangle Q$ if and only if the names $z_1, \ldots, z_m$ are pairwise distinct, and we have $P \equiv (\nu z_1, \ldots, z_m)(\overline{x}\langle y_1, \ldots, y_n\rangle \mid Q)$ where $x \notin \{z_1, \ldots, z_m\}$ and $\{z_1, \ldots, z_m\} \subseteq \{y_1, \ldots, y_n\}$.

3.2 A Secrecy Theorem

The following definition is inspired by Abadi's definition of secrecy [2] for the untyped spi calculus [3]. Abadi attributes the underlying idea to Dolev and Yao [8]: that a name is kept secret from an opponent if after no series of interactions is the name transmitted to the opponent. (In the presence of encryption, the definition is rather more subtle than this.) An alternative we do not pursue here is to formulate secrecy using testing equivalence [1,3].

We model the external opponent simply by the finite set of names S known to it. We inductively define a relation $(P_0, S_0)\ \mathcal{R}\ (P, S)$ to mean that starting from a process P_0 and an opponent knowing S_0, we may reach a state in which P_0 has evolved into P, and the opponent now knows S.

(1) $(P_0, S_0)\ \mathcal{R}\ (P_0, S_0)$
(2) If $(P_0, S_0)\ \mathcal{R}\ (P, S)$ and $P \to Q$ then $(P_0, S_0)\ \mathcal{R}\ (Q, S)$.
(3) If $(P_0, S_0)\ \mathcal{R}\ (P, S)$, $P \xrightarrow{x} (y_1, \ldots, y_n)Q$, $x \in S$, and $(\{z_1, \ldots, z_n\} - S) \cap fn(P_0) = \varnothing$ then $(P_0, S_0)\ \mathcal{R}\ (Q\{y_1 \leftarrow z_1, \ldots, y_n \leftarrow z_n\}, S \cup \{z_1, \ldots, z_n\})$.
(4) If $(P_0, S_0)\ \mathcal{R}\ (P, S)$, $P \xrightarrow{\overline{x}} (\nu z_1, \ldots, z_m)\langle y_1, \ldots, y_n\rangle Q$ and $x \in S$ and $\{z_1, \ldots, z_m\} \cap (S \cup fn(P_0)) = \varnothing$ then $(P_0, S_0)\ \mathcal{R}\ (Q, S \cup \{y_1, \ldots, y_n\})$.

Clause (1) says that (P_0, S_0) is reachable from itself. Clause (2) allows the process component to evolve on its own. Clause (3) allows the process to input the tuple $\langle z_1, \ldots, z_n\rangle$ from the opponent, provided the channel x is known to the opponent. The names $\{z_1, \ldots, z_n\} - S$ are freshly created by the opponent; the condition $(\{z_1, \ldots, z_n\} - S) \cap fn(P_0) = \varnothing$ ensures these fresh names are not confused with names initially known by P_0. Clause (4) allows the process to output the tuple $\langle y_1, \ldots, y_n\rangle$ to the opponent, who then knows the names $S \cup \{y_1, \ldots, y_n\}$, provided the channel x is known to the opponent. The names $\{z_1, \ldots, z_n\}$ (included in $\{y_1, \ldots, y_n\}$) are freshly created by the process; the condition $\{z_1, \ldots, z_m\} \cap (S \cup fn(P_0)) = \varnothing$ ensures these fresh names are not confused with names currently known by the opponent or initially known by P_0.

Next, we give definitions of when a name is revealed to an opponent, and formalize the secrecy property of group creation discussed in Section 1.

Revealing Names, Preserving their Secrecy:

Suppose S is a set of names and P is a process.
Then P *may reveal* x *to* S if and only if there are P' and S' such that $(P, S)\ \mathcal{R}\ (P', S')$ and $x \in S'$; otherwise, P *preserves the secrecy of* x *from* S.
Moreover, $(\nu x)P$ *may reveal the restricted name* x *to* S if and only if there is a name $y \notin S \cup fn(P)$ such that $P\{x \leftarrow y\}$ may reveal y to S;
otherwise $(\nu x)P$ *preserves the secrecy of the restricted name* x *from* S.

Theorem 1 (Secrecy). *Suppose that* $E \vdash (\nu G)(\nu x{:}T)P$ *where* $G \in fg(T)$. *Let* S *be the names occurring in* $dom(E)$. *Then the erasure* $(\nu x)erase(P)$ *of* $(\nu G)(\nu x{:}T)P$ *preserves the secrecy of the restricted name* x *from* S.

We sketch a proof in the next section. The group restriction (νG) is essential. A typing $E \vdash (\nu x{:}T)P$ does not in general imply that the erasure $(\nu x)erase(P)$ preserves the secrecy of the restricted name from a set S. For example, consider the typing $\varnothing, G, x{:}G[G[\,]] \vdash (\nu y{:}G[\,])\overline{x}\langle y\rangle$. Then the erasure $(\nu y)\overline{x}\langle y\rangle$ reveals the restricted name to any set S such that $x \in S$.

4 Proof of Secrecy

The proof of the secrecy theorem is based on an auxiliary type system that partitions channels into untrusted channels, with type Un, and trusted ones, with type $Ch[T_1, \ldots, T_n]$, where each T_i is either a trusted or untrusted type. The type system insists that names are bound to variables with the same trust level (that is, the same type), and that no trusted name is ever transmitted on an untrusted channel. Hence an opponent knowing only untrusted channel names will never receive any trusted name.

Types:

$T ::=$	channel type
$\quad Ch[T_1, \ldots, T_n]$	trusted polyadic channel
$\quad Un$	untrusted name

For any group G, we can translate group-based types into the auxiliary type system as follows: any type that does not contain G free becomes Un, while a type $H[T_1, \ldots, T_n]$ that contains G free is mapped onto $Ch[[\![T_1]\!]_G, \ldots, [\![T_n]\!]_G]$. This translation is proved to preserve typability. This implies that an opponent knowing only names whose type does not contain G free, will never be able to learn any name whose type contains G free. This is the key step in proving the secrecy theorem.

Next, we define the three judgments of the auxiliary type system: first, $E \vdash \diamond$ means that E is well-formed; second, $E \vdash x : T$ means that $x{:}T$ is in E, and that E is well-formed; and, third, $E \vdash P$ means that P is well-formed in the environment E.

Typing Rules:

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash \diamond \quad x \notin dom(E)}{E, x{:}T \vdash \diamond} \qquad \frac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}$$

$$\frac{E, x{:}T \vdash P}{E \vdash (\nu x)P} \qquad \frac{E \vdash \diamond}{E \vdash \mathbf{0}} \qquad \frac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q} \qquad \frac{E \vdash P}{E \vdash !P}$$

$$\frac{E \vdash x : Ch[T_1, \ldots, T_n] \qquad E, y_1{:}T_1, \ldots, y_n{:}T_n \vdash P}{E \vdash x(y_1, \ldots, y_n).P}$$

$$\frac{E \vdash x : Ch[T_1, \ldots, T_n] \qquad E \vdash y_1 : T_1 \quad \cdots \quad E \vdash y_n : T_n}{E \vdash \overline{x}\langle y_1, \ldots, y_n \rangle}$$

$$\frac{E \vdash x : Un \qquad E, y_1{:}Un, \ldots, y_n{:}Un \vdash P}{E \vdash x(y_1, \ldots, y_n).P}$$

$$\frac{E \vdash x : Un \qquad E \vdash y_1 : Un \quad \cdots \quad E \vdash y_n : Un}{E \vdash \overline{x}\langle y_1, \ldots, y_n \rangle}$$

The auxiliary type system is defined on untyped processes. Any untrusted opponent may be type-checked, as follows. This property makes the type system suitable for reasoning about processes containing both trusted and untrusted subprocesses.

Lemma 2. *For all P, if $fn(P) = \{x_1, \ldots, x_n\}$ then $\varnothing, x_1{:}Un, \ldots, x_n{:}Un \vdash P$.*

Structural congruence and reduction preserve typings.

Lemma 3. *If $E \vdash P$ and either $P \equiv Q$ or $P \to Q$ then $E \vdash Q$.*

The following fact is the crux of the proof of Theorem 1: an opponent who knows only untrusted names cannot learn any trusted one.

Proposition 4. *Suppose that $\varnothing, y_1{:}Un, \ldots, y_n{:}Un, x{:}T \vdash P$ where $T \neq Un$. Then the process P preserves the secrecy of the name x from $S = \{y_1, \ldots, y_n\}$.*

Next, we translate the types and environments of the π-calculus with groups into our auxiliary system, and state that erasure preserves typing.

Translations of types and environments:

$$[\![H[T_1, \ldots, T_n]]\!]_G \triangleq \begin{cases} Ch[[\![T_1]\!]_G, \ldots, [\![T_n]\!]_G] & \text{if } G \in fg(H[T_1, \ldots, T_n]) \\ Un & \text{otherwise} \end{cases}$$

$$[\![\varnothing]\!]_G \triangleq \varnothing \qquad [\![E, H]\!]_G \triangleq [\![E]\!]_G \qquad [\![E, x{:}T]\!]_G \triangleq [\![E]\!]_G, x{:}[\![T]\!]_G$$

Lemma 4. *If $E \vdash P$ then $[\![E]\!]_G \vdash erase(P)$.*

Finally, we outline the proof of Theorem 1.

Lemma 5. *If $E, x{:}T, E' \vdash P$ and $E \vdash y : T$ then $E, E' \vdash P\{x{\leftarrow}y\}$.*

Lemma 6. *Let S be a finite set of names and P a process. Then P preserves the secrecy of x from S if and only if for all P', S', $(P, S)\ \mathcal{R}\ (P', S')$ implies that $x \notin S'$. Moreover, $(\nu x)P$ preserves the secrecy of the restricted name x from S if and only if for all $y \notin fn(P) \cup S$, $P\{x{\leftarrow}y\}$ preserves the secrecy of y from S.*

Proof of Theorem 1 *Suppose that $E \vdash (\nu G)(\nu x{:}T)P$ where $G \in fg(T)$. Let S be the names occurring in $dom(E)$. Then the erasure $(\nu x)erase(P)$ of $(\nu G)(\nu x{:}T)P$ preserves the secrecy of the restricted name x from S.*

Proof. Since the name x is bound, we may assume that $x \notin S$. Consider any name $y \notin fn(P) \cup S$. By Lemma 6, it is sufficient to show that $erase(P)\{x{\leftarrow}y\}$ preserves the secrecy of y from S. Since $E \vdash (\nu G)(\nu x{:}T)P$ must have been derived using (Proc GRes) and (Proc Res), we have $E, G, x{:}T \vdash P$, with $G \notin dom(E)$. Hence, $[\![E]\!]_G = \varnothing, z_1{:}Un, \ldots, z_n{:}Un$ where $S = \{z_1, \ldots, z_n\}$. Lemma 4 implies that $\varnothing, z_1{:}Un, \ldots, z_n{:}Un, x{:}[\![T]\!]_G \vdash erase(P)$. Since $G \in fg(T)$, $[\![T]\!]_G \neq Un$. So Proposition 4 implies that $erase(P)$ preserves the secrecy of x from S. If $x = y$ we are done, since in that case $erase(P)\{x{\leftarrow}y\} = erase(P)$. Otherwise, suppose $x \neq y$. Since $y \notin \{z_1, \ldots, z_n\}$ we can derive $\varnothing, z_1{:}Un, \ldots, z_n{:}Un,$

$y{:}[\![T]\!]_G, x{:}[\![T]\!]_G \vdash \mathit{erase}(P)$ using a weakening lemma, and also derive $\varnothing, z_1{:}\mathit{Un}, \ldots, z_n{:}\mathit{Un}, y{:}[\![T]\!]_G \vdash y : [\![T]\!]_G$. By the substitution lemma, Lemma 5, these two judgments imply $\varnothing, z_1{:}\mathit{Un}, \ldots, z_n{:}\mathit{Un}, y{:}[\![T]\!]_G \vdash \mathit{erase}(P)\{x{\leftarrow}y\}$. Hence, Proposition 4 implies that $\mathit{erase}(P)\{x{\leftarrow}y\}$ preserves the secrecy of the name y from $S = \{z_1, \ldots, z_n\}$. □

5 Conclusion

We proposed a typed π-calculus in which each name belongs to a group, and in which groups may be created dynamically by a group creation operator. Typing rules bound the communication of names of dynamically created groups, hence preventing the accidental or malicious revelation of secrets. We explained these ideas informally, proposed a formalization based on Abadi's notion of name secrecy, and explained the ideas underlying the proof.

The idea of name groups and a group creation operator arose in our recent work on type systems for regulating mobile computation in the ambient calculus [5]. The new contributions of the present paper are to recast the idea in the simple setting of the π-calculus and to explain, formalize, and prove the secrecy properties induced by group creation. Another paper [6] extends the typed π-calculus of Section 2 with an effect system. That paper establishes a formal connection between group creation and the *letregion* construct of Tofte and Talpin's region-based memory management [15]. That paper generalizes our subject congruence, subject reduction, and erasure results (Lemma 1, Propositions 1 and 3) to the system of types and effects for the π-calculus. We conjecture that the main secrecy result of this paper, Theorem 1, would hold also for the extended system, but we have not studied the details.

The idea of proving a secrecy property for a type system by translation into a mixed trusted and untrusted type system appears to be new. Our work develops the idea of a type system for the π-calculus that mixes trusted and untrusted data, and the idea that every opponent should be typable in the sense of Lemma 2. These ideas first arose in Abadi's type system for the spi calculus [1]. In that system, each name belongs to a global security level, such as *Public* or *Secret*, but there is no level creation construct akin to group creation.

A related paper [4] presents a control flow analysis for the π-calculus that can also establish secrecy properties of names. There is an intriguing connection, that deserves further study, between the groups of our system, and the channels and binders of the flow analysis. One difference between the studies is that the flow analysis has no counterpart of the group creation operator of this paper. Another is that an algorithm is known for computing flow analyses for the π-calculus, but we have not investigated algorithmic aspects of our type system. It would be interesting to consider whether algorithms for Milner's sort systems [9,16] extend to our calculus.

Other related work on the π-calculus includes type systems for guaranteeing locality properties [13,14]. These systems can enforce by type-checking that a name cannot be leaked outside a particular locality.

In summary, group creation is a powerful new construct for process calculi. Its study is just beginning; we expect that its secrecy guarantees will help with the design and semantics of new programming language features, and with the analysis of security properties of individual programs.

Acknowledgements

We thank Rocco de Nicola, Roberto Gorrieri, Tony Hoare, and the anonymous referees for useful suggestions. Giorgio Ghelli was supported by Microsoft Research, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica", project DATA-X.

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999. 374, 378
2. M. Abadi. Security protocols and specifications. In *Proceedings FOSSACS'99*, volume 1578 of *LNCS*, pages 1–13. Springer, 1999. 374
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999. 374
4. C. Bodei, P. Degano, F. Nielson, and H. Nielson. Control flow analysis for the π-calculus. In *Proceedings Concur'98*, volume 1466 of *LNCS*, pages 84–98. Springer, 1998. 378
5. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Proceedings TCS2000*, LNCS. Springer, 2000. To appear. 378
6. S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. In *Proceedings MFCS2000*, LNCS. Springer, 2000. To appear. 378
7. D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976. 367
8. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IC–29(12):198–208, 1983. 374
9. S. J. Gay. A sort inference algorithm for the polyadic pi-calculus. In *Proceedings POPL'93*. ACM, 1993. 378
10. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999. 369, 370
11. M. Odersky. Polarized name passing. In *Proc. FST & TCS*, LNCS. Springer, December 1995. 369
12. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. 369
13. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings POPL'98*, pages 378–390. ACM, 1998. 378
14. P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings ICALP'98*, volume 1443 of *LNCS*, pages 695–706. Springer, 1998. 378
15. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 378
16. V. T. Vasconcelos and K. Honda. Principal typing-schemes in a polyadic π-calculus. In *Proceedings Concur'93*, volume 715 of *LNCS*, pages 524–538. Springer, 1993. 378

On the Reachability Problem in Cryptographic Protocols*

Roberto M. Amadio and Denis Lugiez

Université de Provence, Laboratoire d'Informatique de Marseille
{amadio,lugiez}@cmi.univ-mrs.fr

Abstract. We study the verification of secrecy and authenticity properties for cryptographic protocols which rely on symmetric shared keys. The verification can be reduced to check whether a certain parallel program which models the protocol and the specification can reach an erroneous state while interacting with an adversary. Assuming finite principals, we present a decision procedure for the reachability problem which is based on a 'symbolic' reduction system.

Keywords: cryptographic protocols, verification, symbolic computation.

1 Introduction

Cryptographic protocols seem good candidates for formal verification and several frameworks have been proposed for making possible formal and automatic analyses. In these approaches a 'perfect' encryption scheme is assumed: encryption is an injective function and the only way to decrypt an encrypted message is to know the key with which it was encrypted.

One class of approaches involves state exploration using model-checking techniques [Low96,CJM98,MMS97]. Lowe [Low96], Schneider [Sch96] and several others have used CSP to specify authentication protocols, analysing them with the FDR model-checking tool. Other state exploration approaches are based on logic programming techniques [Mea94]. The main benefit of these approaches is their automation and efficiency in uncovering subtle bugs in protocols (*e.g.,* Lowe's 'man-in-the-middle' attack on the Needham-Schroeder symmetric key protocol). Usually, these methods make simplifying hypotheses on the behaviour of the adversary which are used to bound the state space and thus allow for the application of traditional *finite state* model-checking techniques.

A second class of approaches relies on general-purpose proof assistant tools. Paulson [Pau97] uses induction on traces to formally prove protocol correctness using Isabelle. Bolignano [Bol96] uses a state-based analysis of the protocols, proving invariant properties, with the proofs subsequently mechanised in Coq. Although these approaches are not automatic, recent work (see, *e.g.*,

* A full version of this paper is available as INRIA Research Report 3915, March 2000. The first author is a member of Action INRIA 'MIMOSA' and he is partially supported by WG-CONFER and RNRT-Marvel.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 380–394, 2000.

[DLMS99,Wei99]) suggests that certain authentication protocols can be modelled in decidable fragments of first-order logic.

A more recent trend has been the use of name-passing process calculi for studying cryptographic authentication protocols. Abadi and Gordon have presented the *spi*-calculus [AG97], an extension of the π-calculus with cryptographic primitives. Principals of a protocol are expressed in a π-calculus-like notation, whereas the attacker is represented implicitly by the process calculus notion of 'environment'. Security properties are modelled in terms of contextual equivalences, in contrast to the previous approaches.

In this paper, we follow the classical approach of Dolev and Yao [DY83] where all communications are mediated by an hostile adversary and we formulate the verification task as a reachability problem. This is the problem of determining if a certain (finite) parallel program which models the protocol and the specification can reach an erroneous state while interacting with the adversary. In previous work [AP99], we have already observed that several secrecy and authenticity properties can be expressed in this framework. That work was formulated in a name-passing formalism akin to the π-calculus. We found it difficult to obtain stronger decidability results in that framework and for this reason we move in this paper to a 'first-order' formalisation where 'messages' are modelled as ground terms of a given first-order signature. Standard tools for symbolic computation such as syntactic unification and tree automata are then available.

The contribution of this paper is to present a direct, self-contained, and hopefully illuminating proof of decidability for the reachability problem for finite principals. A finite principal is a process that can perform a finite number of steps and then terminates. The essential difficulty in this problem comes from the fact that we do not assume any bound on the depth of the messages synthesized by the adversary. Nevertheless, it may appear that the restriction to finite principals is so strong that decidability should easily follow. To our regret we were unable to find a 'trivial' proof of this fact. Our approach amounts to produce a system of syntactic equations and membership constraints and a terminating *strategy* to check its consistency. Huima in [Hui99] seems to be the first to address the reachability problem for finite principals. While being more ambitious, his solution is quite involved and does not appear to be complete. Later, in a work independent from ours, Boreale [Bor00] has also proposed a decision procedure for the same problem. His approach seems to be quite different from ours (at the time of writing, the proofs of Boreale's approach are not available and a full comparison is not possible). In [DLMS99], cryptographic protocols are described as 'Horn theories' and various decidability and undecidability results are given. It remains to be seen whether a faithful and efficient translation of our programs into these 'Horn theories' is possible. As pointed out in op. cit., at least the conditional seems to present some difficulty.

2 Model

In this section, we introduce our model, we explain how specifications are expressed, and we state the reachability problem. We consider terms over an infinite signature:

$$\Sigma = \{C_n^0\}_{n\in\omega} \cup \{E^2, \langle \_, \_ \rangle^2\} \ .$$

Thus we have an infinite set of constants and two binary functions E for encoding and $\langle \_, \_ \rangle$ for pairing. We set $\Sigma(n) = \{C_0, \ldots, C_n\} \cup \{E^2, \langle \_, \_ \rangle^2\}$. We use the following notation: $x, y, \ldots$ for (term) variables; V for a set of variables; $T_\Sigma(V)$ for the collection of finite terms over $\Sigma \cup V$; $t, t', \ldots$ for terms in $T_\Sigma(V)$; $\boldsymbol{t}$ for vectors of terms; $[t/x]$ for the substitution of t for x. We denote with $Var(t)$ the variables occurring in the term t.

Names and Messages In our modelling of messages we follow [Pau99,BDNP99]. Thus we distinguish between basic names (agent's names, nonces, keys,...) and composed messages. The set of names $\mathcal{N}$ is defined as $\{C_n^0\}_{n\in\omega}$ and the set $\mathcal{M}$ is defined as the least set that contains $\mathcal{N}$ and such that:

$$\begin{array}{lll} t \in \mathcal{M} \text{ and } t' \in \mathcal{N} & \Rightarrow & E(t,t') \in \mathcal{M} \\ t, t' \in \mathcal{M} & \Rightarrow & \langle t, t' \rangle \in \mathcal{M} \ . \end{array}$$

In a similar way, we define the set $\mathcal{N}(m)$ as the initial sequence $\{C_n^0\}_{n\leq m}$ and the set $\mathcal{M}(m)$ as the least set that contains $\mathcal{N}(m)$ and such that (i) $E(t,t') \in \mathcal{M}(m)$ if $t \in \mathcal{M}(m)$ and $t' \in \mathcal{N}(m)$, and (ii) $\langle t, t' \rangle \in \mathcal{M}(m)$ if $t, t' \in \mathcal{M}(m)$.

Processes Processes are defined as follows:

$$\begin{array}{l} P ::= 0 \mid \mathit{err} \mid !t.P \mid ?x.P \mid \nu x\, P \mid P \mid P' \mid \mathsf{let}\ x = t\ \mathsf{in}\ P \mid \\ \qquad \mathsf{case}\ E(x,t) = t'\ \mathsf{in}\ P \mid \mathsf{case}\ \langle x, y \rangle = t\ \mathsf{in}\ P \mid [t = t']P, P' \ . \end{array}$$

Processes, sometimes called principals, describe the behaviour of the agents participating in the protocol. The informal interpretation is the following: 0 is the process which is terminated in a sound state; *err* is the process which is terminated in an erroneous state; $!t.P$ evaluates t and if t is a message, sends it to the adversary and becomes P (otherwise it terminates); $?x.P$ receives a message t from the adversary and becomes $[t/x]P$; $\nu x\, P$ creates a fresh name C and becomes $[C/x]P$; $P \mid P'$ is the asynchronous parallel composition of P and P'; $\mathsf{let}\ x = t\ \mathsf{in}\ P$ evaluates t and if t is a message becomes $[t/x]P$ (otherwise it terminates); $\mathsf{case}\ E(x,t) = t'\ \mathsf{in}\ P$ evaluates t' and if t' is a message of the shape $E(t'', t)$ it becomes $[t''/x]P$ (otherwise it terminates); $\mathsf{case}\ \langle x, y \rangle = t\ \mathsf{in}\ P$ evaluates t and if t is a message of the shape $\langle t', t'' \rangle$ it becomes $[t'/x, t''/y]P$, (otherwise it terminates); $[t = t']P, P'$ evaluates t and t' and if both are messages then it becomes P if $t \equiv t'$ and P' if $t \neq t'$ (otherwise it terminates). We denote with $FV(P)$ the set of variables occurring free in P.

Configuration Let P be a process and T be a set of possibly open terms. We define $cnst(P)$ and $cnst(T)$ as the set of names that occur in P and T, respectively. A *well-formed configuration* k is a triple (P, n, T) where (i) P is a closed process, (ii) $n \in \omega$, (iii) T is a non-empty finite subset of $\mathcal{M}$, and (iv) $cnst(P) \cup cnst(T) \subseteq \mathcal{N}(n)$. P represents the principals' behaviour, n is a counter used to generate fresh names, and T stands for the knowledge of the adversary. We assume T non-empty to avoid the paradoxical situation where an input cannot be fired because the knowledge of the adversary is empty.

Reduction In figure 1, we define a reduction relation on well-formed configurations. In these rules, we always reduce the leftmost process with the proviso that parallel composition is associative and commutative. Moreover, we take the liberty of writing P as $P \mid 0$ whenever needed to apply a rewriting rule.

$$
\begin{array}{lll}
(!) & (!t.P \mid P', n, T) & \rightarrow (P \mid P', n, T \cup \{t\}) \text{ if } t \in \mathcal{M} \\
(?) & (?x.P \mid P', n, T) & \rightarrow ([t/x]P \mid P', n, T) \text{ if } t \in S(A(T)) \\
(\nu) & (\nu x\, P \mid P', n, T) & \rightarrow ([C_{n+1}/x]P \mid P', n+1, T) \\
(\mathsf{l}) & (\mathsf{let}\ x = t\ \mathsf{in}\ P \mid P', n, T) & \rightarrow ([t/x]P \mid P', n, T) \text{ if } t \in \mathcal{M} \\
(\mathsf{c}_1) & (\mathsf{case}\ E(x, t') = E(t, t')\ \mathsf{in}\ P \mid P', n, T) & \rightarrow ([t/x]P \mid P', n, T) \text{ if } t' \in \mathcal{N}, t \in \mathcal{M} \\
(\mathsf{c}_2) & (\mathsf{case}\ \langle x, y\rangle = \langle t, t'\rangle\ \mathsf{in}\ P \mid P', n, T) & \rightarrow ([t/x, t'/y]P \mid P', n, T) \text{ if } t, t' \in \mathcal{M} \\
(\mathsf{m}_1) & ([t = t]P_1, P_2 \mid P', n, T) & \rightarrow (P_1 \mid P', n, T) \text{ if } t \in \mathcal{M} \\
(\mathsf{m}_2) & ([t = t']P_1, P_2 \mid P', n, T) & \rightarrow (P_2 \mid P', n, T) \text{ if } t \neq t', t, t' \in \mathcal{M}
\end{array}
$$

Fig. 1. Reduction on configurations

In the operational semantics, we suppose that a thread is stuck whenever it tries (i) to evaluate a term which is not a message or (ii) to decrypt with a name a message which is not encrypted with the very same name, or (iii) to project a message which is not a pair. We write $k \rightarrow_R k'$ if the configuration k reduces to the configuration k' by applying the rule R.

The functions S (synthesis) and A (analysis) are closure operators over the powerset of closed terms defined as follows (similar operators have already been considered in the literature, see, *e.g.*, [Pau99]).

Synthesis: $S(T)$ is the least set that contains T and such that

$$
\begin{array}{ll}
t_1, t_2 \in S(T) & \Rightarrow \ \langle t_1, t_2\rangle \in S(T) \\
t_1 \in S(T), t_2 \in T \cap \mathcal{N} & \Rightarrow \ E(t_1, t_2) \in S(T)\ .
\end{array}
$$

Analysis: $A(T)$ is the least set that contains T and such that

$$
\begin{array}{ll}
\langle t_1, t_2\rangle \in A(T) & \Rightarrow \ t_i \in A(T)\ i = 1, 2 \\
E(t_1, t_2) \in A(T), t_2 \in A(T) & \Rightarrow \ t_1 \in A(T)\ .
\end{array}
$$

We remark that well-formed configurations are closed under reduction. We also note that in our modelling, processes can only send messages (not arbitrary

terms) to the adversary and vice versa the adversary can only send messages to the processes. The following properties follow from the inductive definition of the operators S and A.

Proposition 1. (1) *If* $T \subseteq \mathcal{M}$ *then* $S(T), A(T) \subseteq \mathcal{M}$.
(2) *If* $T \subseteq \mathcal{M}(m)$ *then* $S(T), A(T) \subseteq \mathcal{M}(m)$ *for any* $m \geq 0$.
(3) *Define a function* G *as*

$$G(T) = T \cup \{t_1, t_2 \mid \langle t_1, t_2 \rangle \in T\} \cup \{t_1 \mid E(t_1, t_2), t_2 \in T\}$$

Then $A(T) = \bigcup_{n \in \omega} G^n(T)$.

Related modelling We briefly comment some modelling decisions:

- Monniaux [Mon99] and Huima [Hui99] consider a signature with constructors such as encryption and pairing and *destructors* such as decryption and projection. In their approach, terms are considered up to the equality induced by a canonical term rewriting system. In our approach, the decryption and projection functions are handled implicitly: the principals can decrypt and project using the case operators and the adversary can decrypt and project according to the definition of the analysis operator A. In this way, we can work directly with the free algebra (as in [Bor00]).
- Certain authors, *e.g.* [Hui99], allow arbitrary message –not just names– as encryption keys. It remains to be seen whether our approach can deal with this more general framework.
- Monniaux [Mon99] advocates the use of tree automata to represent the set of messages that can be synthesized by the adversary and to abstract the possible values that can be taken by, say, an input variable. This approach can be followed in our framework too since regular tree languages are closed under synthesis and analysis.
- The process calculus considered here can be regarded as a fragment of the *spi*-calculus [AG97] where all communications transit on public, *i.e.*, unrestricted channels.

We turn next to the definition of the reachability problem.

Definition 1. *Let* $k \equiv (P, n, T)$ *be a configuration. We write* $k \downarrow$ *err if* $P \equiv$ *err* $\mid P'$ *(up to associativity, commutativity, and* $P \mid 0 \equiv P'$*). We also write* $k \downarrow_*$ *err if* $k \xrightarrow{*} k'$ *and* $k \downarrow$ *err. We then say that* k *can reach error.*

In this paper, the *reachability problem* is the problem of determining whether a configuration can reach error. It is easy to prove that the problem is NP-hard by reducing satisfaction of boolean formulas to reachability for processes involving just input, output, parallel composition and decryption.

The method to specify a particular property is to program an *observer* process that will reach error exactly when the property is violated. For instance, suppose we want to specify that in $\nu x\, P$ the name x will remain secret. Upon

creating the name x, we spawn an observer process that challenges the adversary to send him the name x. If the adversary succeeds, the observer ends in an erroneous state. Thus we compile the process $\nu x\, P$ as $\nu x\, (?y.[x = y]err, 0 \mid P)$. Similar techniques can be applied to the specification of authenticity properties. To check that a message received by a principal A is the message previously sent by a principal B, we introduce an observer O such that: (i) before sending the message, B makes sure the message is registered with O, (ii) upon receiving a supposedly authentic message, A queries O for a certification, (iii) O reaches error if it receives a certification request which does not correspond to a previously registered message. We refer to [AP99] for the programming of this little protocol and for a substantial example (Yahalom's protocol).

We say that two well-formed configurations $k_i \equiv (P_i, n_i, T_i)$, $i = 1, 2$ are *equivalent*, and we write $k_1 \cong k_2$ if there is a bijection $\sigma : cnst(P_1) \cup cnst(T_1) \to cnst(P_2) \cup cnst(T_2)$ such that $\sigma P_1 = P_2$ and $\sigma T_1 = T_2$. It is easily checked that if $k_1 \cong k_2$ then $k_1 \downarrow_* err$ iff $k_2 \downarrow_* err$. An interesting remark which can be used to optimize the state exploration procedure is that all reductions but input are strongly confluent up to equivalence.

3 Basic Symbolic Reduction

The starting point is the following natural idea: replace the infinitely branching input transition (?) with a transition where the input variable is not instantiated but is subject to a set-theoretic constraint. In this section, we develop a 'symbolic' reduction relation which operates on configurations with free variables subject to set-theoretic constraints and show that it is finitely branching, terminating, 'sound', and 'complete' thus entailing a decision procedure for the reachability problem.

Definition 2. *We define the set $\mathcal{M}_V$ of* open *messages as the least set that contains $\mathcal{N} \cup V$ and such that if $t, t' \in \mathcal{M}_V$ and $t'' \in \mathcal{N}$ then $E(t, t'') \in \mathcal{M}_V$ and $\langle t, t' \rangle \in \mathcal{M}_V$.*

We note that (i) if $E(t, t')$ is a subterm of a term in $\mathcal{M}_V$ then $t' \in \mathcal{N}$ and that (ii) if $t \in \mathcal{M}_V$ and σ is a substitution associating variables to elements of $\mathcal{M}_V$ then $\sigma(t) \in \mathcal{M}_V$. This definition forbids variables in key position i.e. terms containing $E(t, x)$ as a subterm. In section 4, we shall see how to lift this restriction.

Definition 3. *Let $T \subseteq \mathcal{M}_V$ and $K \subseteq_{fin} \mathcal{N}$.*

(1) *Suppose $t \in \mathcal{M}_V$. We say that t' is K-accessible in t iff either $t \equiv t'$ or $t \equiv \langle t_1, t_2 \rangle$ and for some $i \in \{1, 2\}$, t' is K-accessible in t_i or $t \equiv E(t_1, C)$, $C \in K$, and t' is K-accessible in t_1.*

(2) *We define $P_K(T)$, the K-accessible parts of T, as the set of terms t' that are K-accessible in a term $t \in T$.*

(3) *We define $I_K(T)$, the K-irreducible parts of T, as*

$$I_K(T) = P_K(T) \cap \{E(t, C) \mid t \in \mathcal{M}_V \text{ and } C \notin K\} .$$

(4) *If moreover* $T \subseteq \mathcal{M}$, *we define* $S_K(T)$, *the* K*-synthesis of* T, *as the least set of terms that contains* $T \cup K$ *and is closed under pairing and encryption by a name in* K.

(5) *We define* $\mathcal{T}_K$ *as* $\mathcal{T}_K = \{t \in \mathcal{M} \mid A(\{t\}) \cap \mathcal{N} = K\}$. *These are the terms from which exactly the set of names* K *can be learned by analysis.*

(6) *Finally, we define* $K(T)$ *as the least set such that* $C \in P_{K(T)}(T)$ *implies* $C \in K(T)$.

Example 1. if $T = \{C_1, \langle E(C_2, C_1), E(E(C_3, C_3), C_2)\rangle\}$ then $K(T) = \{C_1, C_2\}$ and $I_{K(T)}(T) = \{E(C_3, C_3)\}$. We remark that, assuming T finite, $K(T)$ and $I_{K(T)}(T)$ can be computed in time proportional to the number of symbols in T.

The following properties can be easily derived from the definition 3.

Lemma 1. *Suppose* $T \subseteq \mathcal{M}$, $t, t_1, t_2 \in \mathcal{M}$, $K \subseteq_{fin} \mathcal{N}$, $C \in \mathcal{N}$. *Then:*

(1) *If* $t \in \mathcal{T}_K$ *then* $S(A(\{t\})) = S_K(I_K(\{t\}))$.

(2) $t \in \mathcal{T}_K$ *iff* $K = K(\{t\})$.

(3) $C \in S_K(I_K(T))$ *iff* $C \in K$.

(4) $\langle t_1, t_2\rangle \in S_K(I_K(T))$ *iff* $t_i \in S_K(I_K(T)), i = 1, 2$.

(5) *Suppose* $C \in K$. *Then* $E(t, C) \in S_K(I_K(T))$ *iff* $t \in S_K(I_K(T))$,

(6) *Suppose* $C \notin K$. *Then* $E(t, C) \in S_K(I_K(T))$ *iff* $E(t, C) \in I_K(T)$.

If $T = \{t_1, \ldots, t_n\} \subset \mathcal{M}$ then we abbreviate $\langle t_1, \langle \ldots, \langle t_{n-1}, t_n\rangle \cdots\rangle \in \mathcal{T}_K$ by writing $T \in \mathcal{T}_K$. We now come to the lemma which is the keystone in the definition of symbolic reduction. Let $\boldsymbol{T} \equiv T_1 \subseteq \cdots \subseteq T_n$ be a non decreasing sequence of finite sets of terms in $\mathcal{M}_V$ such that $Var(T_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ and let σ be a *compatible* substitution, where compatibility is defined as:

$$\begin{array}{ll} \sigma(x_i) \in S(A(\sigma(T_i))) & \text{for } i = 1, \ldots, n, \\ \sigma(y) = y & \text{if } y \notin \{x_1, \ldots, x_n\}. \end{array}$$

Lemma 2. *Under the hypotheses above:*

(1) $K(\sigma T_i) = K(T_i)$.

(2) $I_{K(\sigma T_i)}(\sigma T_i) = \sigma(I_{K(T_i)}(T_i))$.

Definition 4. *A symbolic configuration is a triple* (P, T, E) *where:*

(1) P *is a process and* T *is a non-empty finite set of terms in* $\mathcal{M}_V$ *such that for some set of variables* $\{x_1, \ldots, x_n\}$, $FV(P) \cup Var(T) \subseteq \{x_1, \ldots, x_n\}$.

(2) E *is an* environment, *namely a possibly empty list of the shape* $x_1 : T_1, \ldots, x_n : T_n$ *where* $T_1 \subseteq \cdots \subseteq T_n \subseteq T$, $T_1 \neq \emptyset$, *and* $Var(T_i) \subseteq \{x_1, \ldots, x_{i-1}\}$.

We note that $K(T_1) \subseteq \cdots \subseteq K(T_n)$. In figure 2, we define a basic system to rewrite symbolic configurations. As usual, we assume that all bound variables are distinct and different from the free variables. The rules symmetric to (m^s_{3-6}) are omitted. Moreover, the substitution $[t/x](P, T, E)$ is defined as

$([t/x]P, [t/x]T, [t/x]E)$, and $[t/x]E$ is just an abbreviation defined in figure 2 in the special cases we need.

The strategy of the symbolic reduction is to maintain the constraints $x_1 : T_1, \ldots, x_n : T_n$ in the form required to apply lemma 2. We note in particular, that in the rules (m^s_{5-6}) for equalities we do not follow the usual unification procedure. For instance, an equation $[x_i = \langle t_1, t_2\rangle]$ is not directly eliminated by a substitution $[\langle t_1, t_2\rangle / x_i]$ as t_i may contain variables of 'higher rank', *e.g.*, it may contain a variable x_j whose constraint T_j depends on x_i. Instead, we perform the substitution $[\langle x', x''\rangle / x_i]$ and solve the equations $[x' = t_1]$ and $[x'' = t_2]$ where x' and x'' are fresh variables with the same rank as x_i. It remains to be seen whether our procedure is subsumed by a more general constraint-satisfaction method.

For the sake of simplicity, we consider first processes P which satisfy the following conditions:

(1) All terms occurring in P belong to $\mathcal{M}_V$.

(2) P does not contain the operator ν for name creation (thus in a configuration we omit the counter n).

(3) P does not contain the operator $|$ of parallel composition.

(4) All conditionals occurring in P are of the form $[t = t']P, 0$ that we abbreviate as $[t = t']P$.

These conditions are preserved by reduction. In section 4 , we show how to lift these restrictions by using a simple extension of the basic rewrite system.

We can show that a symbolic configuration $(P, T, \emptyset)$ reduces to error iff the configuration (P, T) does. Moreover, we can prove that all symbolic reductions are terminating and finitely branching. This entails a decision procedure for the reachability problem: explore all symbolic reductions and check whether they lead to an erroneous symbolic configuration. We consider two simple examples of application of this procedure.

Example 2. (1) Consider the process

$$P_1 \equiv ?x_1.!E(x_1, C_1).?x_2.\mathsf{case}\ E(x_3, C_1) = x_2\ \mathsf{in}\ \mathsf{case}\ E(x_4, C_0) = x_3\ \mathsf{in}\ \mathit{err}$$

where initially $T_1 = \{C_0\}$. We show $(P_1, T_1, \emptyset) \downarrow_* \mathit{err}$. We have:

$$\begin{array}{l}
(P_1, T_1, \emptyset) \ \rightarrow\ (!E(x_1, C_1) \ldots, T_1, x_1 : T_1) \text{ by } (?^s) \\
\rightarrow\ (?x_2 \ldots, T_2, x_1 : T_1) \text{ where } T_2 = T_1 \cup \{E(x_1, C_1)\},\ \text{by } (!^s) \\
\rightarrow\ (\mathsf{case}\ E(x_3, C_1) = x_2\ \mathsf{in}\ \ldots, T_2, E_2) \text{ where } E_2 = x_1 : T_1, x_2 : T_2,\ \text{by } (?^s) \\
\rightarrow\ [E(x_1, C_1)/x_2]([x_1/x_3]\mathsf{case}\ E(x_4, C_0) = x_3\ \mathsf{in}\ \mathit{err}, T_2, E_2),\ \text{by } (\mathsf{c}^s_5) \\
\equiv\ (\mathsf{case}\ E(x_4, C_0) = x_1\ \mathsf{in}\ \mathit{err}, T_2, x_1 : T_1),\ \text{by substitution} \\
\rightarrow\ (\mathit{err}, T_2, x_4 : T_1),\ \text{by } (\mathsf{c}^s_4)\ .
\end{array}$$

Following the substitutions backwards, we can express the set of successful 'attacks' of the adversary as $x_1 = E(x_4, C_0), x_2 = E(E(x_4, C_0), C_1)$ where $x_4 \in S(\{C_0\})$.

(2) Consider the process

$$P_1 \equiv ?x_1.\mathsf{case}\ E(x_2, C_2) = x_1\ \mathsf{in}\ !x_2.?x_3.\mathsf{case}\ \langle x_4, x_5\rangle = x_3\ \mathsf{in}\ [x_4 = C_0][x_5 = C_1]\mathit{err}$$

$(?^s)\ \ (?x.P,T,E) \ \rightarrow\ (P,T,E,x:T)$
$(!^s)\ \ (!t.P,T,E) \ \rightarrow\ (P,T\cup\{t\},E)$
$(\mathsf{l}^s)\ \ (\mathsf{let}\ x=t\ \mathsf{in}\ P,T,E) \ \rightarrow\ ([t/x]P,T,E)$

$(\mathsf{c}_1^s)\ \ (\mathsf{case}\ \langle x',x''\rangle=\langle t_1,t_2\rangle\ \mathsf{in}\ P,T,E) \ \rightarrow\ ([t_1/x',t_2/x'']P,T,E)$
$(\mathsf{c}_2^s)\ \ (\mathsf{case}\ \langle x',x''\rangle=x_i\ \mathsf{in}\ P,T,E) \ \rightarrow\ [\langle x',x''\rangle/x_i](P,T,E)$

$(\mathsf{c}_3^s)\ \ (\mathsf{case}\ E(x,C)=E(t,C)\ \mathsf{in}\ P,T,E) \ \rightarrow\ ([t/x]P,T,E)$
$(\mathsf{c}_4^s)\ \ (\mathsf{case}\ E(x,C)=x_i\ \mathsf{in}\ P,T,E) \ \rightarrow\ [E(x,C)/x_i](P,T,E)\ \ \text{if } C\in K(T_i)$
$(\mathsf{c}_5^s)\ \ (\mathsf{case}\ E(x,C)=x_i\ \mathsf{in}\ P,T,E) \ \rightarrow [E(t,C)/x_i]([t/x]P,T,E)$
$\text{if } E(t,C)\in I_{K(T_i)}(T_i)$

$(\mathsf{m}_1^s)\ \ ([f(t_1,\ldots,t_n)=f(s_1,\ldots,s_n)]P,T,E) \ \rightarrow\ ([t_1=s_1]\ldots[t_n=s_n]P,T,E)$
$f \text{ constructor}$
$(\mathsf{m}_2^s)\ \ ([x_i=x_i]P,T,E) \ \rightarrow\ (P,T,E)$
$(\mathsf{m}_3^s)\ \ ([x_i=x_j]P,T,E) \ \rightarrow\ [x_i/x_j](P,T,E)\ \ \text{if } i<j$
$(\mathsf{m}_4^s)\ \ ([x_i=C]P,T,E) \ \rightarrow\ [C/x_i](P,T,E)\ \ \text{if } C\in K(T_i)$
$(\mathsf{m}_5^s)\ \ ([x_i=\langle t_1,t_2\rangle]P,T,E) \ \rightarrow\ (\mathsf{case}\ \langle x',x''\rangle=x_i\ \mathsf{in}\ [x'=t_1][x''=t_2]P,T,E)$
$x_i\notin Var(t_i)$
$(\mathsf{m}_6^s)\ \ ([x_i=E(t,C)]P,T,E) \ \rightarrow\ (\mathsf{case}\ E(x,C)=x_i\ \mathsf{in}\ [x=t]P,T,E)\ x_i\notin Var(t)$

$$
\begin{aligned}
[\langle x',x''\rangle/x_i]E \equiv\ & x_1:T_1,\ldots,x_{i-1}:T_{i-1},x':T_i,x'':T_i,\\
& x_{i+1}:[\langle x',x''\rangle/x_i]T_{i+1},\ldots,x_n:[\langle x',x''\rangle/x_i]T_n
\end{aligned}
$$

$$
\begin{aligned}
[E(x,C)/x_i]E \equiv\ & x_1:T_1,\ldots,x_{i-1}:T_{i-1},x:T_i,\\
& x_{i+1}:[E(x,C)/x_i]T_{i+1},\ldots,x_n:[E(x,C)/x_i]T_n
\end{aligned}
$$

$$
\begin{aligned}
[E(t,C)/x_i]E \equiv\ & x_1:T_1,\ldots,x_{i-1}:T_{i-1},\\
& x_{i+1}:[E(t,C)/x_i]T_{i+1},\ldots,x_n:[E(t,C)/x_i]T_n
\end{aligned}
$$

$$
\begin{aligned}
[x_i/x_j]E \equiv\ & x_1:T_1,\ldots,x_{j-1}:T_{j-1},\\
& x_{j+1}:[x_i/x_j]T_{j+1},\ldots,x_n:[x_i/x_j]T_n
\end{aligned}
$$

$$
\begin{aligned}
[C/x_i]E \equiv\ & x_1:T_1,\ldots,x_{i-1}:T_{i-1}\\
& x_{i+1}:[C/x_i]T_{i+1},\ldots,x_n:[C/x_i]T_n\ .
\end{aligned}
$$

Fig. 2. Basic symbolic reduction

where initially $T_1=\{E(C_0,C_2),E(C_1,C_2)\}$. We show that $(P_1,T_1,\emptyset)$ does not reach error. We have:

$$(P_1,T_1,\emptyset) \ \rightarrow\ (\mathsf{case}\ E(x_2,C_2)=x_1\ \mathsf{in}\ \ldots,T_1,x_1:T_1),\ \text{by}\ (?^s)\ .$$

We then apply (c_5^s). Since $\sharp I_\emptyset(T_1)=2$ we have to consider two cases: $x_2=C_0$ or $x_2=C_1$. We develop only the first one, the second being quite similar.

$\rightarrow$ $(!C_0.?x_3 \ldots, T_1, \emptyset)$, by $(\mathsf{c}_5^s), x_2 = C_0$
$\rightarrow$ $(?x_3 \ldots, T_2, \emptyset)$, where $T_2 = T_1 \cup \{C_0\}$, by $(!^s)$
$\rightarrow$ $(\mathsf{case}\ \langle x_4, x_5\rangle = x_3\ \mathsf{in}\ \ldots, T_2, x_3 : T_2)$, by $(?^s)$
$\rightarrow$ $([x_4 = C_0][x_5 = C_1]err, T_2, x_4 : T_2, x_5 : T_2)$, by (c_2^s)
$\rightarrow$ $([x_5 = C_1]err, T_2, x_5 : T_2)$, by (m_4^s), and we are stuck as $C_1 \notin K(T_2)$.

Termination of basic reduction is not obvious and requires a careful analysis of reductions starting from configurations $([s = t]P, T, E)$. A reduction ordering is defined on such sequences and we show that sequences of reduction eventually decrease the size of the configuration. The termination proof is available in the full version of the paper and allows to state the next theorem.

Theorem 1. *Basic symbolic reduction always terminates.*

We relate symbolic reductions and reductions as follows.

Definition 5. *Let $k \equiv (P, T, E)$ be a symbolic configuration and σ be a ground substitution.*

(1) *We write $k \downarrow err$ if $P \equiv err$ and $k \downarrow_* err$ if $k \xrightarrow{*} k'$ and $k' \downarrow err$.*

(2) *We write $\sigma \models E$ iff σ is compatible with the sequence $T_1 \subseteq \cdots \subseteq T_n$ in E.*

Finally, we can state soundness and completeness of symbolic reduction with respect to reduction. Completeness makes an essential use of lemma 2.

Theorem 2. $(P, T, E) \downarrow_* err$ *iff* $\exists\, \sigma \models E\ \ \sigma(P, T) \downarrow_* err$.

4 Extensions of Basic Symbolic Reduction

In this section, we show how to lift the four restrictions imposed in section 3 thus defining a symbolic reduction for the full model introduced in section 2. Let us first consider the four restrictions separately and present our basic ideas.

Variables in key position Suppose we have to deal with a symbolic configuration of the shape $(\mathsf{let}\ x = E(t, x_i)\ \mathsf{in}\ P, T, E)$. We note that for any substitution σ such that $\sigma \models E$, $\sigma(\mathsf{let}\ x = E(t, x_i)\ \mathsf{in}\ P, T)$ reduces iff $\sigma(E(t, x_i)) \in \mathcal{M}$. In general, $\sigma t \in \mathcal{M}$ iff in every subterm $E(t', t'')$ of t, t'' is either a name or a variable, say x. In the later case, we say that x is a variable in *key position* and $\sigma(x)$ ranges over names.

By the constraints imposed by the environment E, if $\sigma(x_i) \in \mathcal{N}$ then $x_i \in K(T_i)$ which is a finite set. Thus there are only a finite number of assignments of the variables in key positions that are compatible with E.

Formally, let $T_{\mathcal{M}} = \{t \in T_\Sigma(V) \mid \exists \sigma\ \ \sigma t \in \mathcal{M}\}$. We note that a term t is in $T_{\mathcal{M}}$ iff in all subterms $E(t, t')$, t' is either a name or a variable. If $t \in T_{\mathcal{M}}$ then let $Var_{key}(t)$ be the set of variables x that occur in t in 'key position', *i.e.*, such that $E(t', x)$ is a subterm of t.

Definition 6. *We write* $\sigma \downarrow (t, E)$ *if (i)* $\sigma = id$ *and* $t \in \mathcal{M}_V$ *or (ii)* $t \in T_{\mathcal{M}}$, $Var_{key}(t) = \{x_{i_1}, \ldots, x_{i_m}\}$, $m \geq 1$, *and* $\sigma(x_l) \in K(T_l)$, *if* $x_l \in Var_{key}(t)$ *and* $\sigma(x_l) = x_l$, *otherwise.*

With this notation, we rewrite the rules for let, output, case, and conditional by combining reduction and instantiation of variables in key position. For instance, the rule (l) becomes:

$$(\mathsf{let}\ x = t\ \mathsf{in}\ P, T, E) \ \rightarrow\ \sigma([t/x]P, T, E) \ \text{ if } \sigma \downarrow (t, E)\ .$$

This is a brute force method. In practice, a more efficient approach is to restrict the range of a variable x_i in key position to the corresponding finite set of names $K(T_i)$.

Restriction As in the reduction of (standard) configurations (figure 1), it is enough to introduce a counter. Then a symbolic configuration is a quadruple (P, n, T, E) where n is a natural number such that $cnst(P) \cup cnst(T) \subseteq \mathcal{N}(n)$. We then add the following rule for restriction.

$$(\nu x\, P, n, T, E) \ \rightarrow\ ([C_{n+1}/x]P, n+1, T, E)\ .$$

Parallel composition We assume that parallel composition is associative and commutative and that $P \equiv P \mid 0$. Then, without loss of generality, we always reduce the leftmost thread of a process and we assume that there is at least another thread running in parallel. For instance, the rule for output is rewritten as follows.

$$(!t.P \mid P', T, E) \ \rightarrow\ (P \mid P', T \cup \{t\}, E)$$

We note that the confluence of non-input reductions does not apply –literally– to symbolic reductions. For instance, consider the symbolic configuration

$$(\mathsf{let}\ x = E(t, x_1)\ \mathsf{in}\ 0 \mid [x_1 = \langle C_0, C_0\rangle]err, \{C_0\}, x_1 : \{C_0\})\ .$$

The output and the conditional do not commute as to fire the (l) rule we have to instantiate the variable x_1 with a name thus precluding the reachability of error in the other parallel component. We expect that useful confluence results can be obtained when the threads running in parallel do not share variables.

Conditional We enrich a symbolic configuration with another component I that is a finite set of inequalities $s \neq t$ or a special symbol $\bot$ (which is never satisfied). We write $\sigma \models I$ iff $I \neq \bot$ and σ satisfies all inequalities in I. We present in figure 3 a standard set of simplification rules for inequalities.

We note that each rule decreases the number of symbols. Then, we can state the following proposition.

Proposition 2. *Rewriting by the rules for inequalities* (i_{1-4}) *in figure 3 always terminates.*

We can now introduce our most general notion of symbolic reduction.

$$
\begin{array}{llll}
(\mathsf{i}_1) & \{t \neq t\} \cup I \;\rightarrow\; \bot & (\mathsf{i}_3) & \{f(\boldsymbol{s}) \neq f(\boldsymbol{t})\} \cup I \;\rightarrow\; \{s_i \neq t_i\} \cup I \\
 & & & \qquad 1 \leq i \leq arity(f) \\
(\mathsf{i}_2) & \{f(\boldsymbol{s}) \neq g(\boldsymbol{t})\} \cup I \;\rightarrow\; I & (\mathsf{i}_4) & \{x \neq t\} \cup I \;\rightarrow\; I \;\text{ if } x \in Var(t) \text{ and } t \not\equiv x
\end{array}
$$

Fig. 3. Reduction for inequalities

Definition 7. *We say that a set of inequalities is* simplified *if it is not equal to* $\bot$ *and the reduction rules* (i_{1-4}) *do not apply.*

Definition 8. *A symbolic configuration is a quintuple* (P, n, T, E, I) *where:*

(1) P *is a process (as specified above) and* T *is a non-empty finite set of terms in* $\mathcal{M}_V$ *such that for some set of variables* $\{x_1, \ldots, x_n\}$, $FV(P) \cup Var(T) \subseteq \{x_1, \ldots, x_n\}$.

(2) E *is an environment,* $x_1 : T_1, \ldots, x_n : T_n$ *as specified in definition 4.*

(3) n *is a natural number such that if* $C_m \in cnst(P) \cup cnst(T) \cup cnst(I)$ *then* $m \leq n$.

(4) I *is a finite set of inequalities or* $\bot$.

We present in figures 4,5 the full system for symbolic reduction where we denote with u either a variable or a constant and with P a process that admits as subprocesses two special forms of the conditional, namely $[t = t']P$ and $[t \neq t']P$, provided $t, t' \in \mathcal{M}_V$.

Definition 9. *Let* $k \equiv (P, n, T, E, I)$ *be a symbolic configuration. We write* $k \downarrow$ *err if* $P \equiv err \mid P'$ *and* I *is simplified. As usual, we write* $k \downarrow_* err$ *if* $k \stackrel{*}{\rightarrow} k'$ *and* $k' \downarrow err$.

This definition of symbolic reachability of error is justified by the observation that if I is simplified then we can always find a substitution σ such that $\sigma \models E$ and $\sigma \models I$ (henceforth abbreviated as $\sigma \models E, I$).

Next we comment the reduction rules. The first group of rules presented in figure 4 deals with all operators but conditional and applies the ideas sketched above to handle variables in key position, restriction, and parallel composition. The rules to handle the conditional are rules (m^s_{1-7}) of figure 5. In the first rule (m^s_1), we instantiate the terms in the conditional $([t = t']P_1, P_2)$ and split it in two parts $\sigma([t = t']P_1)$ and $\sigma([t \neq t']P_1)$. Then, we develop the rules that handle the simpler conditionals $[t = t']P$ and $[t \neq t']P$ where $t, t' \in \mathcal{M}$. Equalities are reduced by the rules (m^s_{2-7}) that are similar to the corresponding rules of figure 2 (omitting the rules symmetric to (m^s_{4-7})). Inequalities $[t \neq t']P$ are shifted to I which is then simplified by rule (i^s_{1-4}). The reduction process may diverge when we interleave the rules for equalities in two parallel threads with shared variables. For instance, consider the reduction

$$
\begin{array}{l}
([x = \langle y, z\rangle]0 \mid [y = \langle x, w\rangle]0, \ldots) \\
\rightarrow ([x' = y][x'' = z]0 \mid [y = \langle\langle x', x''\rangle, w\rangle]0, \ldots, \ldots) \\
\rightarrow ([x' = \langle y', y''\rangle][x'' = z]0 \mid [y' = \langle x', x''\rangle][y'' = w]0, \ldots)
\end{array}
$$

$(?^s)$ $(?x.P \mid P', n, T, E, I) \to (P \mid P', n, T, E, x : T, I)$

$(!^s)$ $(!t.P \mid P', n, T, E, I) \ \to\ \sigma(P \mid P', n, T \cup \{t\}, E, I)$ if $\sigma \downarrow (t, E)$

(l^s) $(\mathsf{let}\ x = t\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ \sigma([t/x]P \mid P', n, T, E, I)$ if $\sigma \downarrow (t, E)$

(ν^s) $(\nu x\, P \mid P', n, T, E, I) \ \to\ ([C_{n+1}/x]P \mid P', n+1, T, E, I)$

(c_1^s) $(\mathsf{case}\ \langle x', x''\rangle = \langle t_1, t_2\rangle\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ \sigma([t_1/x', t_2/x'']P \mid P', n, T, E, I)$
if $\sigma \downarrow (\langle t_1, t_2\rangle, E)$

(c_2^s) $(\mathsf{case}\ \langle x', x''\rangle = x_i\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ [\langle x', x''\rangle/x_i](P \mid P', n, T, E, I)$

(c_3^s) $(\mathsf{case}\ E(x, u) = E(t, u')\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ \sigma([t/x]P \mid P', n, T, E, I)$
if $\sigma \downarrow (E(E(t, u'), u), E), \sigma(u) = \sigma(u')$

(c_4^s) $(\mathsf{case}\ E(x, C) = x_i\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ [E(x, C)/x_i](P \mid P', n, T, E, I)$
if $C \in K(T_i)$

(c_5^s) $(\mathsf{case}\ E(x, x_j) = x_i\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to\ [E(x, C)/x_i, C/x_j](P \mid P', n, T, E, I)$
if $i \neq j,\ C \in K(T_{min(i,j)})$

(c_6^s) $(\mathsf{case}\ E(x, C) = x_i\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to [E(t, C)/x_i]([t/x]P \mid P', n, T, E, I)$
if $E(t, C) \in I_{K(T_i)}(T_i)$

(c_7^s) $(\mathsf{case}\ E(x, x_j) = x_i\ \mathsf{in}\ P \mid P', n, T, E, I) \ \to [E(t, C)/x_i, C/x_j]$
$([t/x]P \mid P', n, T, E, I)$
if $E(t, C) \in I_{K(T_i)}(T_i), i < j, C \in K(T_j)$

Fig. 4. Symbolic reduction rules (without conditional)

After two reduction steps and modulo renaming, we are back to a configuration of the shape $([x = \langle y, z\rangle]P \mid [y = \langle x, w\rangle]Q, \ldots)$. To avoid this pathological loop, we enforce the following reduction strategy on the rules (m_{2-7}): if one of these rule is applied in a thread $[s = t]Q$ of the configuration $([s = t]Q \mid \ldots, n, T, E, I)$, then we keep applying these rules till we get stuck or we obtain a configuration $\sigma(Q \mid \ldots, n, T', E', I')$. We call this strategy *eager reduction of equations*. The proof of theorem 1 shows that the reduction of a component $([s = t]Q \mid \ldots, n, T, E, I)$ always terminates, and yields a configuration $\sigma(Q \mid \ldots, n, T', E', I')$.

Theorem 3. (1) *Symbolic reduction with the eager reduction of equations always terminates.*
(2) $(P, n, T, E, I) \downarrow_* err$ *iff* $\exists \sigma\ \ (\sigma \models E, I$ *and* $\sigma(P, n, T) \downarrow_* err)$.

V. Vanackere is currently implementing our decision procedure and testing its performance on some typical protocols (the first experiments are encouraging).

$$(\mathsf{m}_1^s)\ ([t = t']P_1, P_2 \mid P', n, T, E, I) \rightarrow \begin{cases} \sigma([t = t']P_1 \mid P', n, T, E, I) \\ \sigma([t \neq t']P_2 \mid P', n, T, E, I) \end{cases} \sigma \downarrow \langle t, t' \rangle$$

$$(\mathsf{m}_2^s)\ ([f(\boldsymbol{t}) = f(\boldsymbol{s})]P \mid P', n, T, E, I) \ \rightarrow\ ([\boldsymbol{t} = \boldsymbol{s}]P \mid P', n, T, E, I)\ f \text{ constr.}$$

$$(\mathsf{m}_3^s)\ ([x_i = x_i]P \mid P', n, T, E, I) \ \rightarrow\ (P \mid P', n, T, E, I)$$

$$(\mathsf{m}_4^s)\ ([x_i = x_j]P \mid P', n, T, E, I) \ \rightarrow\ [x_i/x_j](P \mid P', n, T, E, I) \ \text{ if } i < j$$

$$(\mathsf{m}_5^s)\ ([x_i = C]P \mid P', n, T, E, I) \ \rightarrow\ [C/x_i](P \mid P', n, T, E, I) \ \text{ if } C \in K(T_i)$$

$$(\mathsf{m}_6^s)\ ([x_i = \langle t_1, t_2 \rangle]P \mid P', n, T, E, I)$$
$$\rightarrow\ (\mathsf{case}\ \langle x', x'' \rangle = x_i\ \mathsf{in}\ [x' = t_1][x'' = t_2]P \mid P', n, T, E, I) \text{ if } x_i \notin Var(t_i)$$

$$(\mathsf{m}_7^s)\ ([x_i = E(t, C)]P \mid P', n, T, E, I)$$
$$\rightarrow\ (\mathsf{case}\ E(x, C) = x_i\ \mathsf{in}\ [x = t]P \mid P', n, T, E, I) \text{ if } x_i \notin Var(t)$$

$$(\mathsf{i}^s)\quad ([s \neq t]P \mid P', n, T, E, I) \ \rightarrow\ (P \mid P', n, T, E, I \cup \{s \neq t\})$$

$$(\mathsf{i}_{1-4}^s)\ (P \mid P', n, T, E, I) \ \rightarrow\ (P \mid P', n, T, E, I') \quad \text{if } I \rightarrow I' \text{ by } (\mathsf{i}_{1-4})$$

Fig. 5. Symbolic reduction rules (for conditional)

We intend to enhance our method to handle public keys and complex keys (any term can be used to encrypt). At least for public keys, it appears that no major change is needed in the decision procedure. The exact complexity of our decision procedure remains to be studied.

References

AG97. M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proc. ACM Computer and Comm. Security*, 1997. 381, 384

AP99. R. Amadio and S. Prasad. The game of the name in cryptographic tables. In *Proc. ASIAN99, SLNCS 1742*, pages 15–26, 1999. 381, 385

BDNP99. M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *Proc. IEEE Logic in Comp. Sci.*, 1999. 382

Bol96. D. Bolignano. Formal verification of cryptographic protocols. In *Proc. ACM Conference on Computer Communication and Security*, 1996. 380

Bor00. M. Boreale. Symbolic analysis of cryptographic protocols in the spi-calculus. Personal communication, 2000. 381, 384

CDG$^+$. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Draft available at *http://www.grappa.univ-lille3.fr/tata*.

CJ97. J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Technical report, 1997. Available at http://www.cs.york.ac.uk/~jac/papers/ drareview.ps.gz.

CJM98. E. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Conf. on Progr. Concepts and Methods (PROCOMET)*, 1998. 380

DLMS99. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In Proc. Formal methods and security protocols, FLOC Workshop, Trento, 1999. 381

DY83. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983. 381

Hui99. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Formal methods and security protocols, FLOC Workshop, Trento*, 1999. 381, 384

Low96. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. TACAS*, SLNCS, 1996. 380

Mea94. C. Meadows. A model of computation for the nrl protocol analyzer. In *Proc. IEEE Computer Security Foundations Workshop*, 1994. 380

MMS97. J. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proc. IEEE Symp. on Security and Privacy*, 1997. 380

Mon99. D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Proc. Static Analysis Symp., SLNCS*, 1999. 384

Pau97. L. Paulson. Proving properties of security protocols by induction. In *Proc. IEEE Computer Security Foundations Workshop*, 1997. 380

Pau99. L. Paulson. Proving security protocols correct. In *Proc. IEEE Logic in Comp. Sci.*, 1999. 382, 383

Sch96. S. Schneider. Security properties and CSP. In *Proc. IEEE Symp. Security and Privacy*, 1996. 380

Wei99. C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Proc. CADE 99*. SLNCS 1632, 1999. 381

Secure Information Flow for Concurrent Processes

Jan Jürjens*

LFCS, Division of Informatics, University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, GB
jan@dcs.ed.ac.uk
http://www.dcs.ed.ac.uk/home/jan

Abstract. Information flow security is that aspect of computer security concerned with how confidential information is allowed to flow through a computer system. This is especially subtle when considering processes that are executed concurrently. We consider the notion of *Probabilistic Noninterference* (PNI) proposed in the literature to ensure secure information flow in concurrent processes. In the setting of a model of *probabilistic dataflow*, we provide a number of important results towards simplified verification that suggest relevance in the interaction of probabilistic processes outside this particular framework:
PNI is shown to be compositional by casting it into a rely-guarantee framework, where the proof yields a more general Inductive Compositionality Principle. We deliver a considerably simplified criterion equivalent to PNI by "factoring out" the probabilistic behaviour of the environment. We show that the simpler nonprobabilistic notion of *Nondeducibility-on-Strategies* proposed in the literature is an instantiation of PNI, allowing us to extend our results to it.

1 Introduction

1.1 Motivation

Information security of computer systems has received increased attention in recent years. A topical source of concern is the security of the Internet, which is becoming the world's largest public electronic marketplace. In addition to existing threats e. g. from hacker attacks, there are issues arising from the increasing use of mobile code in languages such as Java. If the Internet should provide the platform for commercial transactions, it is vital that sensitive information (like a credit card number or a cryptographic key) is not leaked out from a business site (e. g. by hackers) or from a user's computer (e. g. by malicious applets). Another example are multi-application smart cards (possibly containing sensitive medical data), especially when considering post-issuance code downloading.

* Supported by the Studienstiftung des deutschen Volkes and the Division of Informatics.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 395–409, 2000.

In this paper we consider *confidentiality*, i. e. the prevention of unauthorised access to information, in a model of concurrent processes. More precisely, we examine the situation of "multilevel-security", where e. g. programs H and L (that could be *Trojan Horses*) running on an operating system M have access to a shared resource. H (for high) is assumed to have access to confidential information that L (for low) should not obtain. Here ensuring confidentiality means preventing information flow from H through M to L. To minimise the verification necessary one would prefer the security of the system composed of H, M and L to only depend on suitable requirements on M.

Leakage of information can happen in rather subtle ways via *covert channels* (discovered by Butler Lampson in 1973). An example for a means of sending a bit of information is the increased use of processor power by H that could be detected by L.

Since eliminating covert channels in existing systems can be expensive, mathematical models have been proposed to detect them in the early phases of system development. This research offers the possibility to mechanically verify formal specifications with respect to security requirements. Since security issues can be very subtle on the one hand, and very costly when neglected on the other, it seems a realistic area for practical use of formal methods.

Recently there has been work on applying the approach to secure information flow considered in this paper to safety-critical systems (e. g. with the goal to avoid interference of possibly faulty off-the-shelf components with safety-relevant ones [DS99]). This suggests usefulness of the results given in this paper outside security of concurrent systems.

1.2 Previous Work

Early approaches to ensure confidentiality in multi-level secure systems (most well-known the model proposed by Bell and LaPadula in 1975) made use of access control, i. e. restricting the operations the respective users could perform. However, these models did not treat covert channels. Therefore the notion of *Noninterference* [GM82] was proposed to reason about secure information flow in deterministic systems. It formalises the idea that absence of information flow from H to L means that L cannot distinguish different behaviours of H.

There are several approaches to generalize Noninterference to nondeterministic systems. [WJ90] showed that earlier models were either too weak or too strong and introduced the notion of *Nondeducibility-on-Strategies* (NOS). It did not take into account probabilistic aspects, and thus considered systems to be secure that are insecure when allowing an attacker to draw conclusions from statistical inference (for an example cf. section 5).

This motivated probabilistic notions inspired by Shannon's information theory, most importantly the *Applied Flow Model* (AFM, [McL90,Gra92]) and *Probabilistic Noninterference* (PNI, [Gra92]). [Gra92] showed that (in the above situation) if M satisfies PNI then the communication channel from H to L has capacity zero.

To enable modular design of and reasoning about secure systems, special attention has been paid to compositionality of security properties, e. g. in [McL96]. For an excellent overview cf. [McL94].

Additionally there has been a significant amount of work on process algebras for information flow including [Ros95,FG97,Low99,RG99,RS99], cf. Section 6.

1.3 Present Work

The main contribution of this paper is the proof of several important properties of Probabilistic Noninterference with the ultimate goal to make it a feasible property for mechanical verification. These results give insights into the interaction between probabilistic processes outside the scope of the specific notions considered.

For comparison, we note some weaknesses of the Applied Flow Model.

We then consider PNI in the setting of a model of *probabilistic dataflow* and prove various results towards a feasible verification of PNI, including equivalent definitions that are easier to verify and compositionality.

The first result shows that in the definition of PNI one only has to quantify over deterministic environments. This greatly reduces the amount of checking to be done when verifying PNI. Further research into this observation should provide similar simplifications for other properties of interacting probabilistic processes.

We show compositionality of PNI, solving an open problem posed in [Gra92]. Rely-guarantee specifications aim to permit compositionality without additional verification effort. We prove compositionality using a rely-guarantee formulation. This facilitates future research in providing weaker conditions on components with secure composition. The proof uses an Inductive Compositionality principle to be used in more general situations.

Finally we show that the nonprobabilistic notion of Nondeducibility-on-Strategies proposed in the literature is an instantiation of PNI, such that our results on PNI specialise to NOS.

Ultimate goal of this research is the mechanical verification of secure information flow. In a companion paper [Jür00] we express PNI in the framework of discrete Markov chains used for probabilistic modelchecking [dAKN+00] by making use of a new equivalent coinductive definition in a state machine setting, and provide further simplifications of checking PNI.

Due to space limitations we can only sketch the proofs; the complete proofs will appear in an extended version.

2 Probabilistic Models of Secure Information Flow

2.1 System Model

Before considering AFM and PNI we define the underlying trace-based model [Gra92] similar to dataflow models.

We consider concurrently executing probabilistic processes interacting by transmitting sequences of data values over unidirectional FIFO communication channels. Communication is asynchronous in the sense that transmission of a value cannot be prevented by the receiver. Technically, this means that processes are input-total (i. e. at each state the receiving of any input induces a state change) which is not a real restriction since one can model a "deadlocked" process by one that never outputs anything (arguably, this is the only property of a deadlocked process its environment can observe anyhow).

Processes can take two roles: that of a *system* whose security wrt. information flow should be examined and that of an *environment* that may try to exploit possible holes in the system's security.

The interface of a process is given by disjoint finite sets of input resp. output ports I resp. O (possibly empty, with union $C := I \cup O$). We write $\mathcal{E}_O$ for the set of functions (*output events*) $o : O \to \mathcal{D}_o$ and $\mathcal{E}_I$ for the *input events* $i : I \to \mathcal{D}_i$ (where $\mathcal{D}_o, \mathcal{D}_i$ are finite data sets containing the possible data values on the channels o, i, including an element ε representing absence of a value). For a system M with I (resp. O) the set of input (resp. output) ports we write $\mathcal{E}^M := (\mathcal{E}_O \times \mathcal{E}_I)^*$ (set of *system histories*, i. e. even-length alternating sequences of outputs and inputs), and for an environment E (with I, O as above) we write $\mathcal{E}^E := \mathcal{E}_I \times (\mathcal{E}_O \times \mathcal{E}_I)^*$ (set of *environment histories*, odd-length alternating sequences starting with input). Intuitively, the difference is that the system starts the interaction with the environment by outputting a value, and after that both sides take turns (the fact that the system starts simplifies the composition of systems defined later). This is a slight strengthening of the capabilities of the environment from [Gra92] necessary to ensure compositionality (cf. section 4). All other results in this paper do not depend on this.

A *system* is a triple $M = (I, O, P)$ with sets of input (resp. output) ports I (resp. O) and $P : \mathcal{E}_O \times \mathcal{E}^M \to [0, 1]$ such that for each system history $w \in \mathcal{E}^M$ we have $\sum_{o \in \mathcal{E}_O} P(o, w) = 1$.

An *environment* is a triple $E = (I, O, P)$ with I, O as above and $P : \mathcal{E}_O \times \mathcal{E}^E \to [0, 1]$ such that for each system history $w \in \mathcal{E}^E$ we have $\sum_{o \in \mathcal{E}_O} P(o, w) = 1$.

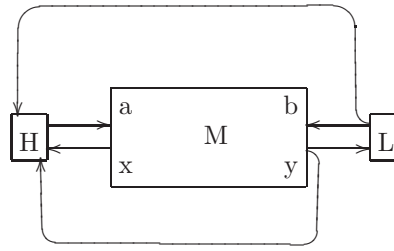
We write $P(o \mid w)$ for $P(o, w)$. $P(o|w)$ is the probability that after the history w the next output of the process is o. Say that a history has length t if it contains t outputs.

In the following we consider systems M with sets of input resp. output channels divided disjointly into low and high input resp. output channels: $I = I_l \cup I_h$, $O = O_l \cup O_h$. We write the high input values as a tuple a, and similarly the low input, high output and low output values as b, x and y, resp. .[1]

We assume the environment to consist of two independent components H and L such that M is connected via the high channels to H and via the low channels to L. The output from L and the low output from M are additionally given as input to H, but L cannot directly observe the values on the high channels (*secure environment criterion*). Thus $M = (I, O, P_M((x, y) \mid w))$ (w ranging

[1] Thus in the picture below a line represents a *tuple* of channels.

over $\mathcal{E}^M$), $L = (O_l, I_l, P_L(b \mid w \restriction_l))$ ($w \restriction_l$ the restriction of $w \in \mathcal{E}^{H,L}$ to the low channels) and $H = (O \cup I_l, I_h, P_H(a \mid w))$ (w in $\mathcal{E}^{H,L}$).



2.2 Applied Flow Model

We give the simpler verification condition [Gra92] equivalent to AFM.

Definition 1. *A system M satisfies the* Applied Flow Model *(AFM) if for all traces w, w' that are identical wrt. the low input and output, and for all low outputs y,*

$$\sum_x P_M((x, y) \mid w) = \sum_x P_M((x, y) \mid w').$$

This means that the probability of the next low output of M depends only on the *low* part of the previous input-/output-history.

AFM is relatively simple and implies PNI, but it is too strict: there are visibly secure systems (satisfying PNI), that do not satisfy AFM [Gra92]. Worse, below we show AFM to contradict the usual assumptions on the model.

The example S for a system that satisfies PNI but not AFM [Gra92] consists of a random number generator R whose output is directly passed on to the high output of S, and simultaneously is sent to a delay component D in S. D outputs any value received after one time step, and its output is connected to the low output of S. S is secure: since H has no way of influencing the low output of S, it cannot send information to L.

However, if we drop the one-step delay, i. e. if the low output receives the data at the same time as the high output, then we obtain a system S' that does satisfy AFM ! This is unrealistic since by definition of the Applied Flow Model L has state (i. e. memory), so in the case of S', L has all information it has in S, and even one step earlier.

The following obvious weakening of AFM still fails to capture all secure processes:

Definition 2. *A process M fulfils* weak AFM *if for all possible traces w and w' that are identical wrt. the low input and output* and the high output*, and all low-level output events y,*

$$\sum_x P_M((x, y) \mid w) = \sum_x P_M((x, y) \mid w').$$

Weak AFM correctly considers the example above to be secure. But the following secure process (that satisfies PNI) does not satisfy weak AFM. Let $\mathcal{D} = \{0, 1\}$. M starts by outputting $x_1 = 0$ resp. $x_1 = 1$ each with probability $1/2$ to H, and $y_1 = \varepsilon$ to L. Upon input of values a_1 (resp. b_1) from H (resp. L) M then outputs $x_2 = \varepsilon$ to H and y_2 to L depending on x_1:

$x_1 = 0$**:** $y_2 := a_1$ with probability $1/4$ and $\neg a_1$ with probability $3/4$
$x_1 = 1$**:** $y_2 := \neg a_1$ with probability $1/4$ and a_1 with probability $3/4$

receives 0 and 1 each with probability $1/2$, so there is no flow from H to L. But M does not satisfy weak AFM: $P(a_1 \mid (a_1, x_1 = 0)) = 1/4 \neq 3/4 = P(a_1 \mid (\neg a_1, x_1 = 0))$.

2.3 Probabilistic Noninterference

For a system M and environment H, L we define the probability $P^M_{H,L}(w)$ of a system history $w \in \mathcal{E}^M$ inductively by $P^M_{H,L}(\varepsilon) = 1$ and

$$P^M_{H,L}(w.(x,y).(a,b)) = P^M_{H,L}(w) \cdot P_M((x,y) \mid w) \cdot P_H(a \mid w.(x,y)) \cdot P_L(b \mid w\,\lfloor_l\,.y)$$

where . denotes concatenation (we leave out the superscript M when possible).

$P^M_{H,L}(w)$ gives the probability that the execution of the system M in the environment H, L results in w.

The probability $P^l_{H,L}(w_l)$ of a low system history w_l is

$$P^l_{H,L}(w_l) = \sum_{w : w\lfloor_l = w_l} P_{H,L}(w)$$

Definition 3. *A system M satisfies* Probabilistic Noninterference *if for all H, H', L and for all low system histories w_l*

$$P_{H,L}(w_l) = P_{H',L}(w_l).$$

This means that the probability of a low trace arising from the interaction between M with high and low processes only depends on the low process.

3 Deterministic Environments Are Sufficient

Definition 4. *A probabilistic process (I, O, P) is called* deterministic *if 0 and 1 are the only occurring probabilities, i. e. for all histories w and all output events o we have $P(o \mid w) \in \{0, 1\}$.*[2]

We now show that in fact in the definition of PNI it is sufficient to consider 'deterministic H, H', L:

[2] Thus exactly one output may occur at each point; in the following we specify deterministic processes by indicating these values.

Theorem 1 (Determinism). *A system M satisfies Probabilistic Noninterference iff for all deterministic H, H', L and for all low system histories w_l we have*

$$P_{H,L}(w_l) = P_{H',L}(w_l).$$

The reason is that each of M, H, L can only observe the values coming from the other components, and not directly the probability that certain values are sent. Therefore allowing H and L to send values with varying probabilities does not increase their capability to communicate through M since by construction probabilities are propagated only indirectly.

This theorem simplifies verification of PNI considerably and allows significantly simpler proofs of earlier results on PNI. Further results on simplified verification are given in [Jür00].

Proof. (of the Theorem)

Call a process (I, O, P) *n-deterministic* if its first n outputs are deterministically determined by the previous history, i. e. for all histories w of length $< n$ and all output events o we have $P(o \mid w) \in \{0, 1\}$.

Fix M and assume that for all deterministic H, H', L and all low system histories w^l, $P_{H,L}(w^l) = P_{H',L}(w^l)$. Given $w^l = (y_1, b_1, \ldots, y_t, b_t)$ we must show $P_{H,L}(w^l) = P_{H',L}(w^l)$ for all H, H', L. Define $w^l_s = (y_1, b_1, \ldots, y_s, b_s)$ for $s \leq t$. We show inductively for $n = t, \ldots, 0$ that $P_{H,L}(w^l) = P_{H',L}(w^l)$ for all n-deterministic H, H', L.

This is clear by assumption for $n = t$ since for any t-deterministic process one can find a deterministic one with the same behaviour at the first t outputs. Supposing the statement for $t \geq n > 0$ we prove it for $s = n - 1$: Suppose we are given s-deterministic H, H', L. Thus for each $r \leq s$ there exists $a(x_1, \ldots, x_r)$ such that $P_H(a(x_1, \ldots, x_r) \mid w(x_1, \ldots, x_{r-1}).(x_r, y_r)) = 1$ for $w(x_1, \ldots, x_{r-1}) = ((x_1, y_1), (a(x_1), b_1,), \ldots, (x_{r-1}, y_{r-1}), (a(x_1, \ldots, x_{r-1}), b_{r-1}))$. Similarly we have $a'(x_1, \ldots, x_r)$ for H'. By assumption on L we may assume $P_L(b_r \mid w^l_{r-1}.y_r) = 1$ for $r \leq s$. Thus

$$\begin{aligned}
&P_{H,L}(w^l)\\
&= \sum_{x_1,\ldots,x_s} P_M((x_1, y_1) \mid \varepsilon) \cdot \ldots \cdot P_M((x_s, y_s) \mid w(x_1, \ldots, x_{s-1}))\\
&\cdot \sum_{a_{s+1},x_{s+1}} P_M((x_{s+1}, y_{s+1}) \mid w(x_1, \ldots, x_s))\\
&\cdot P_H(a_{s+1} \mid w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})) \cdot P_L(b_{s+1} \mid w^l_s.y_{s+1})\\
&\cdot P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1}).(a_{s+1}, b_{s+1}))
\end{aligned}$$

with

$$P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t)|v) := \sum_{a_{s+2},x_{s+2},\ldots,a_t,x_t} P_{H,L}(v.(x_{s+2}, y_{s+2}).\ldots.(a_t, b_t))/P_{H,L}(v)$$

To derive a contradiction suppose $P_{H,L}(w^l) \neq P_{H',L}(w^l)$, so wlog. $P_{H,L}(w^l) > P_{H',L}(w^l)$. Construct $\tilde{L}$ from L by defining $P_{\tilde{L}}(b_{s+1} \mid w_s^l.y_{s+1}) = 1$ (and leaving the probabilities for time steps $\neq s+1$ unchanged). Then $P_{H,\tilde{L}}(w^l) > P_{H',\tilde{L}}(w^l)$ since $P_{H,\tilde{L}}(w^l) = P_{H,L}(w^l)/P_{\tilde{L}}(b_{s+1} \mid w_s^l)$ (and similarly for H'). For all $x_1, \ldots, x_{s+1}, a_{s+1}$,

$$\begin{aligned}&P_{H,\tilde{L}}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1}).(a_{s+1}, b_{s+1}))\\ &= P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1}).(a_{s+1}, b_{s+1}))\end{aligned}$$

since L and $\tilde{L}$ differ only at the $s+1$st output.

For each $(x_1, \ldots, x_{s+1})$ choose $a(x_1, \ldots, x_{s+1})$ such that

$$\begin{aligned}&\sum_{x_{s+1}} P_M((x_{s+1}, y_{s+1}) \mid w(x_1, \ldots, x_s))\\ &\cdot P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})\\ &.(a(x_1, \ldots, x_{s+1}), b_{s+1}))\end{aligned}$$

is maximal (this is possible since $\mathcal{D}$ is finite). Construct $\tilde{H}$ from H by setting $P_H(a(x_1, \ldots, x_{s+1}) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})) := 1$.

Similarly, for each $(x_1, \ldots, x_{s+1})$ choose $a'(x_1, \ldots, x_{s+1})$ such that

$$\begin{aligned}&\sum_{x_{s+1}} P_M((x_{s+1}, y_{s+1}) \mid w(x_1, \ldots, x_s))\\ &\cdot P_{H',L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})\\ &.(a'(x_1, \ldots, x_{s+1}), b_{s+1}))\end{aligned}$$

is minimal and construct $\tilde{H'}$ from H' by setting

$$P_{H'}(a'(x_1, \ldots, x_{s+1}) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})) := 1.$$

Then $P_{\tilde{H},\tilde{L}}(w^l) \geq P_{H,\tilde{L}}(w^l)$ because for every $x_1, \ldots, x_s, a_{s+1}$,

$$\begin{aligned}&\sum_{x_{s+1}} P_M((x_{s+1}, y_{s+1}) \mid w(x_1, \ldots, x_s))\\ &\cdot P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})\\ &.(a(x_1, \ldots, x_{s+1}), b_{s+1}))\\ &\geq \sum_{x_{s+1}} P_M((x_{s+1}, y_{s+1}) \mid w(x_1, \ldots, x_s))\\ &\cdot P_{H,L}((b_{s+2}, y_{s+2}), \ldots, (b_t, y_t) | w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1}).(a_{s+1}, b_{s+1}))\end{aligned}$$

by definition of $a(x_1, \ldots, x_{s+1})$ and since

$$\begin{aligned}&\sum_{a_{s+1}} P_H(a_{s+1} \mid w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1})) = 1\\ &= P_H(a(x_1, \ldots, x_{s+1}) \mid w(x_1, \ldots, x_s).(x_{s+1}, y_{s+1}))\end{aligned}$$

Similarly $P_{\tilde{H'},\tilde{L}}(w^l) \leq P_{H,\tilde{L}}(w^l)$. Thus $P_{\tilde{H},\tilde{L}}(w^l) > P_{\tilde{H'},\tilde{L}}(w^l)$ contradicting the inductive assumption. □

4 Compositionality

Definition 5. *The* composition $M \otimes N$ *of two systems* M, N *with* $O_M \cap O_N = I_M \cap I_N = \emptyset$ *(which can always be achieved by renaming) is defined as follows:*

- $O_{M\otimes N} = O_M \cup O_N$
- $I_{M\otimes N} = I_M \cup I_N \setminus O_{M\otimes N}$
- $P^{M\otimes N}(o \otimes p|v \otimes w) = P^M(o|v) \cdot P^N(p|w)$ *for* compatible *histories* v, w.

Here $e \otimes f := (e \cup f)$ *for all output events* $e : O_M \to \mathcal{D}$, $f : O_N :\to \mathcal{D}$ *where the union of two functions is the union of the underlying relations (which by disjointness of the channel sets is a function). For input events* $e : I_M \to \mathcal{D}$, $f : I_N :\to \mathcal{D}$, $e \otimes f := (e \cup f)\restriction_{I_{M\otimes N}}$. *The tensor of histories is defined elementwise:* $(v_1, \ldots, v_t) \otimes (w_1, \ldots, w_t) = (v_1 \otimes w_1, \ldots, v_t \otimes w_t)$. *Histories* v, w *of the same length are* compatible *if any value on an output channel appears at the input channel connected to it (if any) at the next time step.*

Note that the above composition makes the model essentially a probabilistic version of I/O-systems [Jon87]. As usual in models of dataflow, noncommunicating parallel composition and feedback are instantiations of the general composition.

In the following we always require that systems are only composed in a way such that low (resp. high) output channels are connected to low (resp. high) input channels.

The following example[3] of a system M shows why the strengthening of the environment is needed to ensure compositionality: M has two input channels A_1, A_2 (both high), a high output channel X and a low output channel Y. M is supposed to encrypt the high input on A_1 with the key provided on A_2 and then with an internally randomly generated one-time key pad and send the result out on Y. The random number should also be output on X, but so that the high input on A_i cannot depend on it.

In the model in [Gra92], the system and the environment output values in parallel (not in turn) each independently of the input received from the other party at the same time. Then the following system meets the above description: At time 1, M inputs binary values a_1, a_2 on D_1, D_2, resp. , and outputs a random binary value x on X. At time 2, M outputs $y := a_1 \oplus a_2 \oplus x$ on Y (where $\oplus$ is exclusive or). Then M is considered secure since $a_1 \oplus a_2$ does not depend on x and so y is 0 or 1 each with probability 1/2 independent of the high process H (assuming a true random generator).

But the system derived from M by feedback of X into A_2 is not secure: it takes a value on A_1 at time 1 and outputs the same value on Y at time 2.

The model considered in this paper is derived from the one in [Gra92] by allowing the output of the environment to depend on input received at the same time (this is the *worst case assumption* that the environment could have a substantially faster reaction). Then M is correctly considered insecure.

[3] A variation of an example in [Sha58].

Theorem 2 (Compositionality). *If M and N satisfy PNI, so does $M \otimes N$.*

The proof of this theorem uses a rather general principle in a rely-guarantee setting inspired by [AL93] which is given below.

Their result on composing rely-guarantee specifications does not apply directly. Already in the nonprobabilistic case the information flow properties are sets of sets of traces [McL96], and not just sets of traces, as considered in [AL93]. The earlier approaches to compositionality of security properties including [McL96] only consider the nonprobabilistic case and so do not apply here.

The general setting is the following: $\mathcal{S}$ is a set of "systems" and $\mathcal{E}$ of "environments" such that $\mathcal{S} \subseteq \mathcal{E}$ (any system can be the environment of another system). $\otimes$ is a binary composition on $\mathcal{E}$ restricting to $\mathcal{S}$. For each $t \in \mathbb{N}_{\geq 0}$ there are properties ϕ_t on environments (i. e. $\phi_t \subseteq \mathcal{E}$) and σ_t on systems relative to environments ($\sigma_t \subseteq \mathcal{S} \times \mathcal{E}$) with $\phi_0 = \mathcal{E}$ and $\sigma_0 = \mathcal{S} \times \mathcal{E}$. If $(M, E) \in \sigma_t$ say "M guarantees σ_t when placed into E". If M guarantees σ_t when placed into any E satisfying ϕ_s we say "M guarantees σ_t assuming ϕ_s". Write ϕ for $\bigwedge_t \phi_t$ and σ for $\bigwedge_t \sigma_t$.

We also suppose that the rely-guarantee condition "guarantee σ assuming ϕ" can be obtained using the "approximations" "guarantee σ_t assuming ϕ_t": For any system M and any t, if M guarantees σ_{t+1} assuming ϕ_{t+1} then M guarantees σ_t assuming ϕ_t. Also M guarantees σ assuming ϕ iff for all t, M guarantees σ_t assuming ϕ_t.

Principle 1 (Inductive Compositionality) *Suppose that for each t:*

a) *For any E, if E satisfies ϕ_{t+1} and M guarantees σ_t assuming ϕ_t then $E \otimes M$ satisfies ϕ_{t+1} provided "σ_t for ϕ_t" is compositional (i. e. provided for each M_1, M_2, if each M_i guarantees σ_s assuming ϕ_s, then $M_1 \otimes M_2$ guarantees σ_s assuming ϕ_s).*

b) *For each E, M_1, M_2: $M_1 \otimes M_2$ guarantees σ_t in the environment E if*
- *M_1 guarantees σ_t in $E \otimes M_2$ and*
- *M_2 guarantees σ_t in $E \otimes M_1$.*

Then the following holds:

If M_1 and M_2 guarantee σ assuming ϕ then $M_1 \otimes M_2$ guarantees σ assuming ϕ.

The proof of the principle has to be omitted

Proof. (of the Theorem) Space permits only to give an idea how to express PNI using this principle. The idea is to weaken the "secure environment criterion" by requiring the environment itself only to fulfil (essentially) PNI, but only up to a specified length of traces. The circularity is broken since the definition of "PNI up to trace length t" on a system only requires assuming "PNI up to trace length $t-1$" on the environment, because of delay in the communication. To formulate PNI, the elements of $\mathcal{E}$ have to be finite sets of environments H, L considered above. □

5 Probabilistic vs. Possibilistic

The Determinism theorem allows us to relate PNI to the nonprobabilistic property *Nondeducibility-on-Strategies* [WJ90].

The model used here is derived from the probabilistic one by disregarding probabilities. The behaviour of a process $(I, O, \mathcal{H})$ is given by its (prefix-closed) set of histories $\mathcal{H}$.

Even though the nondeterminism in this model is technically the same as the usual ("don't care") nondeterminism, the interpretation is different: In the "don't care" case one usually considers properties of traces independent of the existence of other traces. Here the nondeterminism is used to model specific security mechanisms like encryption where the relevant properties do not just concern an executed trace, but have to consider the traces that *could* have been executed instead. Noninterference properties do not concern traces but sets of traces [McL96] and one cannot use the usual refinement by reverse trace set inclusion. We refer to such a use of nondeterminism as *possibilistic nondeterminism.*

We define NOS: A *strategy* for H is a function π that assigns to every environment history $w \in \mathcal{E}_E$ a high input event $a = \pi(w)$. Thus a strategy defines a deterministic high system H and vc. vs. .

A history $(x_1, y_1), (a_1, b_1,), \ldots, (a_t, b_t)$ is *compatible* with a strategy π if for each $s \leq t$, $\pi((x_1, y_1), (a_1, b_1,), \ldots, (x_s, y_s)) = a_s$, i. e. π behaves according to the history. Thus a history w of M is compatible with π iff it is an element of the set of possible traces of the execution of M composed with the H corresponding to π and any deterministic L behaving according to $w \restriction_l$ (i. e. an L that outputs b_s when the previous history is $y_1, b_1, \ldots, y_s$). Call a low history w_l *compatible* with π if there is a history w with $w \restriction_l = w_l$ such that w is compatible with π, and w_l is *possible* if there is a strategy with which it is compatible.

Definition 6. *A system M satisfies* Nondeducibility-on-Strategies *if for any strategy π and any possible low history w_l there is a history w compatible with π such that $w \restriction_l = w_l$ for its low view $w \restriction_l$.*

Intuitively, L cannot observe which strategy H follows.

One can "embed" possibilistic nondeterminism into probabilistic nondeterminism in a natural way as follows: For any possibilistic system M construct a probabilistic system $\hat{M}$ by assigning to all outputs possible at a given point the same (nonzero) probabilities: if for a possible history w the next possible outputs from M are $o_1, \ldots, o_n$, define $P^M(o_i|w) = 1/n$ for each i.

Conversely, from any probabilistic system M one can derive a possibilistic system $|M|$ by "forgetting" the probabilities: each history is possible in $|M|$ if it has nonzero probability in M.

Note that while $|\hat{M}| = M$ always holds, $\widehat{|M|}$ is the system derived from M by "levelling out" the nonzero probabilities at a given state (i. e. assigning them uniform probabilities) and thus in general not equal to M.

We say that a possibilistic system M satisfies PNI if $\hat{M}$ does. Then the Determinism theorem implies that for possibilistic systems, PNI reduces to NOS:

Theorem 3. a) *A possibilistic system M satisfies PNI iff it satisfies NOS.*

b) *If a probabilistic system M satisfies PNI then its "nonprobabilistic quotient" $|M|$ satisfies PNI (and thus NOS).*

Part a) means that, if the system M under consideration is itself nonprobabilistic, then NOS is fully sufficient, even if the systems H and L that M is connected with are probabilistic. On the other hand, this result allows to pass over to the possibilistic case whenever the probabilistic one is too complex to verify. Also, the above results obtained on PNI thus specialise to NOS.

Note that the implication in b) is not reversible: Let the system M receive a high value a and, with probability $1/2$, output it to L, while also with probability $1/2$ it outputs a random value to L. $|M|$ satisfies NOS (and the other nonprobabilistic information flow notions), but PNI correctly considers M to be insecure, because there is a channel from H to L (even if it is not *noiseless*).

The implication "M PNI $\Rightarrow |M|$ NOS" has been observed without giving a proof in [Dut99].

Proof. (of the Theorem) Let M be a possibilistic system.

a) $\Rightarrow$: Suppose M satisfies PNI and we are given a strategy π and a possible low history $w_l = (b_1, y_1), \ldots, (b_t, y_t)$. We need to show that there is a history w compatible with π such that $w \downharpoonright_l = w_l$ for its low view $w \downharpoonright_l$. Since w_l is possible by assumption, there must be a history w' and a strategy π' such that w' is compatible with π' and $w \downharpoonright_l = w_l$. Let H (resp. H') be defined by π (resp. π') and let L output $b_1, \ldots, b_t$ regardless of the y_i. By definition of PNI and of the translation $\hat{M}$ to the probabilistic setting w_l is possible in an execution of M composed with H and L (i. e. the corresponding trace set contains a trace w with $w \downharpoonright_l = w_l$) iff it is possible when composed with H' and L. But the first part of the equivalence holds by assumption on π', and the second is what we needed to show.

$\Leftarrow$: Suppose M satisfies NOS. By the Determinism Theorem and the definition of $\hat{M}$ it is sufficient to show that given deterministic systems H, H', L, each low history w_l is possible in an execution of M composed with H and L iff it is possible when composed with H' and L. By symmetry, it is sufficient to show for each H, H', L, w_l, if w_l is possible with H, L, then also with H', L. Suppose it is possible with H, L. This implies that w_l is possible and that L behaves according to w_l. Then by definition of NOS, for the strategy corresponding to H' there exists a history w compatible with π' such that $w \downharpoonright_l = w_l$. But this means that w_l is possible with H', L.

b) Since $\widehat{|M|}$ is the system derived from M by at each state assigning equal probabilities to the output events with nonzero probability in M, the proof only requires the observation that if two probabilities are equal (such as in the definition of PNI) then they are in particular both nonzero or both zero. □

6 Related Work

Goal of our work is the verification of specifications of secure information flow, including notions that allow detection of covert channels arising from statistical inference.

In recent years there have been reformulations of nonprobabilistic information flow notions in process algebras which provide mechanical support. Thus one could aim to achieve the goal by trying to extend these frameworks with probabilistic aspects. We consider the approaches to information flow using CSP and CCS, process algebras which have been very successful in other areas of concurrency (e. g. deadlock-detection) and security (e. g. cryptographic protocols).

Since the natural translation of noninterference into CSP is not easily addressable by its model checker and to avoid the "refinement paradox" [RWW94,Ros95] introduces the approach of "noninterference through determinism" which however is too restrictive according to [Low99] who points out that the other definitions up to that date for process algebras were also too weak or too strong. He argues that standard models of CSP are not sufficient to treat information flow since they do not make enough distinctions, and offers a solution using a non-standard model of CSP. However, he concludes that "the model is moderately complicated, but I suspect that this is inevitable". He also points out weaknesses of existing probabilistic process algebras.

For the CCS approach, an extensive comparison [FG94] brought forward a reformulation of NOS called BNDC as the preferred notion [FG97]. [Foc96] shows that it is under suitable conditions equivalent to the one in [Ros95], which extends the critique cited above to this approach.

The bi-directional "handshake" communication common to both process algebras requires care, since one wants to allow low behaviour to interfere with high behaviour, but not the other way [RG99].

According to [RS99] there is no consensus yet which notion of noninterference is "correct", thus it seems profitable to examine various notions, including the non-process-algebraic ones presented here.

7 Conclusion and Further Work

After demonstrating weaknesses of the alternative notion AFM we have given several important results on Probabilistic Noninterference towards practical verification. This includes the result that in PNI one needs only quantify over deterministic environments, the proof of compositionality of PNI (an open problem from [Gra92]) - using a new Inductive Compositionality Principle in a rely-guarantee framework - and the proof that the nonprobabilistic notion Nondeducibility-on-Strategies proposed in the literature is an instantiation of PNI.

In a companion paper [Jür00] we express PNI in the framework of discrete Markov chains used for probabilistic modelchecking [dAKN+00] by making use of a new equivalent coinductive definition in a state machine setting, and we provide further results towards simplification of PNI. Further work needs to be

done wrt. efficiency and practicality of using PNI. Also we have been considering a notion of *secure refinement* inspired by [AHKV98,Abr99].

Current work includes extending the model considered here with primitives for symmetric encryption and giving a computational interpretation following the approach of [AR00].

In another direction, since sometimes covert channels cannot be avoided completely, one should consider relaxing the conditions towards accepting small influences of high behaviour on low probabilities, possibly along the lines of [DGJP99].

Acknowledgements

Many thanks for interesting comments go to Samson Abramsky, Bruno Dutertre (also for making [Dut99] available), Riccardo Focardi, Dusko Pavlović, and furthermore to Peter Ryan for providing the opportunity to attend the workshop on Information Flow (London, Dec. 1999). Helpful comments by several anonymous referees are acknowledged.

Part of this research was done during a visit to Kestrel Institute (Palo Alto, CA) in January 2000 whose hospitality is gratefully acknowledged.

References

Abr99. S. Abramsky. A note on reactive refinement, 1999. Manuscript. 408

AHKV98. Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Verlag, 1998. 408

AL93. M. Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems 15*, 1:73–132, January 1993. 404

AR00. M. Abadi and P. Rogaway. Reconciling two views of cryptography (invited lecture). In *TCS 2000*, August 2000. 408

dAKN+00. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In *TACAS'2000*, Lecture Notes in Computer Science, January 2000. 397, 407

DGJP99. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panagaden. Metrics for labeled markov systems. In *CONCUR*, 1999. 408

DS99. Bruno Dutertre and Victoria Stavridou. A model of noninterference for integrating mixed-criticality software components. In *DCCA-7, Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, San Jose, CA, January 1999. 396

Dut99. B. Dutertre. State of the art in secure noninterference, 1999. Manuscript. 406, 408

FG94. R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security, 3(1):5-33*, 1994. 407

FG97. R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transaction of Software Engineering*, 23(9), September 1997. 397, 407

Foc96. R. Focardi. Comparing two information flow security properties. In *Proceeding of the 9th IEEE Computer Security Foundation Workshop*, pages 116–122, June 1996. 407

GM82. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982. 396

Gra92. J. W. Gray. Toward A Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 3–4(1):255–294, 1992. 396, 397, 398, 399, 403, 407

Jon87. B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, Department of Computer Systems, Uppsala University, 1987. Tech. Rep. DoCS 87/09. 403

Jür00. J. Jürjens. Verification of probabilistic secure information flow, 2000. submitted. 397, 401, 407

Low99. Gavin Lowe. Defining information flow. Technical report, Department of Mathematics and Computer Science Technical Report 1999/3, University of Leicester, 1999. 397, 407

McL90. J. McLean. Security Models and Information Flow. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 180–187, Oakland, CA, May 1990. 396

McL94. J. McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, Inc., 1994. 397

McL96. John D. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, January 1996. 397, 404, 405

RG99. A. Roscoe and M. Goldsmith. What is intransitive noninterference ? In *Proceedings of the Computer Security Foundations Workshop of the IEEE Computer Society (CSFW)*, 1999. 397, 407

Ros95. A.W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, 1995. 397, 407

RS99. P. Ryan and S. Schneider. Process algebra and non-interference. In *Proceedings of the* 12^{th} *Computer Security Foundations Workshop of the IEEE Computer Society (CSFW 12)*, 1999. 397, 407

RWW94. A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through determinism. In *ESORICS*, 1994. 407

Sha58. C. Shannon. Channels with side information at the transmitter. *IBM Journal of Research and Development*, 2:289–293, 1958. 403

WJ90. J. T. Wittbold and D. M. Johnson. Information Flow in Nondeterministic Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA, May 1990. 396, 405

LP Deadlock Checking Using Partial Order Dependencies

Victor Khomenko and Maciej Koutny

Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
{Victor.Khomenko,Maciej.Koutny}@ncl.ac.uk

Abstract. Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the ways to exploit such a semantics is to consider (finite prefixes of) net unfoldings — themselves a class of acyclic Petri nets — which contain enough information, albeit implicit, to reason about the reachable markings of the original Petri nets. In [15], a verification technique for net unfoldings was proposed in which deadlock detection was reduced to a mixed integer linear programming problem. In this paper, we present a further development of this approach. We adopt Contejean-Devie's algorithm for solving systems of linear constraints over the natural numbers domain and refine it, by taking advantage of the specific properties of systems of linear constraints to be solved. The essence of the proposed modifications is to transfer the information about causality and conflicts between the events involved in an unfolding, into a relationship between the corresponding integer variables in the system of linear constraints. Experimental results demonstrate that the new technique achieves significant speedups.

1 Introduction

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use model checking [4]. The main drawback of model checking is that it suffers from the state explosion problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To help in coping with this, a number of techniques have been proposed which can roughly be classified as aiming at an implicit compact representation of the full state space of a reactive concurrent system, or at an explicit representation of a reduced (though sufficient for a given verification task) state space of the system. Techniques aimed at reduced representation of state spaces are typically based on the independence (commutativity) of some actions, often relying on the partial order view of concurrent computation. Such a view is the basis for algorithms employing

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 410–425, 2000.

McMillan's unfoldings ([10,11,14]), where the entire state space of a Petri Net is represented implicitly using an acyclic net to represent a system's actions and local states. The net unfolding technique presented in [14,15] reduces memory requirement, but the deadlock checking algorithms proposed are quite slow, even for medium-size unfoldings.

In [15], the problem of deadlock checking a Petri net was reduced to a mixed integer linear programming problem. In this paper, we present a further development of this approach. We adopt the Contejean-Devie's algorithm ([1,2,5,6,7]) for efficiently solving systems of linear constraints over the domain of natural numbers. We refine this algorithm by employing unfolding-specific properties of the systems of linear constraints to be solved in model checking aimed at deadlock detection. The essence of the proposed modifications is to transfer the information about causality and conflicts between events involved in an unfolding into a relationship between the corresponding integer variables in the system of linear constraints. The results of initial experiments demonstrate that the new technique achieves significant speedups.

The paper is organised as follows. In section 2 we provide basic definitions concerning Petri nets and, in particular, net unfoldings. Section 3 briefly recalls the results presented in [15] where the deadlock checking problem has been reduced to the feasibility test of a system of linear constraints. Section 4 is based on the results developed in [1,2,5,6,7] and recalls the main aspects of the Contejean-Devie's Algorithm (CDA) for solving systems of linear constraints over the natural numbers domain. The algorithm we propose in this paper is a variation of CDA, developed specifically to exploit partial order dependencies between events in the unfolding of a Petri net. Our algorithm is described in section 5; we provide theoretical background, useful heuristics, as well as outlining ways of reducing the number of variables and constraints in the original system presented in [15]. Section 6 contains results of experiments obtained for a number of benchmark examples, while section 7 describes possible directions for future research. The proofs of the results can be found in [12].

2 Basic Definitions

In this section, we first present basic definitions concerning Petri nets, and then recall (see [10]) notions related to net unfoldings.

Petri Nets A *net* is a triple $N = (S, T, F)$ such that S and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e. $M : S \to \mathbb{N} = \{0, 1, 2, \ldots\}$. As usual, we will denote ${}^\bullet z = \{y \mid (y, z) \in F\}$ and $z^\bullet = \{y \mid (z, y) \in F\}$, for all $z \in S \cup T$. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma = (N, M_0)$ comprising a finite net $N = (S, T, F)$ and an (initial) marking M_0. A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if for every $s \in {}^\bullet t$, $M(s) \geq 1$. Such a transition can be *executed*, leading to a marking M' defined by $M' = M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$ or

$M[\rangle M'$. The set of *reachable* markings of Σ is the smallest (w.r.t. set inclusion) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[\rangle M'$ then $M' \in [M_0\rangle$. For a finite sequence of transitions, $\sigma = t_1 \dots t_k$, we denote $M_0[\sigma\rangle M$ if there are markings $M_1, \dots, M_k$ such that $M_k = M$ and $M_{i-1}[t_i\rangle M_i$, for $i = 1, \dots, k$.

A marking is *deadlocked* if it does not enable any transitions. The net system Σ is *deadlock-free* if no reachable marking is deadlocked; *safe* if for every reachable marking M, $M(S) \subseteq \{0, 1\}$; and *bounded* if there is $k \in \mathbb{N}$ such that $M(S) \subseteq \{0, \dots, k\}$, for every reachable marking M.

Marking Equation Let $\Sigma = (N, M_0)$ be a net system, and $S = \{s_1, \dots, s_m\}$ and $T = \{t_1, \dots, t_n\}$ be sets of its places and transitions, respectively. We will often identify a marking M of Σ with a vector $M = (\mu_1, \dots, \mu_m)$ such that $M(s_i) = \mu_i$, for all $i \leq m$. The *incidence matrix* of Σ is an $m \times n$ matrix $\mathcal{N} = (\mathcal{N}_{ij})$ such that, for all $i \leq m$ and $j \leq n$,

$$\mathcal{N}_{ij} = \begin{cases} 1 & \text{if } s_i \in t_j^\bullet \setminus {}^\bullet t_j \\ -1 & \text{if } s_i \in {}^\bullet t_j \setminus t_j^\bullet \\ 0 & \text{otherwise .} \end{cases}$$

The *Parikh vector* of a finite sequence of transitions σ is a vector $x_\sigma{=}(x_1, \dots, x_n)$ such that x_i is the number of the occurrences of t_i within σ, for every $i \leq n$. One can show that if σ is an execution sequence such that $M_0[\sigma\rangle M$ then $M = M_0 + \mathcal{N} \cdot x_\sigma$. This provides a motivation for investigating the feasibility (or solvability) of the following system of equations:

$$\begin{cases} M = M_0 + \mathcal{N} \cdot x \\ M \in \mathbb{N}^m \text{ and } x \in \mathbb{N}^n \end{cases}$$

If we fix the marking M, then the feasibility of the above system is a necessary condition for M to be reachable from M_0.

Branching Processes Two nodes of a net $N = (S, T, F)$, y and y', are *in conflict*, denoted by $y \# y'$, if there are distinct transitions $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *self-conflict* if $y \# y$.

An *occurrence net* is a net $ON = (B, E, G)$ where B is the set of *conditions* (places) and E is the set of *events* (transitions). It is assumed that: ON is acyclic (i.e. $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of G. $Min(ON)$ will denote the minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation*. Two nodes are *co-related*, denoted by $y \; co \; y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to S \cup T$ such that: $h(B) \subseteq S$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$; the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$; the restriction of h to $Min(ON)$ is

a bijection between $Min(ON)$ and M_0; and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$.

A *branching process* of Σ [9] is a quadruple $\pi = (B, E, G, h)$ such that (B, E, G) is an occurrence net and h is a homomorphism from ON to Σ. A branching process $\pi' = (B', E', G', h')$ of Σ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted by $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ.

Configurations and Cuts A *configuration* of an occurrence net ON is a set of events C such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. A *cut* is a maximal w.r.t. set inclusion set of conditions B' such that $b \; co \; b'$, for all $b, b' \in B'$. Every marking reachable from $Min(ON)$ is a cut.

Let C be a finite configuration of a branching process π. Then $Cut(C) = (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset of places $h(Cut(C))$ is a reachable marking of Σ, denoted $Mark(C)$. A marking M of Σ is *represented* in π if the latter contains a finite configuration C such that $M = Mark(C)$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

A branching process π of Σ is *complete* if for every reachable marking M of Σ, there is a configuration C in π such that $Mark(C) = M$, and for every transition t enabled by M, there is a configuration $C \cup \{e\}$ such that $e \notin C$ and $h(e) = t$.

Although, in general, an unfolding is infinite, for every bounded net system Σ one can construct a finite complete prefix Unf_Σ of the unfolding of Σ. Moreover, there are *cut-off* events[1] in Unf_Σ such that, for every reachable marking M of Σ, there exists a configuration C in Unf_Σ such that $M = Mark(C)$ and no event in C is a cut-off event.

3 Deadlock Detection Using Linear Programming

In the rest of this paper, we will assume that $Unf_\Sigma = (B, E, G, h)$ is a finite complete prefix of the unfolding of a bounded net system $\Sigma = (S, T, F, M_0)$. We will denote by M_{in} the canonical initial marking of Unf_Σ which places a single token in each of the minimal conditions and no token elsewhere. Furthermore, we will assume that $b_1, b_2, \ldots, b_p$ and $e_1, e_2, \ldots, e_q$ are respectively the conditions and events of Unf_Σ, and that $\mathcal{C}$ is the $p \times q$ incidence matrix of Unf_Σ. The set of cut-off events of Unf_Σ will be denoted by E_{cut}.

We now recall the main results from [15]. A finite and complete prefix Unf_Σ may be treated as an acyclic safe net system with the initial marking M_{in}. Each

[1] Intuitively, cut-off events are nodes at which the potentially infinite unfolding may be cut without losing any essential information about the behaviour of Σ; see [9,10,11,14,15] for details.

reachable deadlocked marking in Σ is represented by a deadlocked marking in Unf_Σ. However, some deadlocked markings of Unf_Σ lie beyond the cut-off events and may not correspond to deadlocks in Σ. Such deadlocks can be excluded by prohibiting the cut-off events from occurring.

Since for an acyclic Petri net the feasibility of the marking equation is a sufficient condition for a marking to be reachable, the problem of deadlock checking can be reduced to the feasibility test of a system of linear constraints.

Theorem 1. *([15]) Σ is deadlock-free if and only if the following system has no solution (in M and x):*

$$\begin{cases} M = M_{in} + \mathcal{C} \cdot x & \\ \sum_{b \in {}^\bullet e} M(b) \leq |{}^\bullet e| - 1 & \text{for all } e \in E \\ x(e) = 0 & \text{for all } e \in E_{cut} \\ M \in \mathbb{N}^p \text{ and } x \in \mathbb{N}^q & \end{cases} \tag{1}$$

where $x(e_i) = x_i$, for every $i \leq q$.

In order to decrease the number of integer variables, $M \geq \mathbf{0}$ can be treated as a rational vector since $x \in \mathbb{N}^q$ and $M = M_{in} + \mathcal{C} \cdot x \geq \mathbf{0}$ always imply that $M \in \mathbb{N}^p$. Moreover, as an event can occur at most once in a given execution sequence of Unf_Σ from the initial marking M_{in}, it is possible to require that x be a binary vector, $x \in \{0,1\}^q$.

To solve the resulting mixed-integer LP-problem, [15] used the general-purpose LP-solver CPLEX [8], and demonstrated that there are significant performance gains if the number of cut-off events is relatively high since all variables in x corresponding to cut-off events are set to 0.

4 Solving Systems of Linear Constraints

In this paper, we will adapt the approach proposed in [1,2,5,6,7], in order to solve Petri net problems which can be reformulated as LP-problems. We start by recalling some basic results.

The original *Contejean and Devie's algorithm (CDA)* [5,6,7] solves a system of linear homogeneous equations with arbitrary integer coefficients

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q = 0 \\ a_{21}x_1 + \cdots + a_{2q}x_q = 0 \\ \quad \vdots \qquad\qquad\quad \vdots \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q = 0 \end{cases} \tag{2}$$

or $\mathcal{A} \cdot x = \mathbf{0}$ where $x \in \mathbb{N}^q$ and $\mathcal{A} = (a_{ij})$. For every $1 \leq j \leq q$, let

$$\varepsilon_j = (\underbrace{0, \ldots, 0}_{j-1 \text{ times}}, 1, 0, \ldots, 0)$$

be the j-th vector in the canonical basis of $\mathbb{N}^q$. Moreover, for every $x \in \mathbb{N}^q$, let $a(x) \in \mathbb{N}^p$ be a vector defined by

$$a(x) = \begin{pmatrix} a_{11}x_1 + \cdots + a_{1q}x_q \\ a_{21}x_1 + \cdots + a_{2q}x_q \\ \vdots \qquad\qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \end{pmatrix} = x_1 \cdot a(\varepsilon_1) + \cdots + x_q \cdot a(\varepsilon_q)\,, \tag{3}$$

where $a(\varepsilon_j)$ — the j-th column vector of the matrix $\mathcal{A}$ — is called the *j-th basic default vector*.

The set $\mathcal{S}$ of all solutions of (2) can be represented by a finite basis $\mathcal{B}$ which is the minimal (w.r.t. set inclusion) subset of $\mathcal{S}$ such that every solution is a linear combination with non-negative integer coefficients of the solutions in $\mathcal{B}$. It can be shown that $\mathcal{B}$ comprises all solutions in $\mathcal{S}$ different from the trivial one, $x = \mathbf{0}$, which are minimal with respect to the $\leq$ ordering on $\mathbb{N}^q$ ($x \leq x'$ if $x_i \leq x'_i$, for all $i \leq q$; moreover, $x < x'$ if $x \leq x'$ and $x \neq x'$).

The representation (3) suggests that any solution of (2) can be seen as a multiset of default vectors whose sum is $\mathbf{0}$. Choosing an arbitrary order among these vectors amounts to constructing a sequence of default vectors starting from, and returning to, the origin of $\mathbb{Z}^p$. CDA constructs such a sequence step by step: starting from the empty sequence, new default vectors are added until a solution is found, or no minimal solution can be obtained. However, different sequences of default vectors may correspond to the same solution (up to permutation of vectors). To eliminate some of the redundant sequences, a restriction for choosing the next default vector is used.

> A vector $x \in \mathbb{N}^q$ (corresponding to a sequence of default vectors) such that $a(x) \neq \mathbf{0}$ can be incremented by 1 on its j-th component provided that $a(x + \varepsilon_j) = a(x) + a(\varepsilon_j)$ lies in the half-space containing $\mathbf{0}$ and delimited by the affine hyperplane perpendicular to the vector $a(x)$ at its extremity when originating from $\mathbf{0}$ (see figure 1).

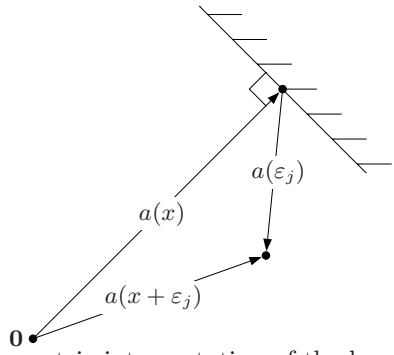


Fig. 1. Geometric interpretation of the branching condition in CDA

This reflects a view that $a(x)$ should not become too large, hence adding $a(\varepsilon_j)$ to $a(x)$ should yield a vector $a(x+\varepsilon_j) = a(x) + a(\varepsilon_j)$ 'returning to the origin'. Formally, this restriction can be expressed by saying that given $x = (x_1, \ldots, x_q)$,

$$\text{increment by 1 an } x_j \text{ satisfying } a(x) \odot a(\varepsilon_j) < 0\ , \tag{4}$$

where $\odot$ denotes the scalar product of two vectors. This reduces the search space without losing any minimal solution, since every sequence of default vectors which corresponds to a solution can be rearranged into a sequence satisfying (4).

Theorem 2. *([7]) The following hold for the CDA shown in figure 2:*

1. *Every minimal solution of the system (2) is computed.* *(completeness)*
2. *Every solution computed by CDA is minimal.* *(soundness)*
3. *The algorithm always terminates.* *(termination)*

- search breadth-first a directed acyclic graph rooted at $\varepsilon_1, \ldots, \varepsilon_q$
- **if** a node y is equal to, or greater than, an already found solution of $\mathcal{A} \cdot x = \mathbf{0}$ **then** y is a terminal node
- **otherwise** construct the sons of y by computing $y+\varepsilon_j$ for each $j \leq q$ satisfying $a(y) \odot a(\varepsilon_j) < 0$

Fig. 2. CDA (breath-first version)

But this algorithm may perform redundant calculations as some vectors can be computed more than once. This can be remedied by using *frozen components*, defined thus. Assume that there is a total ordering $\prec_x$ on the sons of each node x of the search graph constructed by CDA.

> If $x+\varepsilon_i$ and $x+\varepsilon_j$ are two distinct sons of a node x such that $x+\varepsilon_i \prec_x x+\varepsilon_j$, then the i-th component is *frozen* in the sub-graph rooted at $x+\varepsilon_j$ and cannot be incremented even if (4) is satisfied.

The modified algorithm is still complete ([7]), and builds a forest which is a sub-graph of the original search graph.

The ordered version of CDA can be extended to handle bounds imposed on variables and homogeneous systems of equations and inequalities (see [1,2,5,6,7]). Moreover, it can be adapted to solve non-homogeneous systems of inequalities

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q \leq d_1 \\ a_{21}x_1 + \cdots + a_{2q}x_q \leq d_2 \\ \quad\vdots \qquad\qquad\quad \vdots \qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \leq d_p \end{cases} \tag{5}$$

5 An Algorithm for Deadlock Detection

The MIP problem obtained in section 3 can be reduced to a pure integer one, by substituting the expression for M given by the marking equation into other constraints and, at the same time, reducing the total number of constraints. Each equation in $M = M_{in} + \mathcal{C} \cdot x$ has the form

$$M(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \qquad \text{for } b \in B. \tag{6}$$

After substituting these into (1) we obtain the system

$$\begin{cases} \displaystyle\sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) & \text{for all } e \in E \\ M_{in}(b) + \displaystyle\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \geq 0 & \text{for all } b \in B \\ x \in \{0,1\}^q \text{ and } x(e) = 0 \text{ for all } e \in E_{cut} \end{cases} \tag{7}$$

As (7) is a pure integer problem, CDA is directly applicable. However, since the number of variables can be large, it needs further refinement.

In [15], a finite prefix of the unfolding is used only for building a system of constraints, and the latter is then passed to the LP-solver without any additional information. Yet, during the solving of the system, one may use dependencies between variables implied by the causal order on events, which can be easily derived from Unf_Σ. For example, if we set $x(e) = 1$ then each $x(f)$ such that f is a predecessor (in causal order) of e must be equal to 1, and each $x(g)$ such that g is in conflict with e, must be equal to 0. Similarly, if we set $x(e) = 0$ then no event f for which e is a cause can be executed in the same run, and so $x(f)$ should be equal to 0. Our algorithm will use these observations to reduce the search space, and the experimental results indicate that taking into account causal dependencies, in combination with some heuristics, can lead to significant speedups.

Definition 1. *A vector $x \in \{0,1\}^q$ is* compatible *with Unf_Σ if for all distinct events $e, f \in E$ such that $x(e) = 1$, we have:*

$$f \prec e \Rightarrow x(f) = 1 \quad \text{and} \quad f \# e \Rightarrow x(f) = 0\,.$$

The motivation for considering compatible vectors follows from the next result.

Theorem 3. *A vector $x \in \{0,1\}^q$ is compatible with Unf_Σ if and only if there exists an execution sequence starting at M_{in} whose Parikh vector is x.*

Corollary 1. *For each reachable marking M of Σ, there exists an execution sequence in Unf_Σ leading to a marking representing M, whose Parikh vector x is compatible with Unf_Σ and $x(e) = 0$, for every $e \in E_{cut}$.*

In view of the last result, it is sufficient for a deadlock detection algorithm to check only compatible vectors whose components corresponding to cut-off events are equal to zero. This can be done by building *minimal compatible closure* of a vector x (see the definition below) in each step of CDA and freezing all $x(e)$ such that $e \in E_{cut}$.

Definition 2. *A vector* $\overline{x} \in \{0,1\}^q$ *is a* compatible closure *of* $x \in \{0,1\}^q$ *if* $x \leq \overline{x}$ *and* $\overline{x}$ *is compatible with* Unf_Σ. *Moreover,* $\overline{x}$ *is a* minimal *compatible closure if it is minimal with respect to* $\leq$ *among all possible compatible closures of* x.

Let us consider the causal ordering $e_1 \prec e_2 \prec e_3$, $e_2 \prec e_4$ and e_3 *co* e_4 (see figure 3), and $x = (1,0,1,0)$. Then $\overline{x}' = (1,1,1,0)$ and $\overline{x}'' = (1,1,1,1)$ are compatible closures of x, and $\overline{x}'$ is the minimal one.

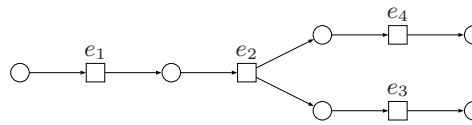


Fig. 3. An occurrence net

Theorem 4. *A vector* $x \in \{0,1\}^q$ *has a compatible closure if and only if for all* $e, f \in E$, $x(e) = x(f) = 1$ *implies* $\neg(e \# f)$. *If* x *has a compatible closure then its minimal compatible closure exists and is unique. Moreover, in such a case if* x *has zero components for all cut-off events, then the same is true for its minimal compatible closure.*

Each step of CDA can be seen as moving from a point $a(x)$ along a default vector $a(\varepsilon_j)$ such that $a(x) \odot a(\varepsilon_j) < 0$, which is interpreted as 'returning to the origin' (see figure 1). However, for an algorithm checking compatible vectors only, each step is moving along a vector which may be represented as a sum of *several* default vectors, and such a geometric interpretation is no longer valid. Indeed, let us consider the same ordering as in figure 3, and the equation

$$a(x) = x_1 + 5x_2 - 3x_3 - 3x_4 = 0$$

(which has a solution $x = (1,1,1,1)$) with an initial constraint $x_1 = 1$. Then we start the algorithm from the vector $x = (1,0,0,0)$, and the sequence of steps should begin from either ε_2 or $\varepsilon_2 + \varepsilon_3$ or $\varepsilon_2 + \varepsilon_4$. But $a(x) \odot a(\varepsilon_2) = 5 \not< 0$, $a(x) \odot a(\varepsilon_2 + \varepsilon_3) = 2 \not< 0$, and $a(x) \odot a(\varepsilon_2 + \varepsilon_4) = 2 \not< 0$, so we cannot choose a vector to make the first step! A possible solution is to interpret each step $\varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ as a sequence of smaller steps $\varepsilon_{i_1}, \ldots, \varepsilon_{i_k}$ where we choose only the first element ε_{i_1} for which $a(\varepsilon_{i_1})$ does return to the origin, and then add the remaining ones in order for $x + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ to be compatible, without worrying

where they actually lead, as it shown in figure 4 (if there is no compatible closure of $x + \varepsilon_{i_1}$ then ε_{i_1} cannot be chosen). This means that we check the condition $a(x) \odot a(\varepsilon_{i_1}) < 0$ which coincides with the original CDA's branching condition, though we are moving along possibly different vector. As the following theorem shows, this technique is still complete.

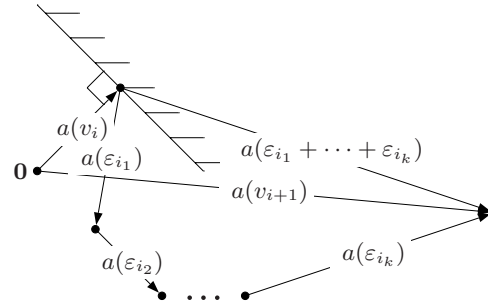


Fig. 4. Geometric interpretation of the new branching condition ($a(\varepsilon_{i_1})$ is 'returning to the origin' although $a(\varepsilon_{i_1} + \cdots + \varepsilon_{i_k})$ may not posses this property; here $v_{i+1} = v_i + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ is the minimal compatible closure of $v_i + \varepsilon_{i_1}$)

Theorem 5. *Every non-trivial minimal compatible solution can be computed using the above method.*

One can prove that the inequalities in the middle of (7) are not essential for an algorithm checking only compatible vectors. Indeed, they are just the result of the substitution of $M = M_{in} + \mathcal{C} \cdot x$ into the constraints $M \geq \mathbf{0}$ and hold for any compatible vector x (see the proof of theorem 3 in [12]). Hence these inequalities can be left out without adding any compatible solution. The reduced system

$$\begin{cases} \displaystyle\sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) \quad \text{for all } e \in E \\ x \in \{0,1\}^q \text{ and } x(e) = 0 \text{ for all } e \in E_{cut} \end{cases} \tag{8}$$

can have minimal solutions which are not compatible with Unf_Σ; however, such solutions are not computed by an algorithm checking only compatible vectors.

Sketch of the Algorithm In the discussion below we refer to the parameters appearing in the generic system (5), since (8) is of that format. The branching condition for (5) can be formulated as

$$x_j \text{ can be incremented by 1 if } \sum_{i=1}^{p} r_i < 0 \,, \tag{9}$$

where each r_i is given by

$$r_i = \begin{cases} 0 & \text{if } a_i \odot x < d_i \text{ and } a_i \odot \varepsilon_j < 0 \\ (a_i \odot x - d_i)(a_i \odot \varepsilon_j) & \text{otherwise} , \end{cases}$$

where a_i=$(a_{i1}, \ldots, a_{iq})$. The algorithm in figure 5 starts from the tuple $(0, \ldots, 0)$ which is the root of the search tree and works in the way similar to the original CDA, but only checks vectors compatible with Unf_{Σ}.

In the general case, the maximal depth of the search tree is q, so CDA needs to store q vectors of length q. In our algorithm, we use just two arrays, X and *FIXED*, of length q:

- $X : array[1..q]\ of\ \{0, 1\}$
  To construct a solution.
- $FIXED : array[1..q]\ of\ integers$
  To keep an information about the levels of fixing the components of X.

The interpretation of these arrays is as follows:

- $FIXED[i] = 0$. Then $X[i]$ must be 0 and this means that $X[i]$ has not been considered yet, and may later be set to 1 or frozen.
- $FIXED[i] = k > 0$ and $X[i] = 0$. Then $X[i]$ has been frozen at some node on level k whose subtree the algorithm is developing. It cannot be unfrozen until the algorithm backtracks to the level k.
- $FIXED[i] = k > 0$ and $X[i] = 1$. Then $X[i]$ has been set to 1 at some node on level k whose subtree the algorithm is developing. This value is fixed for the entire subtree.

Notice that storing the levels of fixing the elements of X allows one to undo changes during backtracking, without keeping all the intermediate values of X. We also use the following auxiliary variables and functions:

- $depth : integer$
  The current depth in the search tree.
- $freeze(i : integer)$
  Freezes all $X[k]$ such that $e_i \preceq e_k$. If there is $X[k] = 1$ to be frozen then *freeze* fails. The corresponding elements of *FIXED* are set to the current value of *depth*.
- $set(i : integer)$
  Sets all $X[j]$ such that $e_j \preceq e_i$ to 1 and uses *freeze* to freeze all $X[k]$ such that $e_i \# e_k$. If there is a frozen $X[j]$ to be set to 1, or $X[k] = 1$ to be frozen then *set* fails. The current value of *depth* is written in the elements of *FIXED*, corresponding to the components being fixed.

Further Optimisation Various heuristics used by general purpose MIP-solvers can be implemented to prune the search tree. For example, if the algorithm has fixed some variables and found out that some of the inequalities have become

Input
$Cons$ — a system of constraints
Unf_Σ — a finite complete prefix of the unfolding

Output
A solution of $Cons$ compatible with Unf_Σ if it exists

Initialisation
$depth \leftarrow 1$
$X \leftarrow (0, \ldots, 0)$
for $i \in \{1, \ldots, q\}$: $FIXED[i] \leftarrow \begin{cases} 1 \text{ if } e_i \in E_{cut} \\ 0 \text{ otherwise} \end{cases}$

Main procedure
if X is a solution of $Cons$ **then return** X
for all $i \in \{k \mid 1 \leq k \leq q \wedge FIXED[k] = 0 \wedge (9) \text{ holds}\}$
  $depth \leftarrow depth + 1$
  if $set(i)$ has succeeded **then**
    apply the procedure to X recursively
    if solution is found **then return** X
  undo changes in the elements $X[j]$ such that $FIXED[j] = depth$
  $depth \leftarrow depth - 1$
  $freeze(i)$ /* never fails here as X is compatible */
return *solution not found*

Fig. 5. Deadlock checking algorithm

infeasible, then it may cut the current branch of the search graph. Moreover, we sometimes can determine the values of variables which have not yet been fixed, or find out that some inequalities have become redundant (see [12] for more details).

After fixing the value of a variable, it is necessary to build a compatible closure for the current vector; new variables can become fixed, so the process can be applied iteratively while it takes effect. If such a closure cannot be built, then the current subtree of the search tree does not contain a compatible solution and may be pruned.

Shortest Trail Finding a shortest path leading to a deadlock may facilitate debugging. In such a case, we need to solve an optimisation problem with the same system of constraints as before, and $\mathcal{L}(x) = x_1 + \cdots + x_q$ as a function to be minimised.

The algorithm can easily be adopted for this task. The only adjustment is not to stop after the first solution has been found, but to keep the current optimal solution together with the corresponding value of the function $\mathcal{L}$ (see [12] for more details).

Table 1. Experimental results

Problem	Deadlock-free	Original net |S|	Original net |T|	Unfolding |B|	Unfolding |E|	Unfolding $\|E_{cut}\|$	Time [s] *McM*	Time [s] *MIP*	Time [s] *PO*
buf100	√	200	101	10101	5051	1	0.01	24577	0.02
mutual	√	49	41	887	479	79	4.42	70	0.02
ab_gesc	√	58	56	3326	1200	511	33.93	260	0.17
sdl_arg	√	163	96	644	199	10	0.04	20	<0.01
sdl_arg_ddlk		157	92	657	223	7	0.01	25	<0.01
RW(2)	√	54	60	498	147	53	0.02	1	<0.01
RW(3)	√	72	120	4668	1281	637	22.14	time	0.06
RW(4)	√	94	224	51040	13513	7841	mem	—	14.55
SEM(2)	√	27	25	61	32	5	<0.01	<0.01	<0.01
SEM(3)	√	38	36	165	86	17	<0.01	1	<0.01
SEM(4)	√	49	47	417	216	49	0.09	5	<0.01
SEM(5)	√	60	58	1013	522	129	2.30	81	0.02
SEM(6)	√	71	69	2393	1228	321	37.33	time	0.16
SEM(7)	√	82	80	5533	2830	769	531.50	—	1.17
SEM(8)	√	93	91	12577	6416	1793	time	—	8.75
SEM(9)	√	104	102	28197	14354	4097	—	—	70.25
SEM(10)	√	115	113	62505	31764	9217	—	—	585.13
ELEVATOR(1)	√	59	72	518	287	9	0.10	27	<0.01
ELEVATOR(2)	√	83	130	29413	15366	1796	mem	time	38.50
STACK(3)		20	24	320	174	26	<0.01	3	<0.01
STACK(4)		24	30	968	525	80	0.08	79	<0.01
STACK(5)		28	36	2912	1578	242	4.28	2408	0.01
STACK(6)		32	42	8744	4737	728	145.01	time	0.05
STACK(7)		36	48	26240	14214	2186	mem	—	0.16
STACK(8)		40	54	78728	42645	6560	—	—	0.52
STACK(9)		44	60	236192	127938	19682	—	—	7.36

buf100 — buffer with 2^{100} states
mutual — mutual exclusion algorithm
ab_gesc — alternating bit protocol
sdl_arg — automatic request protocol
sdl_arg_ddlk — automatic request protocol (with deadlock)
RW(n) — reader-writer with n readers
SEM(n) — semaphore example with n processes
ELEVATOR(n) — example with n elevators
STACK(n) — stack of the depth n with test for fullness

time — the test had not stopped after 15 hours
mem — the test terminated because of memory overflow

Parallelisation Aspects The linear programming approach to deadlock detection described in this paper can easily be implemented on a set of parallel processing nodes. For a shared-memory architecture, we just unfold one step of the recursion and distribute the **for all** loop (see figure 5) between processors[2], freezing some elements according to the frozen components rule. Each processor must have its own copy of the arrays X and *FIXED*.

The algorithm is also appropriate for a distributed memory architecture as the amount of message passing required is relatively low. In this case, each

[2] For a balanced distribution of tasks, it is better to create a queue of unprocessed recursive calls.

node must have its own copies of all arrays, the system of constraints, and the unfolding.

6 Experimental Results

For the experiments, we used the PEP tool [3] to generate finite complete prefixes for our partial order algorithm, and for deadlock checking based on McMillan's method ([14,15]) and the *MIP* algorithm[3] ([15]). The results in table 1 have been measured on a PC with $Pentium^{TM}$ III/500MHz processor and 128M RAM (building unfoldings for RW(5), SEM(11), ELEVATOR(3), and STACK(10) were aborted after 20 hours).

Although our testing was limited in scope, it seems that the new algorithm is fast, even for large unfoldings. In [15], it has been pointed out that the *MIP*-approach is good for 'wide' unfoldings with a high number of cut-off events, whereas for unfoldings with a small percentage of cut-off events, McMillan's approach is better. It appears that our approach works well both for 'wide' unfoldings with a high number of cut-off events and conflicts, and for 'narrow' ones with a high number of causal dependencies. The worst case is the absence of both conflicts and partial order dependencies (i.e. when nearly all events are in the *co* relation) combined with a small percentage of cut-off events. As the general problem is NP-complete in the size of unfolding, such examples can be artificially constructed, but we expect that the new algorithm should work well for practical verification problems.

7 Conclusions

Experiments indicate that the algorithm we propose in this paper can solve problems with thousands of variables. This overcomes the existing limitations, as *MIP*-problems with even a few hundreds of integer variables are usually a hard task for general purpose solvers. It is worth emphasising that the limitation was not the size of computer memory, but rather the time to solve an NP-complete problem. With our approach, the main limitation becomes the size of memory to store the unfolding. Our future research will aim at developing an effective parallel algorithm (especially, for a distributed memory architecture) for constructing large unfoldings which cannot fit in the memory of a single processing node, and modifying our algorithm to handle such 'distributed' unfoldings. Our experiments have indicated that the process of model checking a finite complete prefix is often faster than the process of constructing such a prefix. We therefore plan to investigate novel algorithms for fast generation of net unfoldings.

Another possible direction is to investigate an obvious generalisation of the algorithm to systems of non-linear constraints. In this case we cannot use the

[3] To solve the generated system of constraints, the `lp_solve` general purpose LP-solver by M.R.C.M. Berkelaar was used.

pruning condition described in this paper to reduce the search space, but still need to check only compatible vectors.

Finally, [12] discusses how the approach presented here can be generalised to deal with other relevant verification problems, such as mutual exclusion, coverability and reachability analysis.

Acknowledgements

We would like to thank Paul Watson for his support and comments on an earlier version of this paper, Alexander Letichevsky and Sergei Krivoi for drawing our attention to the Contejean-Devie's algorithm and discussions on the theory of Diophantine equations, and Christian Stehno for his help with using the PEP tool. We would also like to thank the anonymous referees for their helpful comments. This research was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293.

References

1. F. Ajili and E. Contejean: Complete Solving of Linear Diophantine Equations and Inequations Without Adding Variables. Proc. of *1st International Conference on principles and practice of Constraint Programming*, Cassis (1995) 1–17. 411, 414, 416
2. F. Ajili and E. Contejean: Avoiding Slack Variables in the Solving of Linear Diophantine Equations and Inequations. *Theoretical Comp. Sci.* 173 (1997) 183–208. 411, 414, 416
3. E. Best and B. Grahlmann: PEP — more than a Petri Net Tool. Proc. of *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 397–401. 423
4. E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8 (1986) 244–263. 410
5. E. Contejean: Solving Linear Diophantine Constraints Incrementally. Proc. of *10th Int. Conf. on Logic Programming*, D. S. Warren (Ed.). MIT Press (1993) 532–549. 411, 414, 416
6. E. Contejean and H. Devie: Solving Systems of Linear Diophantine Equations. Proc. of *3rd Workshop on Unification*, University of Keiserlautern (1989). 411, 414, 416
7. E. Contejean and H. Devie: An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations. *Inf. and Computation* 113 (1994) 143–172. 411, 414, 416
8. CPLEX Corporation: *CPLEX 3.0*. Manual (1995). 414
9. J. Engelfriet: Branching processes of Petri Nets. *Acta Inf.* 28 (1991) 575–591. 413
10. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106. 411, 413

11. J. Esparza: Model Checking Based on Branching Processes. *Science of Computer Programming* 23 (1994) 151–195. 411, 413
12. V. Khomenko and M. Koutny: Deadlock Checking Using Liner Programming and Partial Order Dependencies. Teachnical Report CS-TR-695, Department of Computing Science, University of Newcastle (2000). 411, 419, 421, 424
13. S. Krivoi: About Some Methods of Solving and Feasibility Criteria of Linear Diophantine Equations over Natural Numbers Domain (in Russian). *Cybernetics and System Analysis* 4 (1999) 12–36.
14. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *CAV'92*, Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174. 411, 413, 423
15. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *CAV'97*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363. 410, 411, 413, 414, 417, 423

Pomsets for Local Trace Languages

— Recognizability, Logic & Petri Nets —

Dietrich Kuske and Rémi Morin*

Institut für Algebra, Technische Universität Dresden, D-01062 Dresden, Germany

Abstract. Mazurkiewicz traces can be seen as equivalence classes of words or as pomsets. Their generalisation by local traces was formalized by Hoogers, Kleijn and Thiagarajan as equivalence classes of step firing sequences. First we introduce a pomset representation for local traces. Extending Büchi's Theorem and a previous generalisation to Mazurkiewicz traces, we show then that a local trace language is recognized by a finite step transition system if and only if its class of pomsets is bounded and definable in the Monadic Second Order logic. Finally, using Zielonka's Theorem, we show that each recognizable local trace language is described by a finite safe labelled Petri net.
The complete version [22] of this paper is accessible on the web.

1 Introduction

Labelled partially ordered sets (pomsets) are widely used to model the behavior of a concurrent system [30,15]; in this approach, the order describes the causal dependence of the events while the labelling denotes which action is performed by an event. In particular, the incomparability of two events denotes that they can be executed simultaneously. Typical examples of this line of research are series-parallel pomsets [23], pomsets without autoconcurrency (also known as semiwords or partial words [33,6]) and dependence graphs of Mazurkiewicz traces [24,7]. A dependence graph is a pomset where the order relation is dictated by a static (or global) independence relation.

To any global independence relation, one can naturally associate an equivalence relation on words that identifies words if they differ only in the order in which independent actions occur. Then, a set of words is an equivalence class iff it is the set of linear extensions of some dependence graph. The rich theory of Mazurkiewicz traces is to a large extend based on these two alternative descriptions: as pomsets and as equivalence classes of words.

In the last decade, several generalizations of Mazurkiewicz traces have been proposed, among them partial traces [6], semi-commutations [4], P-traces [1], computations of concurrent automata [9] and local trace languages [16]. The local trace languages have two characteristic properties that distinguish them from Mazurkiewicz traces: First and similarly to concurrent automata, the independence of actions is not static, but depends on the history of a system. So here in one situation two actions might be independent while in another situation

* Supported by the German Research Foundation (DFG/Graduiertenkolleg)

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 426–441, 2000.

they cannot be performed concurrently. The second distinguishing property is that not only pairs of actions are declared independent or dependent, but finite sets of actions. So it might happen that three actions are mutually independent and that they can be executed in any order, but the set of these three actions cannot be performed concurrently. Nevertheless, one considers the trace equivalence associated to a local trace language where this distinction is not visible anymore.

In this paper, we propose two pomset semantics for a local trace language based on the idea of several sequential observers. Any such sequential observer sees a linear execution of the events. Comparing several sequential observations, one can obtain (partial) knowledge about the concurrency in the execution which then is represented as a pomset. In the first pomset semantics (called processes) we detect only some, but not necessarily all concurrency. Differently in the second pomset semantics (called proper pomsets), all concurrency is represented. Of course, these two semantics are closely related: The processes are the order extensions of the proper pomsets and the proper pomsets are those processes whose order cannot be weakened anymore. One can describe the model of Mazurkiewicz traces, of concurrent automata as well as that of P-traces in the realm of local trace languages [17]. Our pomset semantics via proper pomsets generalize dependence graphs of Mazurkiewicz traces, dependence orders of stably concurrent automata [2] as well as CCI-sets of P-traces [1].

We finish this introduction with a description of the three results on these pomset semantics that we obtain here: Any independence alphabet from the theory of Mazurkiewicz traces defines a class of dependence graphs and there is a bijection between the equivalence classes w.r.t. the commutation of independent actions and these dependence graphs. Hence any set of dependence graphs corresponds to a closed language, namely to the language of linear extensions of one of its dependence graphs. In our context of local trace languages, there is no bijection between the equivalence classes and the pomsets. Therefore, not every set of pomsets is associated to a closed language. Our first result *describes the sets of pomsets that correspond to some local trace language* (Theorem 3.9).

Büchi's paradigmatic result [3] on the relation between finite automata and monadic second order logic has been generalized into different directions, e.g. to finite and infinite trees [35,31], to dependence graphs of Mazurkiewicz traces [34,13], to dependence orders of computations of stably concurrent automata [11], to graphs [5], to series-parallel pomsets [21] etc. Here, we present two further generalizations: First, we show that *a local trace language is recognizable* (i.e. the language of a finite step transition system [26]) *if and only if its set of processes can be defined in the monadic second order logic* (Theorem 4.2). Furthermore, the set of proper pomsets of a recognizable local trace language can be defined in monadic second order logic. Conversely, there are non-recognizable local trace languages whose set of proper pomsets is definable. But we show that a local trace language is recognizable *if and only if its set of proper pomsets is bounded and definable in monadic second order logic* (Theorem 4.6).

Finally, we show how this work relates to the theory of Petri nets: *A local trace language is recognizable iff it is the language of a finite safe Petri net* (Theorem 5.3). This result relies on Zielonka's Theorem [37] and answers a question came up in a discussion with Thiagarajan at CONCUR'98.

2 Basic Notions

Preliminaries. We will use the following notations: for any (possibly infinite) alphabet Σ, and any words $u \in \Sigma^\star$, $v \in \Sigma^\star$, we write $u \leq v$ if u is a prefix of v, i.e. there is $z \in \Sigma^\star$ such that $u.z = v$; the empty word is denoted by ε. We write $|u|_a$ for the number of occurrences of $a \in \Sigma$ in $u \in \Sigma^\star$ and $\wp_f(\Sigma)$ denotes the set of finite subsets of Σ; for any $p \in \wp_f(\Sigma)$, $\mathrm{Lin}(p) = \{u \in p^\star \mid \forall a \in p : |u|_a = 1\}$ is the set of linearisations of p. Finally, if $\lambda : \Sigma \to \Sigma'$ is a map from Σ to Σ', we also write $\lambda : \Sigma^\star \to \Sigma'^\star$ and $\lambda : \wp_f(\Sigma) \to \wp_f(\Sigma')$ to denote the naturally associated monoid morphisms. For any set Γ, $\mathrm{Card}(\Gamma)$ denotes the cardinal of Γ. For any positive integer k, $[1,k]$ denotes the set $\{1, 2, ..., k\}$.

Let $t = (E, \preceq)$ be a finite partial order and $x, y \in E$. Then y *covers* x (denoted $x \prec\!\!\!\cdot\, y$) if $x \prec y$ and $x \prec z \preceq y$ implies $y = z$. For all $z \in E$, we denote by $\Downarrow z$ the set of nodes properly below z, i.e. $\Downarrow z = \{y \in E \mid y \prec z\}$. Furthermore, for any subset M of E, $\Downarrow M$ denotes $\bigcup_{m \in M} \Downarrow m$. The elements x and y are *concurrent* or *incomparable* (denoted $x \mathrm{\ co\ } y$) if $\neg(x \preceq y) \wedge \neg(y \preceq x)$.

2.1 Local Independence Relations and Local Trace Languages

In this section, we introduce the model of a local trace language and define when they are recognizable. As mentioned in the introduction, in a local independence relation, we declare not only pairs of actions to be independent, but finite sets. Furthermore, the independence of such a set might depend on the history of the system, i.e. on the sequence of actions already performed:

Definition 2.1. *A* local independence relation *over Σ is a non-empty subset I of $\Sigma^\star \times \wp_f(\Sigma)$. The* (local) trace equivalence $\sim$ *induced by I is the least equivalence on $\Sigma^\star$ such that*

TE_1*:* $\forall u, u' \in \Sigma^\star \, \forall a \in \Sigma : (u \sim u' \Rightarrow u.a \sim u'.a)$*;*

TE_2*:* $\forall (u,p) \in I \, \forall p' \subseteq p \, \forall v_1, v_2 \in \mathrm{Lin}(p') : u.v_1 \sim u.v_2$.

A (local) trace *is an $\sim$-equivalence class $[u]$ of a word $u \in \Sigma^\star$.*

Note that local trace equivalences are Parikh equivalences since trace equivalent words can be seen as permutations of each other. Usually, one considers only *complete* local independence relations which is e.g. justified in [19,25]:

Definition 2.2. *A local independence relation I over Σ is* complete *if*

Cpl_1*:* $(u,p) \in I \wedge p' \subseteq p \Rightarrow (u,p') \in I$*;*

Cpl_2*:* $(u,p) \in I \wedge p' \subseteq p \wedge v \in \mathrm{Lin}(p') \Rightarrow (u.v, p \setminus p') \in I$*;*

Cpl_3*:* $(u, \{a,b\}) \in I \wedge (u.ab.v, p) \in I \Rightarrow (u.ba.v, p) \in I$*;*

Cpl_4*:* $(u.a, \emptyset) \in I \Rightarrow (u, \{a\}) \in I$.

The first axiom says that with any independent set, any of its subsets is independent. By the second axiom, once a set is independent, one can first execute part of this set and the remaining set stays independent. The third axiom asserts that trace-equivalent words lead to the same independence of steps. The last axiom requires that we list only words in the local independence relation that can be executed sequentially.

Definition 2.3. *A* local trace language *is a structure* $\mathcal{L} = (\Sigma, I, L)$ *where* I *is a complete local independence relation on* Σ *and* $L \subseteq \Sigma^\star$ *is such that*

LTL_1*:* $u \in L \Rightarrow (u, \emptyset) \in I$*;*

LTL_2*:* $u \in L \wedge u \sim v \Rightarrow v \in L$*.*

In view of CpI_4, the first axiom requires that L contains only executable sequences of actions. The second axiom asserts that L is closed w.r.t. the trace equivalence associated with the local independence relation.

Note here that these two requirements form a slight generalization of the local trace languages studied in [19,25,17] since the language L is not necessarily prefix-closed. This extension corresponds to our aim to cope with systems provided with final states. A local trace language (Σ, I, L) such that $L = \{u \in \Sigma^\star \mid (u, \emptyset) \in I\}$ is called *saturated*. Saturated local trace languages are precisely those related with event structures and Petri nets in [19,25].

The following example shows that a step might not be independent, even so any two of its linearisations are trace equivalent. This is fundamentally different from the setting of classical traces and of concurrent automata.

Example 2.4. We consider the alphabet $\Sigma = \{a, b, c\}$ and the local trace language $\mathcal{L} = (\Sigma, I, \Sigma^\star)$ such that

$\forall u \in \Sigma^\star \, \forall p \in \wp_f(\Sigma) : (u, p) \in I \Leftrightarrow [\mathrm{Card}(p) \leq 2 \wedge (u = a \Rightarrow \mathrm{Card}(p) \leq 1)]$.

We observe that $abc \sim bac \sim bca \sim cba \sim cab \sim acb$ although $(a, \{b, c\}) \notin I$. Actually, the trace equivalence is precisely Parikh's equivalence.

In [17], recognizable saturated local trace languages are defined as those that can be represented by a finite step transition system [26]. Providing the latter with final states, we can extend naturally this approach to non-saturated languages. Equivalently (cf. [18]), recognizability for local trace languages is defined as follows.

Definition 2.5. *A local trace language* $\mathcal{L} = (\Sigma, I, L)$ *is* recognizable *if and only if* Σ *is finite,* L *is recognizable in* $\Sigma^\star$*, and for each step* $p \in \wp_f(\Sigma)$*, the language* $L_p = \{u \in \Sigma^\star \mid (u, p) \in I\}$ *is recognizable.*

2.2 Global Independence Relations and Mazurkiewicz Traces

Local trace languages are a direct generalization of classical traces [7]. There, the independence between actions does not depend on the context of previously occurred events. Thus we consider a *global independence relation* over Σ to be a binary symmetric and irreflexive relation $\| \subseteq \Sigma \times \Sigma$. Then a *classical trace*

language over $(\Sigma, \|)$ is a language $L \subseteq \Sigma^\star$ which is closed under the commutation of independent actions: $\forall u, v \in \Sigma^\star\, \forall a, b \in \Sigma : ((u.ab.v \in L \wedge a \| b) \Rightarrow u.ba.v \in L)$. This leads us to introduce classical trace languages formally within the general framework of local trace languages as follows.

Definition 2.6. *Let $\|$ be a global independence relation over Σ. A* classical trace language *over $(\Sigma, \|)$ is a local trace language $\mathcal{L} = (\Sigma, I, L)$ such that for all $u \in \Sigma^\star$, for all $n \in \mathbb{N}$, and for all distinct $a_1, ..., a_n \in \Sigma$, we have $(u, \{a_1, ..., a_n\}) \in I$ if and only if $a_i \| a_j$ for all $1 \leq i < j \leq n$.*

Since $L_p = \Sigma^\star$ for any set p of pairwise independent actions and $L_p = \emptyset$ otherwise, a classical trace language $\mathcal{L}_C = (\Sigma, I, L)$ over a finite alphabet Σ is recognizable (Def. 2.5) iff L is a recognizable language in $\Sigma^\star$. Note here that this representation of Mazurkiewicz traces within local trace languages is simpler than the approach followed in [17]; this is due to the fact that we consider in this paper non-saturated local trace languages.

3 Local Trace Languages Are Classes of Pomsets, Too

Classical traces admit several representations, for they can be identified with pomsets. We extend the relationship between traces and pomsets to the setting of local traces. *In this section, we fix a (possibly infinite) alphabet Σ.*

3.1 Pomsets – Basic Structures & Classical Traces

Definition 3.1. *A* pomset *over Σ is a triple $t = (E, \preceq, \xi)$ where $(E, \preceq)$ is a finite partial order and ξ is a mapping from E to Σ. We denote by $\mathbb{P}(\Sigma)$ the class of all pomsets over Σ. A pomset $t = (E, \preceq, \xi)$ is* without autoconcurrency *if $\xi(x) = \xi(y)$ implies $(x \preceq y$ or $y \preceq x)$ for all $x, y \in E$.*

A pomset can be seen as an abstraction of an execution of a concurrent system. In this view, the elements e of E are *events* and their label $\xi(e)$ describes the basic action of the system that is performed by the event. Furthermore, the order describes the causal dependence between the events. In particular, if two events are concurrent, they can be executed in parallel. A pomset is without autoconcurrency if no action can be performed concurrently with itself.

A *prefix* of a pomset $t = (E, \preceq, \xi)$ is the restriction of t to some downward closed subset of E. An *order extension* of a pomset $t = (E, \preceq, \xi)$ is a pomset $t' = (E, \preceq', \xi)$ such that $\preceq \subseteq \preceq'$. A *linear extension* of t is an order extension that is linearly ordered. Linear extensions of a pomset $t = (E, \preceq, \xi)$ without autoconcurrency can naturally be identified with words over Σ. By $\mathrm{LE}(t) \subseteq \Sigma^\star$, we denote the set of linear extensions of a pomset t over Σ. Since all the classes of pomsets considered (resp. defined) in this paper are assumed (resp. easily checked) to be closed under isomorphisms, we will identify isomorphic pomsets.

Let $\mathcal{L}_C = (\Sigma, I, L)$ be a classical trace language over $(\Sigma, \|)$ and let $u \in \Sigma^\star$. Then the trace $[u]$ is precisely the set of linear extensions $\mathrm{LE}(t)$ of some pomset

$t = (E, \preceq, \xi)$, i.e. $[u] = \mathrm{LE}(t)$. We denote by $\mathfrak{p}_C(\mathcal{L}_C)$ the class of all pomsets t such that $\mathrm{LE}(t) = [u]$ for some $u \in L$. Then any $t = (E, \preceq, \xi) \in \mathfrak{p}_C(\mathcal{L}_C)$ satisfies the following additional properties [24]:

MP_1: for all events $e_1, e_2 \in E$ with $\xi(e_1) \not\parallel \xi(e_2)$, we have $e_1 \preceq e_2$ or $e_2 \preceq e_1$;

MP_2: for all events $e_1, e_2 \in E$ with $e_1 \prec\!\!\!\cdot\, e_2$, we have $\xi(e_1) \not\parallel \xi(e_2)$.

In particular, MP_1 ensures that t is without autoconcurrency. Let $\mathbb{M}(\Sigma, \|)$ denote the class of pomsets over Σ satisfying MP_1 and MP_2. It is well-known [7] that $\mathfrak{p}_C$ is a one-to-one correspondence between the classical trace languages over $(\Sigma, \|)$ and the subclasses of $\mathbb{M}(\Sigma, \|)$.

We want to have a similar presentation by classes of pomsets for arbitrary local trace languages. To this aim, in the following subsection we will define several classes of pomsets that are associated to a given local trace language.

3.2 Pomset Representations of Local Trace Languages

Recall that for classical traces, for any word u, there exists a pomset t whose set of linear extensions equals the equivalence class containing u, i.e. $[u] = \mathrm{LE}(t)$. The following example shows that this is not possible for *local* traces that we consider now (cf. also [2,1] for examples in similar settings).

Example 3.2. We consider the local trace language $\mathcal{L} = (\Sigma, I, \Sigma^\star)$ over the alphabet $\Sigma = \{a, b, c\}$ with

$$I = \{(\varepsilon, \{a, c\}), (c, \{a, b\}), (\varepsilon, \{b, c\})\} \cup \Sigma^\star \times \{\{a\}, \{b\}, \{c\}, \emptyset\}.$$

The trace $[acb] = \{acb, cab, cba, bca\}$ is easily shown *not* to be the set of linear extensions of any pomset. That is why we shall in the sequel associate several pomsets to this trace. Actually $[acb]$ will be associated to the pomsets $t_1 = (\Sigma, \mathrm{Id}_\Sigma \cup (\{c\} \times \{a\}), \mathrm{Id}_\Sigma)$ and $t_2 = (\Sigma, \mathrm{Id}_\Sigma \cup (\{c\} \times \{b\}), \mathrm{Id}_\Sigma)$. Note here that $[acb] = \mathrm{LE}(t_1) \cup \mathrm{LE}(t_2)$ and that the pomset $t_3 = (\Sigma, \mathrm{Id}_\Sigma \cup (\{c\} \times \{a, b\}), \mathrm{Id}_\Sigma)$ is an order extension of t_1 and of t_2.

Pomsets associated to classical traces correspond to usual representations of concurrent executions as elementary event structures [27] or non-branching processes [28]. This interpretation can be generalized to local traces as follows.

Definition 3.3. *Let $\mathcal{L} = (\Sigma, I, L)$ be a local trace language. A* process *of $\mathcal{L}$ is a pomset $t = (E, \preceq, \xi)$ without autoconcurrency such that for all prefixes $t' = (E', \preceq', \xi')$ of t, and for all linear extensions $u \in \mathrm{LE}(t')$, we have*

$$(u, \xi(\min_{\preceq}(E \setminus E'))) \in I.$$

We denote by $\mathfrak{p}^\circ(\mathcal{L})$ the class of all processes of $\mathcal{L}$.

Our restriction to pomsets without autoconcurrency corresponds precisely to the fact that we consider sets of concurrent actions, but not multisets (see also [11,20,19]). In Example 3.2, t_1, t_2 and t_3 are three processes of $\mathcal{L}$.

Example 3.4. We consider again the local trace language $\mathcal{L} = (\Sigma, I, \Sigma^\star)$ of Example 2.4. We observe here that the pomset $t = (\Sigma, \mathrm{Id}_\Sigma, \mathrm{Id}_\Sigma)$ is not a process of $\mathcal{L}$, because $(\varepsilon, \{a, b, c\}) \notin I$, whereas $\mathrm{LE}(t) = \mathrm{Lin}\{a, b, c\} = [abc]$.

This example shows that there might be some pomsets t which are not processes of $\mathcal{L}$ although $\mathrm{LE}(t)$ consists of trace equivalent words. However, conversely, the linear extensions of a process always belong to the same trace. That is, if t is a process of a local trace language $\mathcal{L}$, then $u \sim v$ for any linear extensions $u, v \in \mathrm{LE}(t)$.

Note that the class of processes of any local trace language is closed under isomorphisms, prefixes and order extensions. Yet the class of pomsets $\mathfrak{p}_C(\mathcal{L}_C)$ of some classical trace language $\mathcal{L}_C$ is not closed under order extensions. Thus, in order to get a true extension of $\mathfrak{p}_C$, we have to focus now on *proper pomsets*:

Definition 3.5. *Let $\mathcal{L}$ be a local trace language. A* (proper) pomset *of $\mathcal{L}$ is a process $t \in \mathfrak{p}^\circ(\mathcal{L})$ which is not an order extension of another process of $\mathcal{L}$. We denote by $\mathfrak{p}(\mathcal{L})$ the class of all (proper) pomsets of $\mathcal{L}$.*

For instance, in Example 3.2, t_1 and t_2 are proper pomsets of $\mathcal{L}$ but not t_3, because t_3 is an order extension of t_1. The processes $\mathfrak{p}^\circ(\mathcal{L})$ are precisely the order extensions of elements of $\mathfrak{p}(\mathcal{L})$. Now, since we deal in this paper with possibly non-saturated local trace languages, $\mathfrak{p}(\mathcal{L})$ is not enough to represent $\mathcal{L}$ faithfully. Precisely we still have to specify which processes reach a final state.

Definition 3.6. *Let $\mathcal{L} = (\Sigma, I, L)$ be a local trace language. A process of $\mathcal{L}$ is* final *if each of its linear extensions belongs to L. We denote by $\mathfrak{p}_f(\mathcal{L})$ (resp. $\mathfrak{p}_f^\circ(\mathcal{L})$) the class of all final proper pomsets of $\mathcal{L}$ (resp. final processes of $\mathcal{L}$).*

Since linear extensions of a process are trace equivalent, a proper pomset of $\mathcal{L}$ is final if one of its linear extensions belongs to L. We can check that we obtain in that way a true extension of the usual representation $\mathfrak{p}_C$ of classical trace languages by classes of pomsets: Let $\mathcal{L}_C$ be a classical trace language over $(\Sigma, \|)$. First we observe that $\mathfrak{p}_f(\mathcal{L}_C) = \mathfrak{p}_C(\mathcal{L}_C)$, so the final proper pomsets of a classical trace language correspond to the usual dependence graphs associated to it. Second we can show that $\mathfrak{p}(\mathcal{L}_C) = \mathbb{M}(\Sigma, \|)$: this expresses that the proper pomsets describe all the possible concurrent behaviours enabled by the (global) independence relation, regardless of the accepting executions.

3.3 Expressive Power of Local Trace Languages

The aim of this subsection is to establish that *the map from local trace languages to classes of pomsets is one-to-one* (Th. 3.9). Also we give an order-theoretic characterization of the classes of pomsets which arise in that fashion. In this direction, the first step is to exhibit some properties of the class of processes of any local trace language.

Example 3.7. We consider here again the saturated local trace language $\mathcal{L} = (\Sigma, I, \Sigma^\star)$ of Example 3.2. The pomsets t_1 and $t_1' = (\{a, c\}, \mathrm{Id}_{\{a,c\}}, \mathrm{Id}_{\{a,c\}})$ are proper pomsets of $\mathcal{L}$. Therefore the prefix t_1'' of t_1 restricted to $\{a, c\}$ is not a proper pomset of $\mathcal{L}$ since t_1'' is a strict order extension of t_1'. Thus, in general, $\mathfrak{p}(\mathcal{L})$ *is not closed under prefixes*. We notice now that $t_1^\dagger = (\Sigma, \mathrm{Id}_\Sigma, \mathrm{Id}_\Sigma)$ is not

a process of $\mathcal{L}$ because $[acb] = \{acb, cab, cba, bca\}$. Thus, we have two processes $t_1 = (E, \preceq, \xi)$ and $t'_1 = (E', \preceq', \xi')$ such that $E' \subseteq E$ and $t''_1 = (E', \preceq_{|E' \times E'}, \xi_{|E'})$ is a prefix of t_1 and an order extension of t'_1, but $t_1^{\dagger} = (E, \preceq' \cup \preceq_{|E \times (E \setminus E')}, \xi)$ is not a process of $\mathcal{L}$.

Despite of this example, we introduce now some properties of the class of processes of any local trace language (Def. 3.8 and Th. 3.9). These properties are similar to the one studied in Example 3.7, but of course they are different!

Definition 3.8. *A class $\mathcal{P}$ of pomsets is called* consistent *if the two following conditions are satisfied:*

Cons_1: *If two pomsets $t = (E, \preceq, \xi)$ and $t' = (E', \preceq', \xi')$ in $\mathcal{P}$ are such that*
- *$E' \subseteq E$ and $E \setminus E' \subseteq \max_{\preceq} E$,*
- *$(E', \preceq_{|E' \times E'}, \xi_{|E'})$ is a prefix of t and an order extension of t',*

then the pomset $(E, \preceq' \cup (E' \times (E \setminus E')) \cup \mathrm{Id}_E, \xi)$ belongs also to $\mathcal{P}$.

Cons_2: *If a pomset $t = (E, \preceq, \xi)$ with maximal elements $M \subseteq E$ is such that*
- *$\forall m \in M :\ t_m = (E_m, \preceq_{|E_m \times E_m}, \xi_{|E_m}) \in \mathcal{P}$ where $E_m = E \setminus \{m\}$,*
- *$(E, \preceq \cup((E \setminus M) \times M), \xi) \in \mathcal{P}$,*

then the pomset t belongs also to $\mathcal{P}$.

Let $\mathcal{L}$ be a local trace language and $\mathcal{P} = \mathfrak{p}^{\circ}(\mathcal{L})$. If we take a process of $\mathcal{L}$ and weaken its order relation, this weakening has to be compensated somewhere if we want the weakened pomset to be a process of $\mathcal{L}$. The first axiom above states that, if we weaken the order in a downward closed set of events E', this can be compensated by putting the complement $E \setminus E'$ on top of all of E'. For an explanation of the second axiom, suppose $t' = (E, \preceq \cup(E \setminus M \times M), \xi)$ is a process of $\mathcal{L}$, but $t = (E, \preceq, \xi)$ is not, i.e. we have a process t' where any maximal event is above any non maximal event and weakening this property kicks the pomset out of $\mathcal{P}$. Then this being kicked out is already witnessed by a proper prefix $E \setminus \{m\}$ of t' for some $m \in M$.

This leads us to the main result of this section: each local trace language $\mathcal{L}$ is faithfully represented by the classes of pomsets $\mathfrak{p}(\mathcal{L})$ and $\mathfrak{p}_f(\mathcal{L})$. In other words *the mapping $\mathcal{L} \mapsto (\mathfrak{p}(\mathcal{L}), \mathfrak{p}_f(\mathcal{L}))$ is one-to-one.* Moreover, the pairs of classes of pomsets associated to local trace languages in that way are characterized as follows.

Theorem 3.9. *Let $\mathcal{P}$ and $\mathcal{P}'$ be two classes of pomsets without autoconcurrency over Σ such that $\mathcal{P} \neq \emptyset$. There exists a local trace language $\mathcal{L}$ such that $\mathcal{P} = \mathfrak{p}(\mathcal{L})$ and $\mathcal{P}' = \mathfrak{p}_f(\mathcal{L})$ if and only if $\mathcal{P}$ and $\mathcal{P}'$ satisfy the following requirements:*

1. *for all pomsets $t \in \mathcal{P}$ and $t' \in \mathcal{P}$, if t' is an order extension of t then $t' = t$;*
2. *the class of order extensions of $\mathcal{P}$ is consistent and closed under prefixes;*
3. *for all pomsets $t \in \mathcal{P}$ and $t' \in \mathcal{P}'$: If $\mathrm{LE}(t) \cap \mathrm{LE}(t') \neq \emptyset$, then $t \in \mathcal{P}'$.*

In that case there is a unique *local trace language $\mathcal{L}$ such that $\mathcal{P} = \mathfrak{p}(\mathcal{L})$ and $\mathcal{P}' = \mathfrak{p}_f(\mathcal{L})$.*

4 Recognizability, Logical Definability and Boundedness

We aim now at extending to the framework of local trace languages the relationship between recognizability and logical definability known for the free monoid as Büchi's Theorem and already generalized to classical traces [7,13]: *any classical trace language is recognizable if and only if its class of final proper pomsets is definable within the Monadic Second Order logic.*

In this section, we fix a *finite* alphabet Σ. Formulas of the MSO language over Σ that we consider involve first order variables $x, y, z...$ for events and set variables $X, Y, Z...$ for sets of events. They are built up from the atomic formulas $P_a(x)$ for $a \in \Sigma$ (which stands for "the event x is labelled by the action a"), $x \preceq y$, and $x \in X$ by means of the boolean connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ and quantifiers $\exists, \forall$ (both for first order and for set variables). Formulas without free variables are called sentences.

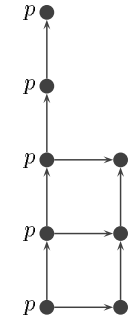
The satisfaction relation $\models$ between pomsets $t = (E, \preceq, \xi)$ and a sentence φ of the monadic second order logic is defined canonically with the understanding that first order variables range over the events of E and second order variables over subsets of E. The class of pomsets *without autoconcurrency* which satisfy a sentence φ is denoted by $\mathrm{Mod}(\varphi)$. We say that a class of pomsets without autoconcurrency $\mathcal{P}$ is *MSO-definable* if there exists a monadic second order sentence φ such that $\mathcal{P} = \mathrm{Mod}(\varphi)$.

Example 4.1. We consider here a Producer-Consumer system. Its alphabet is $\Sigma = \{p, c\}$ where p represents a production of one item and c a consumption. The language of this system describes all the possible sequences for which at each stage there are at least as many productions as consumptions. Formally, $L = \{u \in \Sigma^\star \mid \forall v \leq u : |v|_p \geq |v|_c\}$. We now want to model a possible independency between the producer and the consumer. Provided that there have already been enough items produced, the producer and the consumer can act simultaneously. This can be represented by the local independence relation I defined as follows:

- $(u, \emptyset) \in I \Leftrightarrow u \in L \Leftrightarrow (u, \{p\}) \in I$;
- $(u, \{c\}) \in I \Leftrightarrow (u \in L \wedge |u|_p > |u|_c) \Leftrightarrow (u, \{p, c\}) \in I$.

Note here that $\mathcal{L} = (\Sigma, I, L)$ is not a classical trace language since $ppc \sim pcp$, but $pc \not\sim cp$. The proper pomsets of this local trace language are precisely those of the form depicted on the right. These are MSO-definable by

$$\begin{aligned}
&\forall x, y : P_p(y) \wedge x \preceq y \rightarrow P_p(x) \quad \wedge \\
&\forall y : (P_c(y) \rightarrow \exists x (P_p(x) \wedge x \prec\!\!\!\cdot\, y)) \quad \wedge \\
&\forall x, z : ((P_p(x) \wedge P_c(z) \wedge x \preceq z) \rightarrow \exists y (P_c(y) \wedge x \prec\!\!\!\cdot\, y))
\end{aligned}$$



However this saturated local trace language is not recognizable since L is not recognizable.

This example shows that recognizability is *not* equivalent to MSO-definability in the general framework of local trace languages. Yet, Theorems 4.2 and 4.6 give

two characterizations of recognizable local trace languages in the spirit of Büchi's Theorem.

4.1 Recognizability = Definability of Processes

By [13,11], there exists a formula φ with two free variables x and y such that for any pomset without autoconcurrency $t = (E, \preceq, \xi)$, the set $\varphi^t = \{(x, y) \in E^2 \mid t \models \varphi(x, y)\}$ is a linear extension of $\preceq$. The proof from [8] (cf. proof of Theorem 5.1), is easily seen to work for general pomsets without autoconcurrency, even though the result is stated for Mazurkiewicz traces, only.

Now, we can state our first Büchi-type characterization of recognizable local trace languages in terms of its processes:

THEOREM 4.2. *A local trace language $\mathcal{L}$ is recognizable if and only if its class of processes $\mathfrak{p}^\circ(\mathcal{L})$ and its class of final processes $\mathfrak{p}^\circ_f(\mathcal{L})$ are MSO-definable.*

Proof. (sketch) Let $\mathcal{L}$ be recognizable. By Büchi's Theorem, we find for any $p \subseteq \Sigma$ a formula ψ_p which defines $\{u \in \Sigma^\star \mid (u, p) \in I\}$. Hence, a pomset $t = (E, \preceq, \xi)$ without autoconcurrency is a process of $\mathcal{L}$ iff the following holds for any prefix $t' = (E', \preceq_{|E'}, \xi_{|E'})$:

if the minimal elements of the complementary suffix of t are labeled by $p \subseteq \Sigma$, then the word $(E', \varphi^{t'}, \xi_{|E'})$ satisfies ψ_p

which can be expressed in monadic second order logic. □

4.2 Boundedness Property & Classical Traces

Example 4.1 shows that there exist local trace languages whose class of proper pomsets is MSO-definable, but that are not recognizable. Thus, in general, the duality between recognizability and MSO-definability known for classical traces does not generalize directly to local trace languages. Therefore, now we introduce the notion of boundedness that will enable us to prove the desired generalization (cf. Th. 4.6 below).

DEFINITION 4.3. *[20] Let $t = (E, \preceq, \xi)$ be a pomset and k be a positive integer. A k-*chain covering *of t is a family $(C_i)_{i\in[1,k]}$ of subsets of E such that*

1. *each C_i is a chain in $(E, \preceq)$, i.e. $(C_i, \preceq \cap (C_i \times C_i))$ is a linear order;*
2. *$E = \bigcup_{i\in[1,k]} C_i$;*
3. *for all $x, y \in E$: If $x \prec\!\!\!\cdot\, y$, then there exists $1 \leq i \leq k$ such that $\{x, y\} \subseteq C_i$.*

For any $k \in \mathbb{N}$, $\mathbb{P}_k(\Sigma)$ denotes the class of pomsets over Σ which admit a k-chain covering. A class of pomsets over Σ is *bounded* if it is included in some $\mathbb{P}_k(\Sigma)$. Note here that the class of proper pomsets of the Producer-Consumer system of Example 4.1 is not bounded.

As far as classical traces are concerned, the class $\mathbb{M}(\Gamma, \|)$ of classical traces over $(\Gamma, \|)$ is bounded by $\mathrm{Card}(\Gamma)^2$ whenever Γ is finite. To see this, we simply consider for each pair of dependent actions $a \not\| b$ the set of events labelled by a

or b. Then, by Axiom MP_1, this set forms a chain and by MP_2 the family of these chains forms a chain covering.

Conversely, any bounded class is the projection of some class of classical traces [20]: Consider the alphabet $\Gamma_k = \Sigma \times (\wp_f([1,k]) \setminus \{\emptyset\})$ provided with the global independence relation $\|$ such that $(a,M)\|(b,N)$ iff $M \cap N = \emptyset$. Let $\pi_1 : \Gamma_k \to \Sigma$ be the projection to the first component and define $\pi_1(E, \preceq, \xi) = (E, \preceq, \pi_1 \circ \xi)$ for any $(E, \preceq, \xi) \in \mathbb{M}(\Gamma_k, \|)$. Then $\pi_1(\mathbb{M}(\Gamma_k, \|)) = \mathbb{P}_k(\Sigma)$. Even more: Let $\mathcal{P} \subseteq \mathbb{P}(\Sigma)$ be a bounded and MSO-definable class of pomsets without autoconcurrency. Then the preimage $\pi_1^{-1}(\mathcal{P}) \subseteq \mathbb{M}(\Gamma_k, \|)$ is MSO-definable, too. In other words, any MSO-definable and bounded class of pomsets without autoconcurrency is the projection of some MSO-definable class of classical traces:

Lemma 4.4. *Let $\mathcal{P}$ be a class of pomsets without autoconcurrency over the finite alphabet Σ. If $\mathcal{P}$ is bounded by k and MSO-definable, then there is an MSO-definable subclass $\mathcal{P}_C$ of $\mathbb{M}(\Gamma_k, \|)$ such that $\pi_1 : \Gamma_k \to \Sigma$ induces a surjection from $\mathcal{P}_C$ onto $\mathcal{P}$.*

In view of Theorem 4.2, the following corollary implies that a local trace language with a bounded class of proper pomsets is recognizable iff its classes of proper and of final proper pomsets are MSO-definable:

Corollary 4.5. *Let $\mathcal{L}$ be a local trace language over the finite alphabet Σ such that $\mathfrak{p}(\mathcal{L})$ is bounded by k. Then $\mathfrak{p}(\mathcal{L})$ ($\mathfrak{p}_f(\mathcal{L})$, respectively) is MSO-definable if and only if $\mathfrak{p}^\circ(\mathcal{L})$ ($\mathfrak{p}_f^\circ(\mathcal{L})$, respectively) is MSO-definable.*

Proof. Suppose $\mathfrak{p}^\circ(\mathcal{L})$ is MSO-definable by φ. Note that a process $t = (E, \preceq, \xi)$ is a proper pomset iff, for any $e, f \in E$ with $e \mathrel{—\!\!\prec} f$ and $\xi(f) \neq \xi(f)$, the pomset $(E, \preceq \setminus \{(e,f)\}, \xi)$ is not a process of $\mathcal{L}$. Replacing in φ any subformula of the form $x \preceq y$ by $x \preceq y \wedge \neg(x = e \wedge y = f)$, we obtain a formula φ'. Now

$$\mathfrak{p}(\mathcal{L}) = \mathrm{Mod}(\varphi \wedge \forall e, f : ((e \mathrel{—\!\!\prec} f \wedge \xi(e) \neq \xi(f)) \to \neg\varphi')),$$

i.e. $\mathfrak{p}(\mathcal{L})$ is MSO-definable.

Conversely let $\mathfrak{p}(\mathcal{L})$ be defined by ψ. By Lemma 4.4 there exists an MSO-definable class of classical traces $\mathcal{P}_C \subseteq \mathbb{M}(\Gamma_k, \|)$ with $\pi_1(\mathcal{P}_C) = \mathfrak{p}(\mathcal{L})$. Since the order of any pomset $t \in \mathcal{P}_C$ is dictated by the global independence relation $\|$, the class $\mathcal{P}_C^\dagger$ of order extensions of elements of $\mathcal{P}_C$ is MSO-definable, too. In addition, this class $\mathcal{P}_C^\dagger$ is mapped by π_1 onto the class of order extensions of elements of $\pi_1(\mathcal{P}_C) = \mathfrak{p}(\mathcal{L})$, i.e. onto $\mathfrak{p}^\circ(\mathcal{L})$. Since the projection of an MSO-definable class is MSO-definable, the class $\mathfrak{p}^\circ(\mathcal{L})$ of processes of $\mathcal{L}$ is MSO-definable.

The proof for the final pomsets and processes goes through verbatim. □

4.3 Recognizability = Definability and Boundedness of Pomsets

So far, we showed that a local trace language is recognizable whenever $\mathfrak{p}(\mathcal{L})$ and $\mathfrak{p}_f(\mathcal{L})$ are MSO-definable and bounded. The aim of this section is to show the inverse implication. In view of Cor. 4.5, it remains to show that $\mathfrak{p}(\mathcal{L})$ is bounded whenever $\mathcal{L}$ is recognizable. So, let $\mathcal{L}$ be recognizable. Then there exists a finite

monoid $(S, \cdot)$ and a homomorphism $\eta : \Sigma^\star \to S$ such that, for any $u, v \in \Sigma^\star$ with $\eta(u) = \eta(v)$ and any $p \in \wp_f(\Sigma)$ we have: $(u, p) \in I$ iff $(v, p) \in I$.

By contradiction assume that $\mathfrak{p}(\mathcal{L})$ is unbounded. Then, for any $n \in \mathbb{N}$, there exists $t = (E, \preceq, \xi) \in \mathfrak{p}(\mathcal{L})$ and $x_i, y_i \in E$ for $1 \leq i \leq n$ such that

1. $\xi(x_i) = \xi(x_j)$ and $\xi(y_i) = \xi(y_j)$ for $1 \leq i \leq j \leq n$,
2. $x_i \prec\!\!\!\!\cdot\, y_i$ for $1 \leq i \leq n$, and
3. $x_i \prec x_j$, $y_i \prec y_j$, and x_j co y_i for $1 \leq i < j \leq n$.

Recall that a process t of $\mathcal{L}$ is a proper pomset if its order cannot be weakened without leaving the class $\mathfrak{p}^\circ(\mathcal{L})$. Hence, we cannot remove any covering relation from t, or, the other way round, for any edge in the covering relation of t, there has to be a "reason". In the context of local trace languages, this amounts to say that we find antichains $M_i \subseteq E$ for $1 \leq i \leq n$ such that $y_i \in M_i$ and, for any linear extension u_i of the restriction of t to $F_i = {\Downarrow}M_i \setminus \{x\}$, we have $(u_i, \xi(M_i \cup \{x\})) \notin I$. Since Σ is finite, we might assume $\xi(M_i \setminus {\Downarrow}x_n) = \xi(M_j \setminus {\Downarrow}x_n)$ for $1 \leq i < j < n$.

Next, one shows that $\{x_j\} \cup (M_j \cap {\Downarrow}x_n) \cup (M_i \setminus {\Downarrow}x_n)$ consists of minimal elements of $E \setminus ((F_j \cap {\Downarrow}x_n) \cup (F_i \setminus {\Downarrow}x_n))$ for $1 \leq i < j < n$. Hence we obtain

$$\begin{aligned}\xi(\min(E \setminus ((F_j \cap {\Downarrow}x_n) \cup (F_i \setminus {\Downarrow}x_n)))) &\supseteq \xi(x_j) \cup \xi(M_j \cap {\Downarrow}x_n) \cup \xi(M_i \setminus {\Downarrow}x_n) \\ &= \xi(x_j) \cup \xi(M_j \cap {\Downarrow}x_n) \cup \xi(M_j \setminus {\Downarrow}x_n) = \xi(M_j \cup \{x_j\}).\end{aligned}$$

So far, so good. Due to lack of space, we cannot give the details that prove the following step; the reader is invited to look at the complete and technical proof in [22]. This step uses Ramsey's Theorem [32] and a close analysis of the relation of the sets M_i in the proper pomset t. It results in the existence of $1 \leq i < j < n$ with the following nice property: Let v be some linear extension of the restriction of t to the set $(F_j \cap {\Downarrow}x_n) \cup (F_i \setminus {\Downarrow}x_n)$. Recall that u_j is a linear extension of the set $F_j = (F_j \cap {\Downarrow}x_n) \cup (F_j \setminus {\Downarrow}x_n)$. Then, and this is indeed the crucial point of the proof that comes out of Ramsey's Theorem, $\eta(v) = \eta(u_j)$.

Now we can show that this leads to a contradiction: By the choice of M_j, we have $(u_j, \xi(M_j \cup \{x_j\})) \notin I$. Hence, since η recognizes $\mathcal{L}$, we obtain $(v, \xi(M_j \cup \{x_j\})) \notin I$ and therefore (by Cpl_1)

$$(v, \xi(\min(E \setminus ((F_j \cap {\Downarrow}x_n) \cup (F_i \setminus {\Downarrow}x_n))))) \notin I.$$

But this, indeed, contradicts that $t = (E, \preceq, \xi)$ is a process of $\mathcal{L}$.

Hence, we indicated how to show that the class of proper pomsets $\mathfrak{p}(\mathcal{L})$ is bounded for any recognizable local trace language $\mathcal{L}$. By Theorem 4.2 and Corollary 4.5, we now obtain our second Büchi-type result:

THEOREM 4.6. *A local trace language is recognizable if and only if its class of proper pomsets and its class of final proper pomsets are MSO-definable and bounded.*

Proof. If $\mathfrak{p}(\mathcal{L})$ is bounded and $\mathfrak{p}(\mathcal{L})$ and $\mathfrak{p}_f(\mathcal{L})$ are MSO-definable, then by Corollary 4.5, the classes $\mathfrak{p}^\circ(\mathcal{L})$ and $\mathfrak{p}^\circ_f(\mathcal{L})$ are MSO-definable, too. Hence, Theorem 4.2 asserts that $\mathcal{L}$ is recognizable. If, conversely, $\mathcal{L}$ is recognizable, by Theorem 4.2, the classes $\mathfrak{p}^\circ(\mathcal{L})$ and $\mathfrak{p}^\circ_f(\mathcal{L})$ are MSO-definable. Furthermore, by what we saw above, $\mathfrak{p}(\mathcal{L})$ is bounded. Now Corollary 4.5 yields that the classes $\mathfrak{p}(\mathcal{L})$ and $\mathfrak{p}_f(\mathcal{L})$ are MSO-definable. □

5 Finite Distributed Implementation by Petri Nets

In [26], Mukund extended the so-called *synthesis problem* of elementary nets [14] to the more general model of Petri nets. A similar study was achieved by Hoogers, Kleijn and Thiagarajan in [16]: Using some generalized *regions*, they characterized which local trace languages correspond to the behaviour of unlabelled (possibly non-safe) Petri nets. We show here that any *recognizable* saturated local trace language is the language of a *finite safe labelled* Petri net.

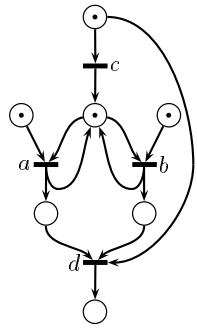


FIG. 1. Petri net $\mathcal{N}$ of Ex. 5.4

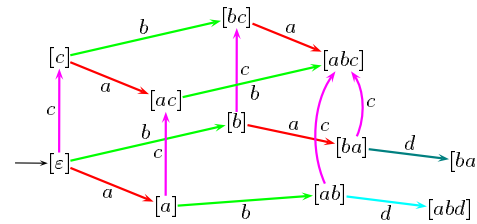


FIG. 2. Traces of Petri net $\mathcal{N}$ (Ex. 5.4)

DEFINITION 5.1. *A* Petri net *is a quadruple* $\mathcal{N} = (S, T, W, \mathrm{M}_{in})$ *where*
- S *is a set of places and* T *is a set of transitions such that* $S \cap T = \emptyset$*;*
- W *is a map from* $(S \times T) \cup (T \times S)$ *to* $\mathbb{N}$*, called* weight function*;*
- M_{in} *is a map from* S *to* $\mathbb{N}$*, called* initial marking*.*

Given a Petri net $\mathcal{N} = (S, T, W, \mathrm{M}_{in})$, $\mathrm{Mar}_{\mathcal{N}}$ denotes the set of all markings of $\mathcal{N}$ that is to say functions $\mathrm{M} : S \to \mathbb{N}$; a step $p \in \wp_f(T)$ is *enabled* at $\mathrm{M} \in \mathrm{Mar}_{\mathcal{N}}$ if $\mathrm{M}(s) \geq \sum_{t \in p} W(s,t)$ for all $s \in S$; in this case, we write $\mathrm{M}\,[p\rangle\,\mathrm{M}'$ where $\mathrm{M}'(s) = \mathrm{M}(s) + \sum_{t \in p}(W(t,s) - W(s,t))$ and say that the transitions of p may be *fired* concurrently and lead to the marking M'. A *step firing sequence* consists of a sequence of markings M_0,..., M_n and a sequence of steps p_1,..., $p_n \in \wp_f(T)$ such that $\mathrm{M}_0 = \mathrm{M}_{in}$ and $\mathrm{M}_{k-1}\,[p_k\rangle\,\mathrm{M}_k$ for all $1 \leq k \leq n$.

DEFINITION 5.2. *A* labelled Petri net *is a structure* $(S, T, W, \mathrm{M}_{in}, \xi)$ *where* $(S, T, W, \mathrm{M}_{in})$ *is a Petri net and* ξ *is a map from* T *to an alphabet* Σ*.*

The local independence relation *associated to a labelled Petri net* $\mathcal{N} = (S, T, W, \mathrm{M}_{in}, \xi)$ *is* $I = \{(\xi(t_1 ... t_n), \xi(p)) \mid (t_1 ... t_n, p) \in T^\star \times \wp_f(T) \wedge\ \mathrm{M}_{in}\,[\{t_1\}\rangle\,\mathrm{M}_1 ...\ [\{t_n\}\rangle\,\mathrm{M}_n\,[p\rangle\,\mathrm{M}_{n+1}$ *for some* $M_i \in Mar_{\mathcal{N}}\}$ *and the set of sequential executions is* $L = \{u \in \Sigma^\star \mid (u, \emptyset) \in I\}$*.*

A labelled Petri net $\mathcal{N}$ *is* well-labelled *if its associated local independence relation is complete. In that case* (Σ, I, L) *is called the local trace language of* $\mathcal{N}$*.*

Since all markings are "accepting", the local trace language of a well-labelled Petri net is saturated.

The labelling ξ is called *deterministic* if for all step firing sequences $\mathrm{M}_{in} = \mathrm{M}_0 [p_1\rangle \ldots \mathrm{M}_{n-1} [p_n\rangle \mathrm{M}_n$ and all transitions $t, t' \in T$:

$$\mathrm{M}_n [\{t\}\rangle \mathrm{M}_{n+1} \wedge \mathrm{M}_n [\{t'\}\rangle \mathrm{M}'_{n+1} \wedge \xi(t) = \xi(t') \Rightarrow t = t'.$$

This restriction ensures that two transitions enabled by a common reachable marking correspond to two distinct actions. This implies in particular that the net is well-labelled and the associated marking graph is a step transition system.

A Petri net is called *finite* if it has only a finite number of places, transitions and actions. It is called *safe* when there are only a finite number of reachable markings. One can easily check that the local trace language of any finite safe well-labelled Petri net is recognizable. Less obvious is the converse. The latter can be established using Theorem 4.6, Lemma 4.4 and [17, Th. 5.4] (which relies on Zielonka's Theorem [37]).

THEOREM 5.3. *A saturated local trace language is recognizable if and only if it is the language of a finite, safe, well-labelled Petri net.*

As shown by Example 5.4, Theorem 5.3 fails if one considers only Petri nets with deterministic labelling.

EXAMPLE 5.4. Let $\mathcal{L} = (\Sigma, I, L)$ be the saturated local trace language associated to the Petri net of Figure 1. Its traces are informally described on Figure 2. We consider now the saturated local trace language $\mathcal{L}'$ over Σ associated to the local independence relation $I' = I \setminus \{(ab, \{d\}), (abd, \emptyset)\}$. We prove here by contradiction that $\mathcal{L}'$ is not the language of a Petri net with a deterministic labelling. For such a Petri net, all the occurrences of b correspond to the firing of the same transition because $\{(a, \{b, c\}), (\varepsilon, \{a, c\}), (c, \{a, b\}), (\varepsilon, \{b, c\})\} \subseteq I'$. Similarly for a. Hence ab and ba lead to the same marking. Now one d-labelled transition is enabled after ba, so the same transition is enabled after ab: $(ab, \{d\}) \in I'$.

References

1. A. Arnold: *An extension of the notion of traces and asynchronous automata.* Theoretical Informatics and Applications **25** (1991) 355–393 426, 427, 431
2. F. Bracho, M. Droste, and D. Kuske: *Representations of computations in concurrent automata by dependence orders.* Theoretical Comp. Science **174** (1997) 67–96 427, 431
3. J. R. Büchi: *Weak second-order arithmetic and finite automata.* Z. Math. Logik Grundlagen Math. **6** (1960) 66–92 427
4. M. Clerbout, M. Latteux, and Y. Roos: *Semi-Commutations.* In [7], chap. **12** (1995) 487–552 426
5. B. Courcelle: *The monadic second-order logic of graphs. I: Recognizable sets of finite graphs.* Information and Computation **85** (1990) 12–75 427
6. V. Diekert: *A partial trace semantics for Petri nets.* Theoretical Comp. Science **134** (1994) 87–105 426
7. V. Diekert and G. Rozenberg: *The Book of Traces.* (World Scientific, 1995) 426, 429, 431, 434, 439

8. V. Diekert and Y. Métivier: *Partial Commutations and Traces*. Handbook of Formal languages, vol. **3** (1997) 457–533 435
9. M. Droste: *Concurrent automata and domains*. International Journal of Foundations of Computer Science **3** (1992) 389–418 426
10. M. Droste and P. Gastin: *Asynchronous cellular automata for pomsets without autoconcurrency*. CONCUR'96, LNCS **1119** (1996) 627–638
11. M. Droste and D. Kuske: *Logical definability of recognizable and aperiodic languages in concurrency monoids*. LNCS **1092** (1996) 233–251 427, 431, 435
12. M. Droste, P. Gastin, and D. Kuske: *Asynchronous cellular automata for pomsets*. Theoretical Comp. Science (2000) – To appear.
13. W. Ebinger and A. Muscholl: *Logical definability on infinite traces*. Theoretical Comp. Science **154** (1996) 67–84 427, 434, 435
14. A. Ehrenfeucht and G. Rozenberg: *Partial (Set) 2-structures*. Part II: State spaces of concurrent systems, Acta Informatica **27** (1990) 343–368 438
15. J. L. Gischer: *The equational theory of pomsets*. Theoretical Comp. Science **61** (1988) 199–224 426
16. P. W. Hoogers, H. C. M. Kleijn, and P. S. Thiagarajan: *A Trace Semantics for Petri Nets*. Information and Computation **117** (1995) 98–114 426, 438
17. J.-Fr. Husson and R. Morin: *On Recognizable Stable Trace Languages*. FOSSACS 2000, LNCS **1784** (2000) 177–191
18. J.-Fr. Husson and R. Morin: *Relationships between Arnold's CCI sets of P-traces and Droste's stably concurrent automata*. Technical report MATH-AL-1-00 (TU-Dresden, 2000) 427, 429, 430, 439 429
19. H. C. M. Kleijn, R. Morin, and B. Rozoy: *A General Categorical Connection between Local Event Structures and Local Traces*. FCT'99, LNCS **1684** (1999) 338–349 428, 429, 431
20. D. Kuske: *Asynchronous cellular and asynchronous automata for pomsets*. CONCUR'98, LNCS **1466** (1998) 517–532 431, 435, 436
21. D. Kuske: *Infinite series-parallel posets: logic and languages*. ICALP (2000) – to appear 427
22. D. Kuske and R. Morin: *Pomsets for local trace languages*. Technical report, TU Dresden (2000) – available at `http://www.math.tu-dresden.de/~morin/papers/km00a.ps` 426, 437
23. K. Lodaya and P. Weil: *Series-parallel languages and the bounded-width property*. Theoretical Comp. Science **237** (2000) 347–380 426
24. A. Mazurkiewicz: *Concurrent program schemes and their interpretations*. Aarhus University Publication (DAIMI PB-78, 1977) 426, 431
25. R. Morin and B. Rozoy: *On the Semantics of Place/Transition Nets*. CONCUR'99, LNCS **1664** (1999) 447–462 428, 429
26. M. Mukund: *Petri Nets and Step Transition Systems*. International Journal of Foundations of Computer Science **3** (1992) 443–478 427, 429, 438
27. M. Nielsen, G. Plotkin, and G. Winskel: *Petri nets, events structures and domains, part 1*. Relationships between Models of Concurrency, Theoretical Comp. Science **13** (1981) 85–108 431
28. M. Nielsen, G. Rozenberg, and P. S. Thiagarajan: *Behavioural Notions for Elementary Net Systems*. Distributed Computing **4** (1990) 45–57 431
29. M. Nielsen, V. Sassone, and G. Winskel: *Relationships between Models of Concurrency*. LNCS **803** (1994) 425–475
30. V. Pratt: *Modelling concurrency with partial orders*. Int. J. of Parallel Programming **15** (1986) 33–71 426

31. M. O. Rabin: *Decidability of second-order theories and automata on infinite trees.* Trans. Amer. Math. Soc. **141** (1969) 1–35 427
32. F. P. Ramsey: *On a problem of formal logic.* Proc. London Math. Soc. **30** (1930) 264–286 437
33. P. H. Starke: *Processes in Petri nets.* Elektronische Informationsverarbeitung und Kybernetik **17** (1981) 389–416 426
34. W. Thomas: *On logical definability of trace languages.* Technical University of Munich, report TUM-I9002 (1990) 172–182 427
35. J. W. Thatcher and J. B. Wright: *Generalized finite automata with an application to a decision problem of second-order logic.* Math. Systems Theory **2** (1968) 57–82 427
36. G. Winskel: *Event structures.* Petri nets: Applications and Relationships to Other Models of Concurrency, LNCS **255** (1987) 325–392
37. W. Zielonka: *Notes on finite asynchronous automata.* Theoretical Informatics and Applications **21** (1987) 99–135 428, 439

Functional Concurrent Semantics for Petri Nets with Read and Inhibitor Arcs*

Paolo Baldan[1], Nadia Busi[2], Andrea Corradini[1], and G. Michele Pinna[3]

[1] Dipartimento di Informatica - Università di Pisa
[2] Dipartimento di Scienze dell'Informazione - Università di Bologna
[3] Dipartimento di Matematica - Università di Siena

Abstract. We propose a functorial concurrent semantics for Petri nets extended with *read* and *inhibitor* arcs, that we call *inhibitor nets*. Along the lines of the seminal work of Winskel on safe nets, the truly concurrent semantics is given at a categorical level via a chain of functors leading from the category **SW-IN** of semi-weighted inhibitor nets to the category **Dom** of finitary prime algebraic domains. As an intermediate semantic model, we introduce *inhibitor event structures*, an extension of prime event structures able to faithfully capture the dependencies among events which arise in the presence of read and inhibitor arcs.

1 Introduction

Several generalizations of Petri nets [15] have been proposed in the literature to overcome the expressiveness limitations of the classical model. At a very basic level Petri nets have been extended with two new kinds of arcs, namely *read arcs* (also called test, read, activator or positive contextual arcs) [7,11,8,16] and *inhibitor arcs* (also called negative contextual arcs) [1,11], which allow a transition to check for the presence, resp. absence of tokens, without consuming them. Read arcs have been shown to be useful to model in a natural way several practical situations (see e.g. [3,2] for more references). A study of the expressiveness of inhibitor arcs, along with a comparison with other extensions proposed in the literature, namely priorities, exclusive-or transitions and switches, is carried out in [13]. In particular, we recall that inhibitor arcs make the model Turing complete, essentially because they allow to simulate the zero-testing operation of RAM machines, not expressible only with flow and read arcs.

The purpose of this paper is to give a truly concurrent semantics to Petri nets extended with read and inhibitor arcs, that we will call *inhibitor nets*. We follow the seminal work on ordinary safe nets of [12,18], where the semantics is given at a categorical level via a chain of coreflections, leading from the category **S-N** of safe (marked) P/T nets to the category **Dom** of finitary prime algebraic domains, through the categories **O-N** of occurrence nets and **PES** of prime event structures (PES's), the last step being an equivalence of categories.

* Research partly supported by the TMR Network GETGRATS, by the Esprit Working Groups *APPLIGRAPH* and by the MURST project *TOSCA*.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 442–457, 2000.

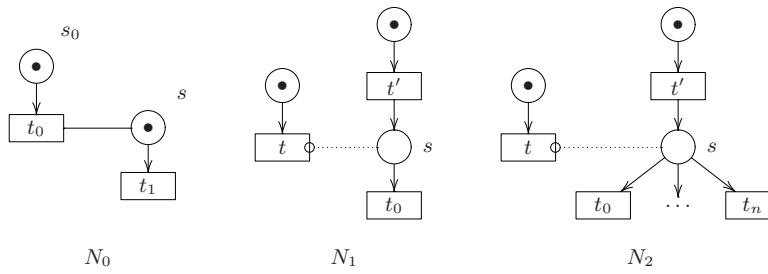


Fig. 1. Some basic contextual and inhibitor nets.

$$\textbf{S-N} \underset{\text{U}}{\overset{\perp}{\rightleftarrows}} \textbf{O-N} \underset{\text{E}}{\overset{\text{N}\ \perp}{\rightleftarrows}} \textbf{PES} \underset{\text{L}}{\overset{\text{P}\ \sim}{\rightleftarrows}} \textbf{Dom}$$

As shown in [10] essentially the same construction applies to the wider category of *semi-weighted* nets, i.e. P/T nets in which the initial marking and the post-set of each transition is a set, rather than a proper multiset.

The mentioned approach has been extended in [4] to nets with read arcs, referred to as *contextual nets* (see also [17]). The key problem for the treatment of contextual nets is illustrated by the net N_0 of Fig. 1 where the same place s is "read" by transition t_0 and "consumed" by transition t_1 (a read arc is represented by an undirected, horizontal line). The firing of t_1 prevents t_0 to be executed, so that t_0 can never follow t_1 in a computation, while the converse is not true, since t_1 *can* fire after t_0. This situation can be interpreted naturally as an *asymmetric conflict* between the two transitions and cannot be represented faithfully in a PES. To model the behaviour of contextual nets, the paper [4] introduces *asymmetric event structures* (AES's), an extension of prime event structures where the symmetric conflict is replaced by an asymmetric conflict relation. Such a feature is still necessary to be able to model the dependencies arising between events in a net with inhibitor arcs (also in the absence of read arcs). However the nonmonotonic features introduced by inhibitor arcs (negative conditions) make the situation far more complicated.

Consider the net N_1 in Fig. 1 where the place s, which inhibits transition t, is in the post-set of transition t' and in the pre-set of t_0 (an inhibitor arc is depicted as a dotted line from s to t, ending with an empty circle). The execution of t' inhibits the firing of t, which can be enabled again by the firing of t_0. Thus t can fire before or after the "sequence" $t'; t_0$, but not in between the two transitions. Roughly speaking there is a sort of atomicity of the sequence $t'; t_0$ w.r.t. t. The situation can be more involved since many transitions $t_0, \ldots, t_n$ may have the place s in their pre-set (see the net N_2 in Fig. 1). Therefore, after the firing of t', the transition t can be re-enabled by any of the conflicting transitions $t_0, \ldots, t_n$.

This leads to a sort of *or*-causality, but only when t fires after t'. With a logical terminology we can say that t causally depends on $t' \Rightarrow t_0 \vee t_1 \vee \ldots \vee t_n$.

To face these complications in this paper we introduce *inhibitor event structures* (IES's), a generalization of AES's equipped with a ternary relation $\rightarrowtriangle(\cdot,\cdot,\cdot)$, called *DE-relation (disabling-enabling relation)*, which allows one to model the dependency between transitions in N_2 as $\rightarrowtriangle(\{t'\}, t, \{t_0, \ldots, t_n\})$. The configurations of an IES, endowed with a computational order, form a prime algebraic domain, and Winskel's equivalence between **PES** and **Dom** generalizes to a coreflection between the category **IES** of inhibitor event structures and **Dom**.

As for ordinary and contextual nets, the connection between nets and event structures is established via an unfolding construction which maps each net into an occurrence net. Then the unfolding can be naturally abstracted to an IES, having the transitions of the net as events. The main difference with respect to the case of ordinary and contextual nets is the absence of a functor performing the backward step from IES's to occurrence inhibitor nets. Hence the problem of characterizing the passage from occurrence inhibitor nets to event structures as a coreflection, and thus of fully extending Winskel's approach to inhibitor nets, remains open. We refer the reader to [3,2] for a detailed treatment of the material presented in this paper and for a discussion of some related issues.

2 Inhibitor Event Structures

This section introduces the class of event structures that we consider adequate for modelling the complex phenomena which arise in the dynamics of inhibitor nets. Let us fix some notational conventions. The powerset of a set X is denoted by $\mathbf{2}^X$, while $\mathbf{2}^X_{fin}$ denotes the set of finite subsets of X and $\mathbf{2}^X_1$ the set of subsets of X of cardinality at most one. Hereafter generic subsets of events will be denoted by upper case letters $A, B, \ldots$, and singletons or empty subsets by $a, b, \ldots$

Definition 1 (pre-inhibitor event structure). *A* pre-inhibitor event structure *(pre-*IES*) is a pair $I = \langle E, \rightarrowtriangle \rangle$, where E is a set of* events *and* $\rightarrowtriangle \subseteq \mathbf{2}^E_1 \times E \times \mathbf{2}^E$ *is a ternary relation called* disabling-enabling relation (DE-relation).

Informally, if $\rightarrowtriangle(\{e'\}, e, A)$ then the event e' inhibits the event e, which can be enabled again by one of the events in A. The first argument of the relation can be also the empty set $\emptyset$, $\rightarrowtriangle(\emptyset, e, A)$ meaning that the event e is inhibited in the initial state of the system. Also the third argument A can be empty, $\rightarrowtriangle(\{e'\}, e, \emptyset)$ meaning that no events can re-enable e' after it has been disabled by e.

The DE-relation allows to represent both causality and asymmetric conflict and thus, concretely, it is the only relation of a (pre-)IES. In fact, if $\rightarrowtriangle(\emptyset, e, \{e'\})$ then the event e can be executed only after e' has been fired. This is exactly what happens in a PES when e' causes e, or in symbols when $e' < e$. More generally, if $\rightarrowtriangle(\emptyset, e, A)$ then we can imagine A as a set of disjunctive causes for e, since at least one of the events in A will appear in every history of the event e. This generalization of causality, restricted to the case in which the set A is pairwise conflictive (namely all distinct events in A are in conflict), will be represented in

symbols as $A < e$. Similar notions of or-causality have been studied in general event structures [18], flow event structures [5] and in bundle event structures [9].

Furthermore, if $\vdash\!\!\circ(\{e'\}, e, \emptyset)$ then e can never follow e' in a computation since there are no events which can re-enable e after the execution of e'. Instead the converse order of execution is admitted, namely e *can* fire before e'. This situation is naturally interpreted as an *asymmetric conflict* between the two events and it is written $e \nearrow e'$. It can be seen also as a *weak* form of *causal dependency*, in the sense that if $e \nearrow e'$ then e precedes e' in all computations containing both events. This explains why a rule below imposes asymmetric conflict to include (also generalized) causality, by asking that $A < e$ implies $e' \nearrow e$ for all $e' \in A$.

Finally, cycles of asymmetric conflict are used to define a notion of conflict on sets of events. If $e_0 \nearrow e_1 \ldots e_n \nearrow e_0$ then all such events cannot appear together in the same computation, since each one should precede the others. This fact is formalized via a conflict relation on sets of events $\#\{e_0, e_1 \ldots, e_n\}$. In particular, binary (symmetric) conflict is represented by asymmetric conflict in both directions.

Definition 2 (dependency relations). *Let $I = \langle E, \vdash\!\!\circ \rangle$ be a pre-*IES*. The relations of* (generalized) causality $< \subseteq \mathbf{2}^E \times E$, asymmetric conflict $\nearrow \subseteq E \times E$ *and* conflict $\# \subseteq \mathbf{2}^E_{fin}$ *are defined by the following set of rules:*

$$\frac{\vdash\!\!\circ(\emptyset, e, A) \quad \#_p A}{A < e}\ (<1) \qquad \frac{A < e \quad \forall e' \in A.\ A_{e'} < e' \quad \#_p(\cup\{A_{e'} \mid e' \in A\})}{(\cup\{A_{e'} \mid e' \in A\}) < e}\ (<2)$$

$$\frac{\vdash\!\!\circ(\{e'\}, e, \emptyset)}{e \nearrow e'}\ (\nearrow 1) \qquad \frac{e \in A < e'}{e \nearrow e'}\ (\nearrow 2) \qquad \frac{\#\{e, e'\}}{e \nearrow e'}\ (\nearrow 3)$$

$$\frac{e_0 \nearrow \ldots \nearrow e_n \nearrow e_0}{\#\{e_0, \ldots, e_n\}}\ (\#1) \qquad \frac{A' < e \quad \forall e' \in A'.\ \#(A \cup \{e'\})}{\#(A \cup \{e\})}\ (\#2)$$

where $\#_p A$ means that A is pairwise conflictive, namely $\#\{e, e'\}$ for all $e, e' \in A$ with $e \neq e'$. We will write $e \# e'$ for $\#\{e, e'\}$ and $e < e'$ for $\{e\} < e'$.

The basic rules (<1), $(\nearrow 1)$ and $(\#1)$, as well as $(\nearrow 2)$ and $(\nearrow 3)$ are justified by the discussion above. Rule (<2) generalizes the transitivity of the causality relation, while rule $(\#2)$ expresses a kind of hereditarity of the conflict with respect to causality.

An inhibitor event structure is a pre-IES where the DE-relation satisfies some further requirements suggested by its intended meaning, and causality and asymmetric conflict are induced "directly" by the DE-relation.

Definition 3 (inhibitor event structure). *An* inhibitor event structure (IES) *is a pre-*IES *$I = \langle E, \vdash\!\!\circ \rangle$ satisfying, for all $e \in E$, $a \in \mathbf{2}^E_1$ and $A \subseteq E$,*

1. *if $\vdash\!\!\circ(a, e, A)$ then $\#_p A$ and $\forall e' \in a.\ \forall e'' \in A.\ e' < e''$;*
2. *if $A < e$ then $\vdash\!\!\circ(\emptyset, e, A)$;*
3. *if $e \nearrow e'$ then $\vdash\!\!\circ(\{e'\}, e, \emptyset)$.*

Given a pre-IES I satisfying only (1) it is always possible to "saturate" the relation $\vdash\!\circ$ in order to obtain an IES, where the relations of causality and (asymmetric) conflict are exactly the same as in I. The *"saturated"* IES is defined as $\overline{I} = \langle E, \vdash\!\circ' \rangle$, where $\vdash\!\circ' = \vdash\!\circ \cup \{(\emptyset, e, A) \mid A < e\} \cup \{(\{e\}, e', \emptyset) \mid e \nearrow e'\}$.

Definition 4 (category IES). *Let $I_0 = \langle E_0, \vdash\!\circ_0 \rangle$ and $I_1 = \langle E_1, \vdash\!\circ_1 \rangle$ be two* IES*'s. An* IES*-morphism $f : I_0 \to I_1$ is a partial function $f : E_0 \to E_1$ such that for all $e_0, e_0' \in E_0$, $A_1 \subseteq E_1$, if $f(e_0)$ and $f(e_0')$ are defined then*

1. $(f(e_0) = f(e_0')) \wedge (e_0 \neq e_0') \quad \Rightarrow \quad e_0 \#_0 e_0'$;
2. $A_1 < f(e_0) \quad \Rightarrow \quad \exists A_0 \subseteq f^{-1}(A_1).\ A_0 < e_0$;
3. $\vdash\!\circ_1(\{f(e_0')\}, f(e_0), A_1) \quad \Rightarrow \quad \exists A_0 \subseteq f^{-1}(A_1).\ \exists a_0 \subseteq \{e_0'\}.\ \vdash\!\circ_0(a_0, e_0, A_0)$.

We denote by **IES** *the category of inhibitor event structures and* IES*-morphisms.*

Condition (1) is standard. Condition (2) generalizes the requirement of preservation of causes $\lfloor f(e) \rfloor \subseteq f(\lfloor e \rfloor)$ of PES (and AES) morphisms. Condition (3), as it commonly happens for event structures morphisms, just imposes the preservation of computations by asking, whenever some events in the image are constrained in some way, that stronger constraints are present in the pre-image. More precisely suppose that $\vdash\!\circ(\{f(e_0')\}, f(e_0), A_1)$. Thus we can have a computation where $f(e_0')$ is executed first and $f(e_0)$ can occur only after one of the events in A_1. Otherwise the computation can start with $f(e_0)$. According to condition (3), e_0 and e_0' are subject in I_0 to the same constraint of their images or, when $a_0 = \emptyset$ or $A_0 = \emptyset$, to stronger constraints selecting one of the possible orders of execution.

The category **PES** of prime event structures can be viewed as a full subcategory of **IES**. The full embedding functor $\mathrm{J}_i : \mathbf{PES} \hookrightarrow \mathbf{IES}$ maps each PES $P = \langle E, \leq, \# \rangle$ to the IES $\langle E, \vdash\!\circ \rangle$ where the DE-relation is defined by $\vdash\!\circ(\emptyset, e, \{e''\})$ if $e'' < e$ and $\vdash\!\circ(\{e'\}, e, \emptyset)$ if $e \# e'$. For any PES morphism $f : P_1 \to P_2$ its image is $\mathrm{J}_i(f) = f$. More generally, the category of asymmetric event structures [4] fully embeds into **IES** (see [2]), and also (extended) bundle event structures [9] and prime event structures with possible events [14] can be seen as special IES's.

3 Inhibitor Event Structures and Domains

The paper [18] shows that the categories **PES** of prime event structures and **Dom** of finitary prime algebraic domains are equivalent, via the functors $\mathrm{P} : \mathbf{Dom} \to \mathbf{PES}$ and $\mathrm{L} : \mathbf{PES} \to \mathbf{Dom}$. This section establishes a connection between IES's and finitary prime algebraic domains, by showing that the mentioned result generalizes to the existence of a categorical coreflection between **IES** and **Dom**. Then we study the problem of removing the non-executable events from an IES, by characterizing the full subcategory $\mathbf{IES}^{\mathbf{e}}$ consisting of the IES's where all events are executable, as a coreflective subcategory of **IES**.

The Domain of Configurations

The domain associated to an IES is obtained by considering the family of its configurations with a suitable order. To understand the notion of IES configuration, consider a set of events C of an inhibitor event structure I, and suppose $e', e, e'' \in C$ and $\vdash\!\circ(\{e'\}, e, A)$ for some A, with $e'' \in A$. Note that two distinct orders of execution of the three events are possible (either $e; e'; e''$ or $e'; e''; e$), which should not be confused from the point of view of causality. Hence, a configuration is not simply a set of events C, but some additional information must be added, in the form of a *choice relation*, to choose among the possible different orders of execution of events in C constrained by the DE-relation (e.g., in the above example a choice relation specifies wether e precedes e' or e'' precedes e).

We first introduce, for a given set of events C, the set $\mathit{choices}(C)$, a relation on C which "collects" *all* the possible precedences between events induced by the DE-relation. A choice relation for C is then a suitable subset of $\mathit{choices}(C)$.

Definition 5 (choice). *Let $I = \langle E, \vdash\!\circ \rangle$ be an* IES *and let $C \subseteq E$. We denote by $\mathit{choices}(C)$ the set*

$$\{(e, e') \mid \exists A.\ \vdash\!\circ_C(\{e'\}, e, A)\} \cup \{(e'', e) \mid \exists A.\ \vdash\!\circ_C(a, e, A) \wedge\ e'' \in A\} \subseteq C \times C,$$

where the restriction of $\vdash\!\circ(,,)$ to C is defined by $\vdash\!\circ_C(a, e, A)$ iff $\vdash\!\circ(a, e, A')$ for some A', with $e \in C$, $a \subseteq C$ and $A = A' \cap C$.
A choice *for C is an irreflexive relation $\hookrightarrow_C \subseteq \mathit{choices}(C)$ such that*

1. *if $\vdash\!\circ_C(a, e, A)$ then $\exists e' \in a.\ e \hookrightarrow_C e'$ or $\exists e'' \in A.\ e'' \hookrightarrow_C e$;*
2. *$(\hookrightarrow_C)^*$ is a finitary partial order.*

Condition (1) intuitively requires that whenever the DE-relation permits two possible orders of execution, the relation $\hookrightarrow_C$ chooses one of them. The fact that $\hookrightarrow_C \subseteq \mathit{choices}(C)$ ensures that $\hookrightarrow_C$ does not impose more precedences than necessary. Condition (2) guarantees that the precedences specified by $\hookrightarrow_C$ are not cyclic and that each event must be preceded only by finitely many others.

Definition 6 (configuration). *Let $I = \langle E, \vdash\!\circ \rangle$ be an* IES*. A* configuration *of I is a pair $\langle C, \hookrightarrow_C \rangle$, where $C \subseteq E$ and $\hookrightarrow_C \subseteq C \times C$ is a* choice *for C.*

It can be shown that the above definition generalizes the notion of PES and AES configuration since the property of admitting a choice implies causal closedness and conflict freeness. In the sequel, with abuse of notation, we will often denote a configuration and the underlying set of events with the same symbol C, referring to the corresponding choice relation as $\hookrightarrow_C$.

The computational order on configurations is a generalization of that introduced in [4] for AES's.

Definition 7 (extension). *Let $I = \langle E, \vdash\!\circ \rangle$ be an* IES *and let C and C' be configurations of I. We say that C'* extends *C, written $C \sqsubseteq C'$, if $C \subseteq C'$ and*

1. *$\forall e \in C.\ \forall e' \in C'.\ e' \hookrightarrow_{C'} e \quad \Rightarrow \quad e' \in C$;*
2. *$\hookrightarrow_C \subseteq \hookrightarrow_{C'}$.*

The poset of all configurations of I, ordered by extension, is denoted by $Conf(I)$.

As expressed by condition (1), a configuration C can be extended only by adding events which are not supposed to happen before other events already in C. Moreover, condition (2) ensures, together with (1), that the past history of events in C remains the same in C'. Indeed if $C \sqsubseteq C'$ then $\hookrightarrow_C = \hookrightarrow_{C'} \cap (C \times C)$, and thus, roughly speaking, C coincides with a "truncation" of C'.

The history of an event in a configuration C is formally defined as a subconfiguration of C. More precisely, for a configuration C and an event $e \in C$ the *history* of e in C is the configuration $\langle C[\![e]\!], \hookrightarrow_{C[\![e]\!]}\rangle$, where $C[\![e]\!] = \{e' \in C \mid e' \hookrightarrow_C^* e\}$ and $\hookrightarrow_{C[\![e]\!]} = \hookrightarrow_C \cap (C[\![e]\!] \times C[\![e]\!])$. Then it is possible to show that the poset of configurations of an IES has the desired algebraic structure.

Theorem 1 (domain of configurations). *For any* IES I *the poset $Conf(I)$ is a (finitary prime algebraic) domain. Its complete primes are the possible histories of events in I, i.e. $Pr(Conf(I)) = \{C[\![e]\!] \mid C \in Conf(I), e \in C\}$.*

The construction which associates the domain of configurations to an IES lifts to a functor from **IES** to **Dom**. Observe that since configurations are not simply sets of events it is not completely obvious, a priori, what should be the image of a configuration through a morphism. Let $f : I_0 \to I_1$ be an IES-morphism and let $\langle C_0, \hookrightarrow_0\rangle$ be a configuration of I_0. It is possible to show that $\hookrightarrow_1 = f(\hookrightarrow_0) \cap choices(f(C_0))$ is the the unique choice relation on $f(C_0)$ included in $f(\hookrightarrow_{C_0})$. Furthermore the function $f^* : Conf(I_0) \to Conf(I_1)$ which associates to each configuration $\langle C_0, \hookrightarrow_0\rangle$ the configuration $\langle f(C_0), \hookrightarrow_1\rangle$ is a domain morphism.

This means that the construction taking an IES into its domain of configurations can be viewed as a functor $\mathrm{L}_i : \mathbf{IES} \to \mathbf{Dom}$ defined as $\mathrm{L}_i(I) = Conf(I)$ for each IES I and $\mathrm{L}_i(f) = f^*$ for each IES-morphism $f : I_0 \to I_1$.

A functor $\mathrm{P}_i : \mathbf{Dom} \to \mathbf{IES}$ going back from domains to IES's can be obtained as the composition of Winskel's functor $\mathrm{P} : \mathbf{Dom} \to \mathbf{PES}$ with the full embedding $\mathrm{J}_i : \mathbf{PES} \to \mathbf{IES}$ defined at the end of Section 2.

Theorem 2 (coreflection IES $\hookrightarrow$ Dom). *The functor $\mathrm{P}_i : \mathbf{Dom} \to \mathbf{IES}$ is left adjoint to $\mathrm{L}_i : \mathbf{IES} \to \mathbf{Dom}$. The counit of the adjunction at an* IES I *is the function $\epsilon_I : \mathrm{P}_i \circ \mathrm{L}_i(I) \to I$, mapping each history of an event e into the event e itself, i.e., $\epsilon_I(C[\![e]\!]) = e$, for all $C \in Conf(I)$ and $e \in C$.*

The above result, together with Winskel's equivalence between the categories **Dom** of domains and **PES** of prime event structures, allows to translate an IES I into a PES $\mathrm{P}(\mathrm{L}_i(I))$. The PES is obtained from the IES essentially by introducing an event for each possible different history of events in the IES.

Removing Non-executable Events

The non-executability of events in an IES is not completely captured by the proof system of Definition 2, in the sense that we cannot derive $\#\{e\}$ for every non-executable event. Here we propose a semantic approach to rule out unused events from an IES, namely we simply remove from a given IES all events which do not appear in any configuration.

Definition 8. *We denote by* $\mathbf{IES^e}$ *the full subcategory of* **IES** *consisting of the* IES*'s* $I = \langle E, \mapsto \rangle$ *such that for any* $e \in E$ *there exists* $C \in Conf(I)$ *with* $e \in C$.

Any IES is turned into an $\mathbf{IES^e}$ object by forgetting the events which do not appear in any configuration.

Definition 9. *We denote by* $\Psi : \mathbf{IES} \to \mathbf{IES^e}$ *the functor mapping each* IES I *into the* $\mathbf{IES^e}$ *object* $\Psi(I) = \overline{\langle \psi(E), \mapsto_{\psi(E)} \rangle}$, *where* $\psi(E)$ *is the set of executable events in* I, *namely* $\psi(E) = \{e \in E \mid \exists C \in Conf(I).\ e \in C\}$. *Moreover if* $f : I_0 \to I_1$ *is an* IES*-morphism then* $\Psi(f) = f_{|\psi(E_0)}$. *With* $\mathrm{J}_{ies} : \mathbf{IES^e} \to \mathbf{IES}$ *we denote the inclusion.*

The inclusion of $\mathbf{IES^e}$ into **IES** is left adjoint to Ψ, i.e., $\Psi \vdash \mathrm{J}_{ies}$, and thus $\mathbf{IES^e}$ is a coreflective subcategory of **IES**. Furthermore the coreflection between **IES** and **Dom** restricts to a coreflection between $\mathbf{IES^e}$ and **Dom**, i.e., if $\mathrm{P}_i^{\mathbf{e}} : \mathbf{Dom} \to \mathbf{IES^e}$ and $\mathrm{L}_i^{\mathbf{e}} : \mathbf{IES^e} \to \mathbf{Dom}$ denote the restrictions of the functors P_i and L_i then $\mathrm{P}_i^{\mathbf{e}} \dashv \mathrm{L}_i^{\mathbf{e}}$.

4 A Category of Inhibitor Nets

Inhibitor nets are an extension of ordinary Petri nets where, by means of read and inhibitor arcs, transitions can check both for the presence and for the absence of tokens in places of the net. To give the formal definition we need some notation for multisets. Let A be a set; a *multiset* of A is a function $M : A \to \mathbb{N}$. The set of multisets of A is denoted by μA. The usual operations and relations on multisets, like multiset union $+$ or multiset difference $-$, are used. We write $M \leq M'$ if $M(a) \leq M'(a)$ for all $a \in A$. If $M \in \mu A$, we denote by $[\![M]\!]$ the multiset defined as $[\![M]\!](a) = 1$ if $M(a) > 0$ and $[\![M]\!](a) = 0$ otherwise; sometimes $[\![M]\!]$ will be confused with the corresponding subset $\{a \in A \mid [\![M]\!](a) = 1\}$ of A. A *multirelation* $f : A \to B$ is a multiset of $A \times B$. We will limit our attention to finitary multirelations, namely multirelations f such that the set $\{b \in B \mid f(a,b) > 0\}$ is finite. Multirelation f induces in an obvious way a function $\mu f : \mu A \to \mu B$, defined as $\mu f(\sum_{a \in A} n_a \cdot a) = \sum_{b \in B} \sum_{a \in A} (n_a \cdot f(a,b)) \cdot b$ (possibly partial, since infinite coefficients are disallowed). If f satisfies $f(a,b) \leq 1$ for all $a \in A$ and $b \in B$, i.e. $f = [\![f]\!]$, then we sometimes confuse it with the corresponding set-relation and write $f(a,b)$ for $f(a,b) = 1$.

Definition 10 (inhibitor net). *A* (marked) inhibitor Petri net (i-net) *is a tuple* $N = \langle S, T, F, C, I, m \rangle$, *where* S *is a set of* places, T *is a set of* transitions *(with* $S \cap T = \emptyset$*),* $F = \langle F_{pre}, F_{post} \rangle$ *is a pair of multirelations from* T *to* S, C *and* I *are relations between* T *and* S, *called the* context *and* inhibitor relation, *respectively, and* m *is a multiset of* S, *called the* initial marking. *If the inhibitor relation* I *is empty then* N *is called a* contextual net (c-net).

We require that for each $t \in T$, $F_{pre}(t,s) > 0$ for some place $s \in S$. Hereafter, when considering an i-net N, we will assume that $N = \langle S, T, F, C, I, m \rangle$. Subscripts on the net name carry over the names of the net components.

As usual, given a finite multiset of transitions $A \in \mu T$ we write ${}^\bullet A$ for its *pre-set* $\mu F_{pre}(A)$ and $A^\bullet$ for its *post-set* $\mu F_{post}(A)$. Moreover, by $\underline{A}$ we denote the *context* of A, defined as $\underline{A} = C([\![A]\!])$, and by ${}^\circledcirc A = I([\![A]\!])$ the *inhibitor set* of A. The same notation is used to denote the functions from S to $\mathbf{2}^T$ defined as, for $s \in S$, ${}^\bullet s = \{t \in T \mid F_{post}(t,s) > 0\}$, $s^\bullet = \{t \in T \mid F_{pre}(t,s) > 0\}$, $\underline{s} = \{t \in T \mid C(t,s)\}$ and ${}^\circledcirc s = \{t \in T \mid I(t,s)\}$.

Let N be an i-net. A finite multiset of transitions A is enabled at a marking M, if M contains the pre-set of A and an additional multiset of tokens which covers the context of A. Furthermore no token must be present nor produced by the transitions in the places of the inhibitor set of A. Formally, a finite multiset $A \in \mu T$ is *enabled* at M if ${}^\bullet A + \underline{A} \leq M$ and $[\![M + A^\bullet]\!] \cap {}^\circledcirc A = \emptyset$. In this case, to indicate that the execution of A in M produces the new marking $M' = M - {}^\bullet A + A^\bullet$ we write $M\,[A\rangle\, M'$. Step and firing sequences, as well as reachable markings are defined in the usual way.

Definition 11 (i-net morphism). *Let N_0 and N_1 be i-nets. An* i-net morphism $h : N_0 \to N_1$ *is a pair $h = \langle h_T, h_S\rangle$, where $h_T : T_0 \to T_1$ is a partial function and $h_S : S_0 \to S_1$ is a multirelation such that (1) $\mu h_S(m_0) = m_1$ and (2) for each $t \in T$,*

(a) $\mu h_S({}^\bullet t) = {}^\bullet h_T(t)$
(b) $\mu h_S(t^\bullet) = h_T(t)^\bullet$
(c) $\mu h_S(\underline{t}) = \underline{h_T(t)}$
(d) $[\![h_S]\!]^{-1}({}^\circledcirc h_T(t)) \subseteq {}^\circledcirc t$.

where $[\![h_S]\!]$ is the set relation underlying the multirelation h_S. We denote by **IN** *the category having i-nets as objects and i-net morphisms as arrows, and by* **CN** *its full subcategory of c-nets.*

Conditions (1), (2.a) - (2.b) are the defining conditions of Winskel's morphisms on ordinary nets, while (2.c) is the obvious condition which takes into account contexts. Condition (2.d) regarding the inhibitor arcs can be understood if we think of morphisms as simulations. Like preconditions and contexts must be preserved to ensure that the morphism maps computations of N_0 into computations of N_1, similarly, inhibitor conditions, which are negative conditions, must be reflected. In fact, condition (2.d) on inhibiting places can be rewritten as

$$s_1 \in [\![\mu h_S(s_0)]\!] \ \wedge\ I_1(h_T(t_0), s_1) \quad \Rightarrow \quad I_0(t_0, s_0),$$

which shows more explicitly that inhibitor arcs are reflected.

Proposition 1 (morphisms preserve the token game). *Let N_0 and N_1 be i-nets, and let $h = \langle h_T, h_S\rangle : N_0 \to N_1$ be an i-net morphism. For each $M, M' \in \mu S$ and $A \in \mu T$, if $M\,[A\rangle\, M'$ then $\mu h_S(M)\,[\mu h_T(A)\rangle\, \mu h_S(M')$. Therefore i-net morphisms preserve reachable markings, i.e. if M_0 is a reachable marking in N_0 then $\mu h_S(M_0)$ is reachable in N_1.*

As in [18,10,4] we will restrict our attention to a subclass of nets where each token produced in a computation has a uniquely determined history.

Definition 12 (semi-weighted and safe i-nets). *An i-net N is called* semi-weighted *if the initial marking m is a set and F_{post} is a relation (i.e., $t^\bullet$ is a set for all $t \in T$). We denote by* **SW-IN** *the full subcategory of* **IN** *having semi-weighted i-nets as objects; the corresponding subcategory of c-nets is denoted by* **SW-CN**. *A semi-weighted i-net is called* safe *if also F_{pre} is a relation and each reachable marking is a set.*

5 Occurrence I-Nets and the Unfolding Constructions

Generally speaking, an occurrence net provides a static representation of some computations of a net, in which the events (firing of transitions) and the relationships between events are made explicit. In [4] the notion of (nondeterministic) occurrence net has been generalized to the case of nets with read arcs. Here, the presence of the inhibitor arcs and the complex kind of dependencies they induce on transitions make it hard to find an appropriate notion of occurrence i-net.

We present two different, in our opinion both reasonable, notions of occurrence i-net and, correspondingly, we develop two unfolding constructions.

In the first construction, given an i-net N, we consider the underlying contextual net N_c obtained by forgetting the inhibitor arcs, and we apply to N_c the unfolding construction for contextual nets of [4], which produces an occurrence contextual net $\mathrm{U}_a(N_c)$. Then, if a place s and a transition t were originally connected by an inhibitor arc in the net N, then we insert an inhibitor arc between each copy of s and each copy of t in $\mathrm{U}_a(N_c)$, thus obtaining the unfolding $\mathrm{U}_i(N)$ of the net N. Then the characterization of the unfolding as a universal construction can be lifted from contextual to inhibitor nets. Furthermore, in this way the unfolding of an inhibitor net is decidable, in the sense that the problem of establishing if a possible transition occurrence actually appears in the unfolding is decidable. The price to pay is that some transitions in the unfolding may not be firable, since they are generated without taking care of inhibitor places.

In the second approach, the dependency relations (of causality and asymmetric conflict) for a net are defined only with respect to a fixed assignment for the net (playing a role similar to choices) which specifies for any inhibitor arc (t, s) if the inhibited transition t is executed before or after the place s is filled and in the second case which one of the transitions in the post-set $s^\bullet$ of the inhibitor place consumes the token. Then the firability of a transition t amounts to the existence of an assignment which is acyclic on the transitions which must be executed before t. Relying on this idea we can define a notion of occurrence net where each transition is really executable. The corresponding unfolding construction produces a net where the mentioned problem of the existence of non-firable transitions disappears, but, as a consequence of the Turing completeness of inhibitor nets, the produced unfolding is not decidable.

Lifting the Unfolding from Contextual to Inhibitor Nets

In the first approach, the unfolding construction disregards the inhibitor arcs.

Consequently the notion of occurrence i-net is defined without taking into account the dependencies between transitions induced by such kind of arcs.

Given a safe i-net N let us define the *read causality relation* as the least transitive relation $<_r$ on $S \cup T$ such that $s <_r t$ if $s \in {}^\bullet t$, $t <_r s$ if $s \in t^\bullet$, and $t <_r t'$ if $t^\bullet \cap \underline{t'} \neq \emptyset$, the only novelty with respect to ordinary nets being the last clause stating that a transition causally depends on transitions generating tokens in its context. The *read asymmetric conflict* $\nearrow_r$ is defined by taking $t \nearrow_r t'$ if t' consumes a token in the context of t, namely $\underline{t} \cap {}^\bullet t' \neq \emptyset$, in such a way that the firing of t' inhibits t. Moreover $t \nearrow_r t'$ if $(t \neq t' \ \wedge \ {}^\bullet t \cap {}^\bullet t' \neq \emptyset)$ to capture the usual symmetric conflict, and finally, according to the weak causality interpretation of the asymmetric conflict, $t \nearrow_r t'$ whenever $t <_r t'$.

Definition 13 (occurrence i-nets). *An* occurrence i-net *N is a safe i-net N where causality $\leq_r$ is a finitary partial order, asymmetric conflict $\nearrow_r$ is acyclic on the causes of each transition, for all $s \in S$ $|{}^\bullet s| \leq 1$ and $m = \{s \in S \mid {}^\bullet s = \emptyset\}$.* **O-IN** *denotes the full subcategory of* **SW-IN** *having occurrence i-nets as objects.*

Let us consider a functor $\mathrm{R}_{ic} : \mathbf{SW\text{-}IN} \to \mathbf{SW\text{-}CN}$ which maps each i-net into the underlying c-net, forgetting the inhibitor relation, and the inclusion $\mathrm{I}_{ci} : \mathbf{SW\text{-}CN} \to \mathbf{SW\text{-}IN}$ (see the diagram below).

$$\mathbf{O\text{-}CN} \underset{\mathrm{U}_a}{\overset{\perp}{\rightleftarrows}} \mathbf{SW\text{-}CN} \underset{\mathrm{R}_{ic}}{\overset{\mathrm{I}_{ci}}{\rightleftarrows}} \mathbf{SW\text{-}IN} \xleftarrow{\mathrm{I}_O} \mathbf{O\text{-}IN}$$

The relations $\leq_r$ and $\nearrow_r$ for an i-net N are exactly the relations of causality and asymmetric conflict of the underlying c-net $\mathrm{R}_{ic}(N)$, as defined in [4]. Thus the notion of occurrence c-net in this paper (i.e., occurrence i-net without inhibitor arcs) coincides with that of [4]. Moreover an occurrence i-net is a safe i-net N such that $\mathrm{R}_{ic}(N)$ is an occurrence c-net. Let **O-CN** be the category of occurrence c-nets, namely the full subcategory of **O-IN** having c-nets as objects. The paper [4] defines an unfolding functor $\mathrm{U}_a : \mathbf{SW\text{-}CN} \to \mathbf{O\text{-}CN}$, mapping each semi-weighted c-net to an occurrence c-net. The functor U_a is shown there to be right adjoint to the inclusion functor of **O-CN** into **SW-CN**. Using the functors R_{ic} and I_{ci} we can lift both the construction and the result to inhibitor nets.

Definition 14 (unfolding). *Let N be a semi-weighted i-net. Consider the occurrence c-net $\mathrm{U}_a(\mathrm{R}_{ic}(N)) = \langle S', T', F', C', \emptyset, m' \rangle$ and the folding morphism $f_N : \mathrm{U}_a(\mathrm{R}_{ic}(N)) \to \mathrm{R}_{ic}(N)$. Define an inhibitor relation on the net $\mathrm{U}_a(\mathrm{R}_{ic}(N))$ by taking for $s' \in S'$ and $t' \in T'$, $I'(s', t')$ iff $I(f_N(s'), f_N(t'))$. Then the unfolding $\mathrm{U}_i(N)$ of the i-net N is the occurrence i-net $\langle S', T', F', C', I', m' \rangle$ and the* folding morphism *is given by f_N seen as a morphism from $\mathrm{U}_i(N)$ to N.*

The i-net $\mathrm{U}_i(N)$ can be shown to be the *least* occurrence i-net which extends $\mathrm{U}_a(\mathrm{R}_{ic}(N))$ with the addition of inhibitor arcs in a way that $f_N : \mathrm{U}_i(N) \to N$ is a well defined i-net morphism.

Theorem 3. *The unfolding extends to a functor* U_i : **SW-IN** $\to$ **O-IN** *which is right adjoint to the obvious inclusion functor* I_O : **O-IN** $\to$ **SW-IN**, *thus establishing a coreflection between* **SW-IN** *and* **O-IN**. *The component at an object* N *in* **SW-IN** *of the counit of the adjunction,* $f : \mathrm{I}_O \circ \mathrm{U}_i \dot{\to} 1$, *is the folding morphism* $f_N : \mathrm{U}_i(N) \to N$.

Executable Occurrence I-Nets

The second approach is inspired by the notion of deterministic process of an i-net introduced in [6], where the inhibitor arcs of the net underlying a process are partitioned into two subsets: the *before* inhibitor arcs and the *after* inhibitor arcs. Then the dependencies induced by such a partition are required to be acyclic in order to guarantee the firability of all the transitions of the net in a single computation. Following this idea, to ensure that each transition of a nondeterministic occurrence net is firable in *some* computation, we require, for each transition t, the *existence* of a so-called assignment which partitions the inhibitor arcs into before and after arcs, without introducing cyclic dependencies on the transitions which must be executed before t.

Definition 15 (assignment). *Let* N *be a safe i-net. An* assignment *for* N *is a function* $\rho : I \to T$ *such that, for all* $(t, s) \in I$, $\rho(t, s) \in {}^\bullet s \cup s^\bullet$.

Intuitively, an assignment ρ specifies for each inhibitor arc (t, s), if the transition t fires *before* or *after* the place s receives a token. If $\rho(t, s) \in {}^\bullet s$ then (t, s) is a before arc, while if $\rho(t, s) \in s^\bullet$ then (t, s) is an after arc. In the last case, since we are considering possibly nondeterministic nets and thus the place s may be in the pre-set of several transitions, the assignment specifies also which of the transitions in $s^\bullet$ consumes the token.

Given a safe net N, once an assignment ρ for N is fixed, new dependencies arise between the transitions of the net, formalized by means of the relations $\prec_i^\rho$ and $\nearrow_i^\rho$. We define $t \prec_i^\rho t'$ iff $\exists s \in {}^{\circledcirc}t' \cap {}^\bullet t.\ \rho(t', s) = t$ and $t \nearrow_i^\rho t'$ iff $\exists s \in {}^{\circledcirc}t \cap t'^\bullet.\ \rho(t, s) = t'$. Observe that, as suggested by the adopted symbols, the additional dependencies can be seen as a kind of causality and asymmetric conflict, respectively. In fact if $t \prec_i^\rho t'$, then t' can happen only after t has removed the token from s, and thus t acts as a cause for t'. If $t \nearrow_i^\rho t'$ then if both t and t' happen in the same computation then necessarily t occurs before t', since t' generates a token in a place s which inhibits t, while according to the interpretation of ρ, t must occur before the place s is filled.

Under a fixed assignment ρ, we can introduce a kind of generalized causality and asymmetric conflict by joining the relations $\leq_r$ and $\nearrow_r$ defined before with the additional dependencies induced by the inhibitor arcs. We define $<_\rho = (<_r \cup \prec_i^\rho)^+$ and $\lhd_\rho = <_\rho \cup \nearrow_r \cup \nearrow_i^\rho$, i.e., $\lhd_\rho$ records both kinds of dependency. Furthermore, for $x \in S \cup T$ we denote by $\lfloor x \rfloor_\rho$ the set $\{t \in T \mid t \leq_\rho x\}$, and similarly, for $X \subseteq S \cup T$, we define $\lfloor X \rfloor_\rho = \bigcup \{\lfloor x \rfloor_\rho \mid x \in X\}$.

Definition 16 (executable occurrence i-net). *An* executable occurrence i-net *is a safe i-net* N *such that (i) for all* $t \in T$ *there exists an assignment*

ρ such that $(\lhd_\rho)_{\lfloor t \rfloor_\rho}$ is acyclic and $\lfloor t \rfloor_\rho$ is finite, (ii) for all $s \in S$, $|{}^\bullet s| \leq 1$, and (iii) $m = \{s \in S \mid {}^\bullet s = \emptyset\}$.

Hence executable occurrence i-nets refine occurrence i-nets by considering also the dependencies induced by inhibitor arcs. We denote by **O-IN$^{\mathbf{e}}$** the full subcategory of **O-IN** having executable occurrence i-nets as objects.

Definition 17 (concurrency). *A set of places $M \subseteq S$ is called* concurrent*, written $conc(M)$, if there exists an assignment ρ such that (i) for all $s, s' \in M$ $\neg(s <_\rho s')$, (ii) $\lfloor M \rfloor_\rho$ is finite and (iii) $\lhd_\rho$ acyclic on $\lfloor M \rfloor_\rho$.*

As for ordinary and contextual nets, a set of places M is concurrent if and only if there is a reachable marking in which all the places of M contain a token. Consequently each transition of an executable occurrence i-net can fire in some computation (and thus each place contains a token at some reachable marking), a property which justifies the name "executable".

Proposition 2. *Let N be an executable occurrence i-net. Then for each transition $t \in T$ there exists a reachable marking M such that t is enabled at M.*

We can now introduce a different unfolding construction, that, when applied to a semi-weighted i-net N, produces an executable occurrence i-net.

Definition 18 ((executable) unfolding). *Let N be a semi-weighted i-net. The* (executable) unfolding $\mathrm{U}_i^{\mathbf{e}}(N) = \langle S', T', F', C', I', m' \rangle$ *of the net N and the* folding morphism $f_N = \langle f_T, f_S \rangle : \mathrm{U}_i^{\mathbf{e}}(N) \to N$ *are the unique executable occurrence i-net and i-net morphism satisfying the following equations:*

$$
\begin{array}{l}
m' = \{\langle \emptyset, s \rangle \mid s \in m\} \\
S' = m' \cup \{\langle t', s \rangle \mid t' \in T' \ \wedge \ s \in f_T(t')^\bullet\} \\
T' = \{t' \mid t' = \langle M_p, M_c, t \rangle \ \wedge \ t \in T \ \wedge \ M_p \cup M_c \subseteq S' \ \wedge \ M_p \cap M_c = \emptyset \\
\qquad \wedge \ conc(M_p \cup M_c) \wedge \ \mu f_S(M_p) = {}^\bullet t \ \wedge \ \mu f_S(M_c) = \underline{t} \\
\qquad \wedge \ \exists \rho.\ (\lfloor t' \rfloor_\rho \text{ finite } \wedge \ \lhd_\rho \text{ acyclic on } \lfloor t' \rfloor_\rho)\}
\end{array}
$$

$$
\begin{array}{lll}
F'_{pre}(t', s') & \text{iff} & t' = \langle M_p, M_c, t \rangle \ \wedge \ s' \in M_p \quad (t \in T) \\
F'_{post}(t', s') & \text{iff} & s' = \langle t', s \rangle \quad (s \in S) \\
C'(t', s') & \text{iff} & t' = \langle M_p, M_c, t \rangle \ \wedge \ s' \in M_c \quad (t \in T) \\
I'(t', s') & \text{iff} & f_S(s', s) \ \wedge \ I(f_T(t'), s) \\
\\
f_T(t') = t & \text{iff} & t' = \langle M_p, M_c, t \rangle \\
f_S(s', s) & \text{iff} & s' = \langle x, s \rangle \quad (x \in T' \cup \{\emptyset\})
\end{array}
$$

As usual, places and transitions in the unfolding represent tokens and firing of transitions in the original net. Each item of the unfolding is a copy of an item in the original net, enriched with the corresponding "history". The folding morphism f maps each item of the unfolding to the corresponding item in the original net. The unfolding can be given also an inductive definition, from which uniqueness easily follows.

The two proposed unfolding constructions are tightly related, in the sense that $\mathrm{U}_i^{\mathbf{e}}(N)$ can be obtained from $\mathrm{U}_i(N)$ simply by removing the non executable transitions. This fact is formalized by defining a "pruning" functor $\Pi : \mathbf{O\text{-}IN} \to \mathbf{O\text{-}IN^e}$ which removes the non executable transitions from a general occurrence i-net thus producing an executable occurrence i-net. The functor $\Pi : \mathbf{O\text{-}IN} \to \mathbf{O\text{-}IN^e}$ is right adjoint to the inclusion functor $\mathrm{J}^{\mathbf{e}} : \mathbf{O\text{-}IN^e} \to \mathbf{O\text{-}IN}$, and thus $\mathbf{O\text{-}IN^e}$ is a coreflective subcategory of **O-IN**. Then one can formally state the relationship between $\mathrm{U}_i^{\mathbf{e}}(N)$ and $\mathrm{U}_i(N)$, providing also an indirect proof of the universality of the new unfolding construction.

Proposition 3. *For any semi-weighted i-net N, $\mathrm{U}_i^{\mathbf{e}}(N) = \Pi(\mathrm{U}_i(N))$. Therefore $\mathrm{U}_i^{\mathbf{e}}$ is right adjoint to the inclusion functor and they establish a coreflection between* **SW-IN** *and* $\mathbf{O\text{-}IN^e}$.

6 Inhibitor Event Structure Semantics for I-Nets

In this section we define an event structure and a domain semantics for i-nets by relating occurrence i-nets and inhibitor event structures. The dependencies arising among transitions in an occurrence i-net can be naturally represented by the DE-relation, and therefore the IES corresponding to an occurrence i-net is obtained by forgetting the places and taking the transitions of the net as events.

Definition 19. *Let N be an occurrence i-net. The pre-*IES *associated to N is defined as $I_N^p = \langle T, \mapsto_N^p \rangle$, with $\mapsto_N \subseteq \mathbf{2}_1^T \times T \times \mathbf{2}^T$, given by: for $t, t' \in T$, $t \neq t'$ and $s \in S$*

1. *if $t^\bullet \cap ({}^\bullet t' \cup \underline{t'}) \neq \emptyset$ then $\mapsto_N^p(\emptyset, t', \{t\})$*
2. *if $({}^\bullet t \cup \underline{t}) \cap {}^\bullet t' \neq \emptyset$ then $\mapsto_N^p(\{t'\}, t, \emptyset)$;*
3. *if $s \in {}^{\circledcirc}t$ then $\mapsto_N^p({}^\bullet s, t, s^\bullet)$.*

The IES *associated to N, denoted by $\mathrm{E}_i(N) = \langle T, \mapsto_N \rangle$, is obtained by saturating I_N^p, i.e., $\mathrm{E}_i(N) = \overline{I_N^p}$.*

Clauses (1) and (2) encode, by using the DE-relation, the causal dependencies and the asymmetric conflicts induced by the flow and read arcs (we could have written if $t <_r t'$ then $\mapsto_N^p(\emptyset, t', \{t\})$ and if $t \nearrow_r t'$ then $\mapsto_N^p(\{t'\}, t, \emptyset)$). The last clause fully exploits the expressiveness of the DE-relation to represent the dependencies induced by inhibitor places.

Since the transition component of an i-net morphism is an IES-morphism between the corresponding IES's we have the following result.

Proposition 4. *The construction which maps each i-net N to the corresponding* IES *$\mathrm{E}_i(N)$ can be extended to a functor $\mathrm{E}_i : \mathbf{O\text{-}IN} \to \mathbf{IES}$ by defining $\mathrm{E}_i(h) = h_T$ for each morphism $h : N_0 \to N_1$.*

One can verify that if N is an executable occurrence i-net then $\mathrm{E}_i(N)$ is an $\mathbf{IES^e}$ object, and thus the functor E_i restricts to a functor $\mathrm{E}_i^{\mathbf{e}} : \mathbf{O\text{-}IN^e} \to \mathbf{IES^e}$.

The converse step, from IES's to occurrence i-nets, instead, turns out to be very problematic. An object level constructions can be defined, associating to any IES a corresponding i-net. However the problem of finding a functorial construction (if any) is still unsolved. See [2,3] for a wider discussion suggesting how the difficulties are intimately connected to or-causality.

7 Conclusions

We have defined a functorial concurrent semantics for semi-weighted Petri nets with read and inhibitor arcs. The proposed constructions, which generalize Winskel's work on safe ordinary nets and the work in [4] on contextual nets, are summarized in Fig. 2. Unfortunately, the objective of providing a coreflective

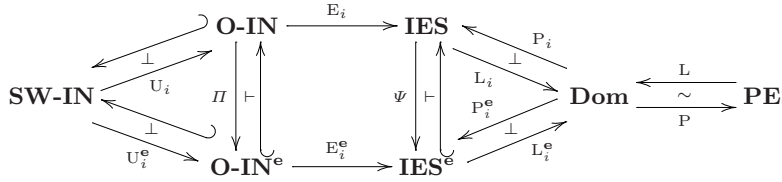


Fig. 2. A summary of the constructions in the paper (unnamed functors are inclusions).

semantics for inhibitor nets is partially missed, since the construction mapping each occurrence i-net to an IES is not expressed as a coreflection. Hence the problem of fully extending to i-nets Winskel's chain of coreflections remains open.

Acknowledgements

We are grateful to the anonymous referees for their useful comments on the submitted version of this paper.

References

1. T. Agerwala and M. Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. *Computer Architecture News*, 4(2):81–86, 1973. 442
2. P. Baldan. *Modelling concurrent computations: from contextual Petri nets to graph grammars*. PhD thesis, Department of Computer Science, University of Pisa, 2000. 442, 444, 446, 456
3. P. Baldan, N. Busi, A. Corradini, and G. M. Pinna. Domain and event structure semantics for Petri nets with read and inhibitor arcs. Technical report TR-00-05, Department of Computer Science, University of Pisa, 2000. 442, 444, 456

4. P. Baldan, A. Corradini, and U. Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. *Proceedings of FoSSaCS '98*, *LNCS* 1378. Springer, 1998. 443, 446, 447, 450, 451, 452, 456
5. G. Boudol. Flow Event Structures and Flow Nets. In *Semantics of System of Concurrent Processes*, *LNCS* 469. Springer, 1990. 445
6. N. Busi and G. M. Pinna. Process semantics for Place/Transition nets with inhibitor and read arcs. *Fundamenta Informaticæ*, 40(2-3):165–197, 1999. 453
7. S. Christensen and N. D. Hansen. Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. *Applications and Theory of Petri Nets*, *LNCS* 691. Springer, 1993. 442
8. R. Janicki and M. Koutny. Semantics of inhibitor nets. *Information and Computation*, 123:1–16, 1995. 442
9. R. Langerak. *Transformation and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, 1992. 445, 446
10. J. Meseguer, U. Montanari, and V. Sassone. Process versus unfolding semantics for Place/Transition Petri nets. *Theoret. Comp. Sci.*, 153(1-2):171–210, 1996. 443, 450
11. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6), 1995. 442
12. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoret. Comp. Sci.*, 13:85–108, 1981. 442
13. J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981. 442
14. G. M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoret. Comp. Sci.*, 138(2):425–454, 1995. 446
15. W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer, 1985. 442
16. W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. In *Proceedings of ICALP'97*, *LNCS* 1256. Springer, 1997. 442
17. W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proceedings of CONCUR'98*, *LNCS* 1466. Springer, 1998. 443
18. G. Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, *LNCS* 255. Springer, 1987. 442, 445, 446, 450

The Control of Synchronous Systems*

Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang

Electrical Engineering and Computer Sciences, University of California at Berkeley
{dealfaro,tah,fmang}@eecs.berkeley.edu

Abstract. In the synchronous composition of processes, one process may prevent another process from proceeding unless compositions without a well-defined product behavior are ruled out. They can be ruled out semantically, by insisting on the existence of certain fixed points, or syntactically, by equipping processes with types, which make the dependencies between input and output signals transparent. We classify various typing mechanisms and study their effects on the control problem.

A static type enforces fixed, acyclic dependencies between input and output ports. For example, synchronous hardware without combinational loops can be typed statically. A dynamic type may vary the dependencies from state to state, while maintaining acyclicity, as in level-sensitive latches. Then, two dynamically typed processes can be syntactically compatible, if all pairs of possible dependencies are compatible, or semantically compatible, if in each state the combined dependencies remain acyclic. For a given plant process and control objective, there may be a controller of a static type, or only a controller of a syntactically compatible dynamic type, or only a controller of a semantically compatible dynamic type. We show this to be a strict hierarchy of possibilities, and we present algorithms and determine the complexity of the corresponding control problems.

Furthermore, we consider versions of the control problem in which the type of the controller (static or dynamic) is given. We show that the solution of these fixed-type control problems requires the evaluation of partially ordered (Henkin) quantifiers on boolean formulas, and is therefore harder (nondeterministic exponential time) than more traditional control questions.

1 Introduction

The formulation of the control problem builds on the notion of parallel composition: given a transition system M (the "plant"), is there a transition system N (the "controller") such that the compound system $M \| N$ meets a given objective? Hence it is not surprising that even small variations in the definition of composition may influence the outcome of the control problem, as well as the

* This research was supported in part by the DARPA grants NAG2-1214 and F33615-C-98-3614, the SRC contract 99-TJ-683.003, the MARCO grant 98-DT-660, and the NSF CAREER award CCR-9501708.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 458–473, 2000.

hardness of its solution. (The latter distinguishes control from verification, whose complexity —PSPACE for invariant verification— is remarkably resilient against changes in the definition of parallel composition.) At the highest level, one can distinguish between asynchronous and synchronous forms of composition. Pure asynchronous (or interleaving) composition is disjunctive: one component proceeds at a time, so that an action of the compound system is an action of some component. Pure synchronous (or lock-step) composition is conjunctive: all components proceed simultaneously, so that an action of the compound system is a tuple of actions, one for each component. While many concurrency models exhibit mixed forms of composition (e.g., interleaving of internal actions and synchronization of communication actions [Mil89]), it is natural to start by considering the control problem for the two pure forms of composition. The study of these control problems corresponds to the study of winning conditions of games, where the two players (plant vs. controller) choose moves (actions) to prevent (resp. accomplish) the control objective.

In practice, the most important control objective is invariance: the controller strives to forever keep the plant within a safe set of states. The problem of invariance control can be solved by a fixed-point iteration: first, we find a strategy that keeps the plant safe for a single step; then, a strategy that keeps the plant safe for two steps; etc. We henceforth refer to invariance control as the "multi-step" control problem, and to the problem of keeping the plant safe for a single step, as the "single-step" control problem. This allows us to separate concerns: the definition of parallel composition enters the solution of the single-step problem, but independently of the type of composition, the multi-step problem can always be solved by iteratively solving single-step problems. In other words, we can independently study (1) the single-step control problem, and the definition of parallel composition plays a central role in this study, *or* (2) the multi-step control problem (for invariance or even more general, ω-regular objectives), assuming to be given a solution to the single-step problem. While (2) has been researched extensively in the literature [BL69,GH82,RW87,EJ91,McN93,TW94,Tho95], it is (1) we focus on in this paper.

We assume that the plant M is specified in a compact form, by a transition predicate on boolean variables, so that the state space of M is exponentially larger than the description of M, which is the input to the control problem. For solving the multi-step control problem, the number of single-step iterations is bound by the number of states. Therefore, if the single-step problem can be solved in exponential time, then so can the multi-step problem. Conversely, it can be shown that even if the single-step problem can be solved in constant time, the multi-step problem is still complete for EXP (deterministic exponential time). This seems to indicate that the single-step problem is of little interest, and it may explain why not much attention has been paid to the single-step problem previously. To our surprise, we found that for certain natural forms of parallel composition, the single-step control problem can *not* be solved in (deterministic) exponential time, and therefore its complexity dominates also the one of multi-step control.

An essential property of systems is to be *non-blocking*, in the sense that every state should have at least one successor state [BG88,Hal93,Kur94,Lyn96]. Non-blocking is essential for compositional techniques such as assume-guarantee reasoning [AL95,McM97,AH99]. In control, non-blocking means that the controller should never prevent the plant from moving. While the asynchronous composition of non-blocking processes is always non-blocking, synchronous composition needs to be restricted to ensure non-blocking. A second kind of restriction arises from modeling "typed" components, where the type specifies the input ports and output ports of a component, as well as permissible and impermissible dependencies between input and output signals [AH99]. In particular, hardware components are usually typed in this way, for example, in order to avoid combinational loops. In control, if we restrict our attention to typed controllers, then a controller may not exist even when an untyped controller would exist. These two kinds of common restrictions on synchronous composition, non-blocking and typing, are related, as typing can be used for syntactically enforcing the semantic concept of non-blocking for synchronously composed systems.

If the plant is given by a boolean transition predicate, and parallel composition is asynchronous, then single-step control amounts to evaluating the conjunction of a $\forall$ formula ("all actions of the plant are safe") and an $\exists$ formula ("some action of the controller is safe"). Hence, the complexity class of asynchronous single-step control is DP (which contains the differences of languages in NP). For synchronous systems, the various restrictions on composition give rise to different control problems. One way of ensuring non-blocking is to consider only Moore processes. A Moore process is a non-blocking process in which the next values of the output signals do not depend on the next values of the input signals. The composition of Moore processes is again Moore, and therefore non-blocking. If both the system and the controller are Moore processes, then the single-step control formula has the quantifier prefix $\exists\forall$ ("the controller can choose the new input signals, so that regardless of the new output signals, the system is safe").

A more liberal way of ensuring non-blocking is to consider typed processes, i.e., processes that explicitly specify the dependencies between the new values of input and output signals. We distinguish between "static" types, where the input-output dependencies are fixed, and "dynamic" types, where the dependencies can change from state to state. Dynamic types may be composed either "syntactically" (by requiring that all possible combinations of dependency relations of the component processes are acyclic), or "semantically" (by requiring acyclicity at all states of the compound system). Both static and dynamic types ensure that the compound system is again typed, and therefore non-blocking. We consider two variants of the typed control problems: one in which we are free to choose both the controller and its type, and one in which we must find a controller of a specified type. If we can choose the type of the controller, the control problem can be solved by considering for the controller an exponential number of types of a simple form, namely, types that represent linearly ordered input-output dependencies. The single-step control problem resulting

from each linear order of dependencies gives rise to a boolean formula with a linear quantifier prefix, with any number of $\forall\exists$ alternations, which puts the problem into PSPACE. If the type of the desired controller is given, the single-step control problem becomes considerably harder. This is because a (static or dynamic) type may specify partially ordered input-output dependencies. These partially-ordered dependencies correspond to boolean formulas with partially ordered (Henkin) quantifiers [Hen61,Wal70,BG86], whose complexity class for satisfiability is NE (a weak form of nondeterministic exponential time) [GLV95].

The solution of control problems in presence of types gives rise to additional surprising phenomena. For example, with static or syntactically composed dynamic types, two states s and t may both be controllable even though there is not a single controller that controls both s and t (two different controllers are required). Hence, while types provide an efficient mechanism for ensuring the non-blocking of synchronously composed systems, they cause difficulties in control. On the other hand, these difficulties are often not artificial, but they correspond to real input/output constraints in the design of controllers.

2 Types for Synchronous Composition

Preliminaries. Let X be a set of variables. In this paper we consider all variables to range over the set $\mathbb{B}$ of booleans. We write $X' = \{x' \mid x \in X\}$ for the set of corresponding primed variables. A state s over X is a truth-value assignment $s\colon X \mapsto \mathbb{B}$ to the variables in X. We write s' for the truth-value assignment $s'\colon X' \mapsto \mathbb{B}$ defined by $s'(x') = s(x)$ for all $x \in X$. Given a subset $Y \subseteq X$, we write $s[Y]$ for the restriction of s to the variables in Y. Given a predicate φ over the variables in X, we write $\varphi[\![s]\!]$ for the truth value of φ when the variables in X are interpreted according to s. Given a predicate τ over the variables in $X \cup X'$, and states s, t over X, we write $\tau[\![s, t']\!]$ for the truth value of τ when the variables in X are interpreted according to s, and the variables in X' are interpreted according to t'.

Modules and composition. A *module* M consists of the following three components:

- A finite set X^o_M of *output variables*. These variables are updated by the module.
- A finite set X^i_M of *input variables*. These variables are updated by the environment. The sets X^o_M and X^i_M must be disjoint. We write $X_M = X^o_M \cup X^i_M$ for the set of all module variables. The *states* of M are the truth-value assignments to the variables in X_M.
- A predicate τ_M over the set $X_M \cup X'_M$ of unprimed and primed variables. The predicate τ_M, called *transition predicate*, relates the current (unprimed) and next (primed) values of the module variables.

Two modules M and N are *composable* if their output variables X^o_M and X^o_N are disjoint. Given two composable modules M and N, the *synchronous* (*lock-step*)

composition $M||N$ is the module with the components $X^o_{M||N} = X^o_M \cup X^o_N$, $X^i_{M||N} = (X^i_M \cup X^i_N) \setminus X^o_{M||N}$, and $\tau_{M||N} = (\tau_M \wedge \tau_N)$. The *asynchronous* (*interleaving*) *composition* $M|N$ is the module with the same output and input variables as $M||N$, but with the transition predicate $\tau_{M|N} = ((\tau_M \wedge (X'^o_N = X^o_N)) \vee (\tau_N \wedge (X'^o_M = X^o_M)))$.

Non-blocking modules. We are interested in the non-blocking modules, a condition necessary for compositional techniques [AH99]. A module M is *non-blocking* if every state has a successor; that is, for each state s there is a state t such that $\tau_M[\![s, t']\!]$. The asynchronous composition of two composable non-blocking modules is again non-blocking. Hence, we say that any two composable non-blocking modules are *async-composable*. However, there are composable non-blocking modules whose synchronous composition is not non-blocking.

Example 1. Let module M be such that $X^o_M = \{x\}$, $X^i_M = \{y\}$, and $\tau_M = ((y' \wedge \neg x') \vee (\neg y' \wedge x'))$. Let module N be such that $X^o_N = \{y\}$, $X^i_N = \{x\}$, and $\tau_N = ((x' \wedge y') \vee (\neg x' \wedge \neg y'))$. Then M and N are non-blocking and composable. However, the transition predicate of $M||N$ is unsatisfiable, i.e., no state of $M||N$ has a successor. □

It requires exponential time to check if a module M is non-blocking, which amounts to evaluating the boolean Π^p_2 formula $(\forall X_M)(\exists X'_M)\, \tau_M$. To eliminate the need for this exponential check whenever two modules are composed synchronously, we define four increasingly larger classes of modules for which the non-blocking of synchronous composition can be checked efficiently.

Moore modules. A *Moore module* is a module that (a) is non-blocking, and (b) such that the next values of output variables do not depend on the next values of input variables; that is, for all states s, t, and u, if $\tau_M[\![s, t']\!]$ and $t[X^o_M] = u[X^o_M]$, then $\tau_M[\![s, u']\!]$. These two conditions can be enforced syntactically, in a way that permits checking in linear time. For example, the transition predicate of a Moore module can be specified as a set of nondeterministic guarded commands, one for each primed output variable x' in X'^o_M. The guarded command for x' assigns a value to x' such that (a) one of the guards negates the disjunction of the other guards, and (b) the guards and the right-hand sides of all assignments contain no primed variables. The synchronous composition of two composable Moore modules is again a Moore module, and therefore non-blocking. Hence, we say that any two composable Moore modules are *moore-composable*. However, since many non-blocking modules are not Moore modules, more general types of modules are of interest.

Statically typed modules (or Reactive Modules [AH99]). A *dependency relation* for a module M is an acyclic binary relation $\succ \subseteq X^o_M \times X_M$ between the output variables and the module variables (acyclicity means that the transitive closure is irreflexive). The module M *respects* the dependency relation $\succ$ at state s if, for all states t with $\tau_M[\![s, t']\!]$, for each subset $Y^i \subseteq X^i_M$ of input variables, and for each truth-value assignment u^i to the variables in Y^i, there is a state u with $\tau_M[\![s, u']\!]$ such that $u[Y^i] = u^i$, and $u[Z] = t[Z]$ for

$Z = \{z \in X_M \mid (\text{not } z \succ^* y) \text{ for all } y \in Y^i\}$, where $\succ^*$ is the reflexive-transitive closure of $\succ$. A *statically typed module* $(M, \succ_M)$ consists of a module M and a dependency relation for M, such that (a) the module M is non-blocking, and (b) the module M respects the dependency relation $\succ_M$ at all states. These two conditions, as well as the acyclicity requirement on dependency relations, can be enforced syntactically in a way that permits checking in linear time. For example, we can use guarded commands as with Moore modules, except that the guards and the right-hand sides of assignments are allowed to contain primed variables with the following proviso: if the guarded command for x' contains a primed variable y', then $x \succ_M y$. We refer to the dependency relation $\succ_M$ of a statically typed module $(M, \succ_M)$ as a *static type* for the module M. Note that if $\succ'$ is a dependency relation for M, and $\succ_M$ is a subset of $\succ'$, then $\succ'$ is also a static type for M.

Every non-blocking module has a static type (have each output variable depend on all input variables). Hence, there are composable modules with static types whose synchronous composition does not have a static type. However, static types suggest a sufficient condition for the existence of compound static types which can be checked efficiently. Two statically typed modules $(M, \succ_M)$ and $(N, \succ_N)$ are statically composable, or *static-composable*, if (1) the modules M and N are composable, and (2) the relation $\succ_{M||N} = \succ_M \cup \succ_N$ is acyclic. Then, the relation $\succ_{M||N}$ is a static type for the synchronous composition $M||N$. Since acyclicity can be checked in linear time, so can the requirement if two statically typed modules are *static*-composable. However, two statically typed modules $(M, \succ_M)$ and $(N, \succ_N)$ may not be *static*-composable even though the compound module $M||N$ is non-blocking.

Example 2. A module may have two static types, neither of which is a subset of the other. Let module M be such that $X^o_M = \{x_0, x_1\}$, $X^i_M = \{y\}$, and $\tau_M = (x'_0 \oplus x'_1 \oplus y'_0)$. Using guarded commands, we can specify M in two ways:

$$M' = \left\{ \begin{array}{lcl} [\!] \ \mathrm{T} & \rightarrow & x'_0 := \neg(x'_1 \oplus y') \\ & \wedge & \\ [\!] \ \mathrm{T} & \rightarrow & x'_1 := \mathrm{T} \\ [\!] \ \mathrm{T} & \rightarrow & x'_1 := \mathrm{F} \end{array} \right\} \qquad M'' = \left\{ \begin{array}{lcl} [\!] \ \mathrm{T} & \rightarrow & x'_0 := \mathrm{T} \\ [\!] \ \mathrm{T} & \rightarrow & x'_0 := \mathrm{F} \\ & \wedge & \\ [\!] \ \mathrm{T} & \rightarrow & x'_1 := \neg(x'_0 \oplus y') \end{array} \right\}$$

Note that both M' and M'' have the same transition predicate, namely τ_M, but they have different static types: the static type $\succ_{M'}$ for M' is $\{x_0 \succ x_1,\ x_0 \succ y\}$, while $\succ_{M''} = \{x_1 \succ x_0,\ x_1 \succ y\}$. Choosing different static types (i.e., implementations of the transition predicate) can have implications on composability with other modules. Let module N be such that $X^o_N = \{y\}$, $X^i_N = \{x_0, x_1\}$, and $\tau_N = (y' = x'_0)$ (or, using guarded commands, $[\!] \ \mathrm{T} \rightarrow y' := x'_0$). The static type $\succ_N$ for N is $\{y \succ x_0\}$. Then $(M', \succ_{M'})$ is not *static*-composable with $(N, \succ_N)$, but $(M'', \succ_{M''})$ is. □

Dynamically typed modules. Example 2 suggests the following generalization of static types. A *composite dependency relation* for a module M is a set $D = \{(\psi^1, \succ^1), \ldots, (\psi^m, \succ^m)\}$ of pairs, where each ψ^i is a predicate over the module

variables X_M, and each $\succ^i$ is a dependency relation for M, such that for each state s of M, there is exactly one predicate ψ^i, $1 \le i \le m$, with $\psi^i[\![s]\!]$. If $\psi^i[\![s]\!]$, then we write $\succ^s$ for the corresponding dependency relation $\succ^i$. A *dynamically typed module* (M, D_M) consists of a module M and a composite dependency relation $D_M = \{(\psi^i_M, \succ^i_M) \mid 1 \le i \le m\}$, such that (a) the module M is non-blocking, and (b) at every state s, the module M respects the dependency relation $\succ^s_M$. These two conditions, as well as the requirements on a composite dependency relation, can again be enforced syntactically in a way that permits checking in polynomial time. For example, each predicate ψ^i_M, $1 \le i < m$, can be required to contain the conjunct $\bigwedge_{j \ne i} \neg\psi^j_M$, and ψ^m_M can be required to be equal to $\bigwedge_{1 \le i < m} \neg\psi^i_M$. If we use guarded commands to specify the transition predicate, then for each guarded command, the guard can be required to contain a conjunct of the form ψ^i_M, for some $1 \le i \le m$, and together with the right-hand sides of assignments satisfy the proviso for the corresponding dependency relation $\succ^i_M$.

Example 3. Level-sensitive latches are commonly used in the design of high performance systems such as pipelined microprocessors. Typically different parts of a system are active depending on the phase of the clock. As an example, consider a circuit consisting of three modules M_1, M_2, and M_3. Module M_1 is an inverter that connects the output of the $\neg c$-clocked level-sensitive latch M_3 to the input of the c-clocked level-sensitive latch M_2. The output of the latch M_2 is connected to the input of the latch M_3. Using guarded commands, the three modules can be specified as follows:

$$M_1 = \{ [\![\ \mathrm{T} \ \to x' := \neg z' \} \quad M_2 = \left\{ \begin{array}{ll} [\![\ c & \to y' := x' \\ [\![\ \neg c & \to y' := y \end{array} \right\} \quad M_3 = \left\{ \begin{array}{ll} [\![\ c & \to z' := z \\ [\![\ \neg c & \to z' := y' \end{array} \right\}$$

The dynamic types for the modules are $D_{M_1} = \{(\mathrm{T}, x \succ z)\}$, $D_{M_2} = \{(c, y \succ x), (\neg c, \emptyset)\}$, and $D_{M_3} = \{(c, \emptyset), (\neg c, z \succ y)\}$. □

We refer to the composite dependency relation D_M of a dynamically typed module (M, D_M) as a *dynamic type* for the module M. Like static types, dynamic types suggest sufficient conditions for the non-blocking of synchronous composition. Furthermore, the conditions for the composability of dynamic types are more liberal than *static*-composability, and thus they are applicable in more situations. Consider two dynamically typed modules (M, D_M) and (N, D_N) with $D_M = \{(\psi^i_M, \succ^i_M) \mid 1 \le i \le m\}$ and $D_N = \{(\theta^j_N, \succ^j_N) \mid 1 \le j \le n\}$. We write $\succ^{i,j}$ for the union $\succ^i_M \cup \ \succ^j_N$, where $1 \le i \le m$ and $1 \le j \le n$. We provide two definitions of composability for dynamically typed modules, one purely syntactic, and the other in part semantic.

- The dynamically typed modules (M, D_M) and (N, D_N) are syntactically dynamically composable, or *dsynt-composable*, if (1) the modules M and N are composable, and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \le i \le m$ and $1 \le j \le n$. Then, $D_{M||N} = \{(\psi^i_M \wedge \theta^j_N, \succ^{i,j}) \mid 1 \le i \le m \text{ and } 1 \le j \le n\}$, is a dynamic type for the synchronous composition $M||N$.

– The dynamically typed modules (M, D_M) and (N, D_N) are semantically dynamically composable, or *dsem-composable*, if (1) the modules M and N are composable, and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$ for which the conjunction $\psi_M^i \wedge \theta_N^j$ is satisfiable. Then, $D_{M||N} = \{(\psi_M^i \wedge \theta_N^j, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \text{ and } (\exists X_{M||N})(\psi_M^i \wedge \theta_N^j)\}$ is a dynamic type for $M||N$.

Note that it can be checked in quadratic time whether two dynamically typed modules are *dsynt*-composable, while it requires exponential time (by evaluating a quadratic number of boolean Π_1^p formulas) to check if they are *dsem*-composable. However, checking if two dynamically typed modules are *dsem*-composable is still simpler than checking if the synchronous composition of two untyped modules is non-blocking (Π_1^p vs. Π_2^p).

Proposition 1. *(1) There are two dynamically typed modules (M, D_M) and (N, D_N) that are dsynt-composable but not static-composable, even though the union of all dependency relations in D_M is a static type for M, and the union of all dependency relations in D_N is a static type for N. (2) There are two dynamically typed modules that are dsem-composable but not dsynt-composable. (3) There are two dynamically typed modules (M, D_M) and (N, D_N) that are not dsem-composable, even though the synchronous composition $M||N$ is non-blocking.*

Example 4. The dynamically typed modules (M_1, D_{M_1}) and (M_2, D_{M_2}) of Example 3 are *dsynt*-composable, and the compound module $N = M_1 \| M_2$ has the dynamic type $D_N = \{(c, y \succ x \succ z), (\neg c, x \succ z)\}$. The modules (N, D_N) and (M_3, D_{M_3}) are not *dsynt*-composable, but they are *dsem*-composable. The compound module $N \| M_3$ has the dynamic type $\{(c, y \succ x \succ z), (\neg c, x \succ z \succ y)\}$. If the variable c in modules N and M_3 is replaced by its primed counterpart c' in both transition relations, then the dependency relation for N becomes $\{y \succ x \succ z,\ y \succ c\}$ for every state, and that for M_3 becomes $\{z \succ y,\ z \succ c\}$ for every state. Then (N, D_N) and (M_3, D_{M_3}) are not *dsem*-composable, even though the synchronous composition $N \| M_3$ is non-blocking. □

Summary. Let $\Lambda = \{async, moore, static, dsynt, dsem\}$ be the set of *module classes*. We summarize this section by defining, for each module class $\alpha \in \Lambda$, a set $\mathcal{M}_\alpha$ of modules: for $\alpha = async$, let $\mathcal{M}_{async}$ be the set of non-blocking modules; for $\alpha = moore$, let $\mathcal{M}_{moore}$ be the set of Moore modules; for $\alpha = static$, let $\mathcal{M}_{static}$ be the set of statically typed modules; and for $\alpha = dsynt$ and $\alpha = dsem$, let $\mathcal{M}_{dsynt} = \mathcal{M}_{dsem}$ be the set of dynamically typed modules. Define the *module class ordering* $async < moore < static < dsynt < dsem$. Then, for $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, every module $M \in \mathcal{M}_\alpha$ can be considered to be a module in $\mathcal{M}_\beta$ by adjusting its type or the semantics of composition, if necessary. Precisely: an *async*-module can be considered as a *moore*-module by changing the semantics of composition; a *moore*-module can be considered a *static*-module with the empty dependency relation; and a *static*-module can be considered a dynamically typed module with a single dependency relation. We also define for each module class

$\alpha \in \Lambda$ a corresponding composition operator $\|_\alpha$: if $\alpha = \mathit{async}$, then $\|_\alpha = |$; otherwise, $\|_\alpha = \|$.

3 Untyped and Typed Control Problems

Single-step vs. multi-step verification. Given a module M, a state s of M, and a predicate φ over the module variables X_M, the *single-step verification problem* (M, s, φ) asks whether for all states t, if $\tau[\![s, t']\!]$, then $\varphi[\![t]\!]$. The single-step verification problem amounts to evaluating the boolean Π_1^p formula $(\forall X'_M)$ $(\tau_M \rightarrow \varphi')[\![s]\!]$, where φ' results from φ by replacing all variables with their primed counterparts. A *run* r of a module M is a finite sequence $s_0 s_1 \ldots s_k$ of states of M such that $\tau_M[\![s_i, s'_{i+1}]\!]$ for all $0 \leq i < k$. The run r is *s-rooted*, for a state s of M, if $s_0 = s$. The run r *stays in* φ, for a predicate φ over the set X_M of module variables, if $\varphi[\![s_i]\!]$ for all $0 \leq i \leq k$. Given a module M, a state s of M, and a predicate φ over X_M, the *multi-step* (*invariant*) *verification problem* (M, s, φ) asks whether all s-rooted runs of M stay in φ. The multi-step verification problem can be solved by iterating the solution for the single-step verification problem. The number of states, which is exponential, gives a tight bound on the number of iterations.

Theorem 1. (cf. [AH98]) *The single-step verification problem is complete for coNP. The multi-step verification problem is complete for PSPACE.*

In control, it is natural to require that the controller falls into the same module class as the plant. Consider a module class $\alpha \in \Lambda$ and a module $M \in \mathcal{M}_\alpha$. The module $N \in \mathcal{M}_\alpha$ is an *α-controller* for M if (1) M and N are α-composable, and (2) $X^o_N = X^i_M$ and $X^i_N = X^o_M$. According to this definition, a controller for M is an environment of M that has no state on its own. For the control problems we consider in this paper, the results would remain unchanged if we were to consider controllers with state. As in verification, we distinguish between single-step and multi-step control. The single-step (resp. multi-step) control problem asks if there is a controller for a module that ensures that, starting from a given state, a given predicate holds after one step (resp. any number of steps). Precisely, for a module class α, a module $M \in \mathcal{M}_\alpha$, a state s of M, and a predicate φ over the set X_M of module variables, the *single-step* (resp. *multi-step*) *α-control problem* (M, s, φ) asks whether there is an α-controller N for M such that the answer to the single-step (resp. multi-step) verification problem $(M \|_\alpha N, s, \varphi)$ is Yes. If the answer is Yes, then the state s is single-step (resp. multi-step) *controllable* by N with respect to the *control objective* φ.

Fixed-type control. For $\alpha \in \{\mathit{static}, \mathit{dsynt}, \mathit{dsem}\}$, we also consider a variant of the control problems in which the type of the controller module is known (but its transiton relation is not). An instance (M, γ, s, φ) of the *fixed-type* single-step (resp. multi-step) α-control problem consists of an instance (M, s, φ) of the single-step (resp. multi-step) α-control problem together with a type γ for the controller. For $\alpha = \mathit{static}$, the type γ is a dependency relation for an α-controller for M; for $\alpha \in \{\mathit{dsynt}, \mathit{dsem}\}$, the type γ is a composite dependency

Class	Composability Check	Single-Step		Multi-Step	
		Arbitrary	Fixed	Arbitrary	Fixed
async	$\mathcal{O}(n)$	DP	—	EXP	—
moore	$\mathcal{O}(n)$	Σ_2^p	—	EXP	—
static	$\mathcal{O}(n)$	PSPACE	NE	EXP	NE
dsynt	$\mathcal{O}(n^2)$	PSPACE	NE	EXP	NE
dsem	coNP	PSPACE	NE	EXP	NE

(a) Complexity Results.

Class	MSG	MG
async	yes	yes
moore	yes	yes
static	no	no
dsynt	no	no
dsem	yes	no

(b) Existence of controllers.

Table 1. (a) Complexity of composability checking, as well as single-step and multi-step control for the various module classes. For statically and dynamically typed modules, we consider both arbitrary and fixed controller types. The quantity n is the size of the module description. Each problem is complete for the corresponding complexity class. (b) Existence of most state-general (MSG) and most general (MG) controllers.

relation for an α-controller for M. The instance (M, γ, s, φ) asks whether there is an α-controller N of type γ for M such that the answer to the single-step (resp. multi-step) verification problem $(M \parallel_\alpha N, s, \varphi)$ is Yes.

Generality of controllers. For a module class α, consider a module $M \in \mathcal{M}_\alpha$, a single-step (resp. multi-step) control objective φ, and two α-controllers N and N' for M. The controller N is *as state-general as* N' if all states s of M that are single-step (resp. multi-step) controllable by N' with respect to φ are also single-step (resp. multi-step) controllable by N with respect to φ. Moreover, if N and N' are equally state-general (i.e., N is as state-general as N', and vice versa), then N is *as choice-general as* N' if the transition predicate $\tau_{N'}$ implies τ_N (i.e., N permits as much nondeterminism as N'). An α-controller is *most state-general* if it is as state-general as any other α-controller. A α-controller is *most general* if (1) it is most state-general, and (2) it is as choice-general as any other most state-general α-controller.

Summary of results. In the following section, we present algorithms for solving the various types of control problems. The complexity results are summarized in Table 1(a). We recall that the complexity class DP consists of the languages that are intersections of an NP language and a coNP language. If n is the input size, the complexity class NE is $\bigcup_{k>0} \mathrm{NTIME}(2^{kn})$, and the complexity class EXP is $\bigcup_{k>0} \mathrm{DTIME}(2^{n^k})$. By the padding argument, any problem complete for NE is also complete for NEXP $= \bigcup_{k>0} \mathrm{NTIME}(2^{n^k})$ [Pap94]. Hence, assuming P $\neq$ NP, for the module classes *static*, *dsynt*, and *dsem*, the fixed-type multi-step control problems are harder than the multi-step control problems with arbitrary controller type. In addition, we summarize in Table 1(b) all results on the existence of most state-general and most general controllers.

4 Algorithms and Complexity of Control

We determine the complexity for solving the single-step and multi-step α-control problems for all five module classes $\alpha \in \Lambda$. In each case, the multi-step control problem can be solved by iterating an exponential number of times the solution for the corresponding single-step control problem.

Asynchronous control. Given a non-blocking module M, a state s of M, and a predicate φ over the module variables X_M, the single-step *async*-control problem amounts to evaluating the boolean formula

$$\big((\forall X_M'^o)(\tau_M \wedge (X_M'^i = X_M^i) \rightarrow \varphi') \wedge (\exists X_M'^i)(\tau_M \wedge (X_M'^o = X_M^o) \wedge \varphi')\big)[\![s]\!].$$

Hence, in the asynchronous case, the single-step control problem is complete for DP. It follows from [CKS81] that the multi-step version is complete for exponential time (cf. [HK97]).

Theorem 2. *The single-step async-control problem is complete for DP. The multi-step async-control problem is complete for EXP.*

Proposition 2. *For every non-blocking module and every control objective, there is a most general single-step async-controller, and there is a most general multi-step async-controller.*

Moore control. Given a Moore module M, a state s of M, and a predicate φ over X_M, the single-step *moore*-control problem amounts to evaluating the boolean Σ_2^p formula $(\exists X_M'^i)(\forall X_M'^o)(\tau_M \rightarrow \varphi')[\![s]\!]$. The multi-step hardness proof is similar to the asynchronous case.

Theorem 3. *The single-step moore-control problem is complete for Σ_2^p. The multi-step moore-control problem is complete for EXP.*

Proposition 3. *For every Moore module and every control objective, there is a most general single-step moore-controller, and there is a most general multi-step moore-controller.*

Statically typed control. Consider a statically typed module $(M, \succ_M)$, and let $X_M = \{x_1, \ldots, x_n\}$. A linear order $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ of the variables in X_M is *compatible* with the dependency relation $\succ_M$ if each output variable follows in the ordering the variables on which it depends. Precisely, $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ is compatible with $\succ_M$ if for all $1 \leq j, k \leq n$, if $x_{i_j} \succ x_{i_k}$, then $k < j$. Given a predicate φ over X_M, for each linear order $\ell = x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, we define the boolean formula $C(\ell, \varphi) = (\lambda_{i_1} x'_{i_1})(\lambda_{i_2} x'_{i_2}) \cdots (\lambda_{i_n} x'_{i_n})(\tau_M \rightarrow \varphi')$, where for $1 \leq k \leq n$, we have $\lambda_{i_k} = \forall$ if $x_{i_k} \in X_M^o$, and $\lambda_{i_k} = \exists$ if $x_{i_k} \in X_M^i$. The following lemma states that, in order to decide whether a state is single-step *static*-controllable, it suffices to consider all linear orders of variable dependencies.

Lemma 1. *Given a statically typed module $(M, \succ_M)$, a control objective φ over X_M, and a state s of M, the state s is single-step static-controllable with respect to φ iff there is a linear order ℓ of X_M compatible with $\succ_M$ such that $C(\ell, \varphi)[\![s]\!]$.*

Theorem 4. *The single-step static-control problem is complete for PSPACE. The multi-step static-control problem is complete for EXP.*

The single-step *static*-control problem is in PSPACE, because we can check each linear order in PSPACE. Hardness for PSPACE follows from the fact that, given a boolean formula $(\forall x_0)(\exists y_0)\cdots(\forall x_n)(\exists y_n)\varphi$, we can encode the problem of deciding its truth value as the *static*-control problem with the control objective φ for a module M with the variables $X^o_M = \{x_0, \ldots, x_n\}$ and $X^i_M = \{y_0, \ldots, y_n\}$, the valid transition relation τ_M, and the dependency relation $\succ_M = \{(x_i, y_j) \mid 1 \leq j < i \leq n\}$. The corresponding multi-step problem is again complete for EXP. Note that controllability by a Moore module corresponds to the special case in which every output variable depends on all input variables; that is, $\succ_M = X^o_M \times X^i_M$. The dual case is that on an empty dependency relation $\succ_M = \emptyset$. Here, the controller can choose the next values of the input variables dependent on the next values of all output variables, and the single-step control problem amounts to evaluating the boolean Π^p_2 formula $(\forall X'^o_M)(\exists X'^i_M)(\tau_M \wedge \varphi')[\![s]\!]$.

We consider now the case in which the type $\succ_N \subseteq X^i_M \times X_M$ of the controller is fixed. We assume that $\succ_M \cup \succ_N$ is acyclic; otherwise, M and N are not *static*-composable, and the answer to the fixed-type control problems is No. Let $X^o_M = \{x_1, \ldots, x_m\}$ and $X^i_M = \{y_1, \ldots, y_k\}$. Intuitively, for $1 \leq i \leq k$, the next value for y_i can be chosen in terms of the current values of the module variables, as well as in terms of the next values of the output variables on which y_i depends. Hence, a controller with fixed static type $\succ_N$ can be thought of as a set $\{f_1, \ldots, f_k\}$ of Skolem functions: for $1 \leq i \leq k$, the Skolem function f_i provides a next value for y_i, and has as arguments the variables in $X_M \cup \{x' \in X'^o_M \mid y_i \succ_N x\}$. This set of Skolem functions corresponds to the following boolean formula with Henkin quantifiers [Hen61,Wal70]:

$$H(\succ_N, \varphi) \;=\; \begin{pmatrix} (\forall \{x' \in X'^o_M \mid y_1 \succ_N x\})(\exists y'_1) \\ \cdots \\ (\forall \{x' \in X'^o_M \mid y_k \succ_N x\})(\exists y'_n) \end{pmatrix} (\tau_M \rightarrow \varphi').$$

The fixed-type single-step *static*-control problem can be solved as follows.

Lemma 2. *Given a statically typed module $(M, \succ_M)$, a static controller type $\succ_N \subseteq X^i_M \times X_M$ that is static-composable with $\succ_M$, a control objective φ over X_M, and a state s of M, the state s is single-step static-controllable with respect to φ by a controller with static type $\succ_N$ iff $H(\succ_N, \varphi)[\![s]\!]$.*

Deciding the truth value of a boolean formula with Henkin quantifiers is complete for NE, even if the formula has the restricted form shown above [GLV95].

Theorem 5. *The fixed-type single-step and multi-step static-control problems are complete for NE.*

Unlike Moore modules, a statically typed module may not have a most state-general controller.

Proposition 4. *There is a statically typed module and a control objective such that there is no most state-general single-step static-controller, nor a most state-general multi-step static-controller.*

Example 5. Let module M have the output variables $X^o_M = \{x_0, x_1, z\}$, the input variables $X^i_M = \{y_0, y_1\}$, the transition predicate $\tau_M = (z' = z)$, and the static type $\succ_M = \{x_0 \succ y_0,\ x_1 \succ y_1\}$. The control objective is $\varphi = (z \wedge (y_1 = x_0)) \vee (\neg z \wedge (y_0 = x_1))$. For every state s of M there is a controller N such that s is *static*-controllable by N with respect to φ: if $z[\![s]\!]$, then N has the transition predicate $\tau_N = (y'_1 = x'_0)$ and the static type $\succ_N = \{y_1 \succ x_0\}$; if $\neg z[\![s]\!]$, then $\tau_N = (y'_0 = x'_1)$ and $y_0 \succ_N x_1$. However, because of the acyclicity requirement for dependency relations, there is no single *static*-controller that controls all states of M. For the same reason, M also does not have a most state-general multi-step *static*-controller for the control objective φ. □

Dynamically typed control. The solution of control problems for dynamically typed modules closely parallels the solution for statically typed modules.

Lemma 3. *Given a dynamically typed module $(M, \{(\psi^i_M, \succ^i_M) \mid 1 \le i \le m\})$, a control objective φ over X_M, and a state s of M, the following assertions hold:*

- *The state s is single-step dsynt-controllable with respect to φ iff there is a linear order ℓ of X_M that is compatible with all $\succ^i_M$, for $1 \le i \le m$, such that $C(\ell, \varphi)[\![s]\!]$.*
- *The state s is single-step dsem-controllable with respect to φ iff there is a linear order ℓ of X_M that is compatible with $\succ^s_M$, such that $C(\ell, \varphi)[\![s]\!]$.*

Theorem 6. *For $\alpha \in \{dsynt, dsem\}$, the single-step α-control problem is complete for PSPACE, and the multi-step α-control problem is complete for EXP.*

Hence, the control problems for statically and dynamically typed modules have the same complexity. This applies also to the fixed-type control problems.

Lemma 4. *Given a dynamically typed module (M, D_M), a module class $\alpha \in \{dsynt, dsem\}$, a dynamic controller type $D_N = \{(\psi^i_N, \succ^i_N) \mid 1 \le i \le m\}$ that is α-composable with D_M, a control objective φ over X_M, and a state s of M, the state s is single-step α-controllable with respect to φ by a controller with dynamic type D_N iff $H(\succ^s_N, \varphi)[\![s]\!]$.*

Theorem 7. *For $\alpha \in \{dsynt, dsem\}$, the fixed-type single-step and multi-step α-control problems are complete for NE.*

Dynamically typed modules with syntactic composition do not necessarily have a most state-general controller. In contrast, dynamically typed modules with semantic composition always have a most state-general controller, but they may not have a most general one.

Proposition 5. *(1) There is a dynamically typed module and a control objective such that there is no most state-general single-step dsynt-controller, nor a most state-general multi-step dsynt-controller. (2) For every dynamically typed module and every control objective, there is a most state-general single-step dsem-controller, and there is a most state-general multi-step dsem-controller. (3) There is a dynamically typed module and a control objective such that there is no most general single-step dsem-controller, nor a most general multi-step dsynt-controller.*

Example 6. The module M of Example 5 can be viewed as a dynamically typed module whose dependency relation is the same for every state. The control objective is $\varphi = ((y_1 = x_0) \vee (y_0 = x_1))$. There exist at least two single-step *dsem*-controllers that control every state of M: the first controller N_1 has the transition predicate $\tau_{N_1} = (y'_1 = x'_0)$; the second controller N_2 has the transition predicate $\tau_{N_2} = (y'_0 = x'_1)$. However, there is no most general single-step *dsem*-controller. To control M with respect to φ in a most general way, a controller N with the transition predicate $\tau_N = ((y'_0 = x'_1) \vee (y'_1 = x'_0))$ would be required. Such a controller can be typed only if dynamic types are generalized to admit disjunctions of composite dependency relations. □

The relative power of controllers. Recall the module class ordering *async* < *moore* < *static* < *dsynt* < *dsem*. The following proposition establishes that this ordering strictly orders the power of controllers.

Proposition 6. *For all module classes $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, there is a module $M \in \mathcal{M}_\alpha$, a control objective φ over X_M, and a state s of M, such that s is not single-step (resp. multi-step) α-controllable with respect to φ, but s is single-step (resp. multi-step) β-controllable with respect to φ.*

Discussion. One may be inclined to define the following "unrestricted synchronous control problem": given a non-blocking module M, a state s of M, and a predicate φ over X_M, is there a module N composable with M such that (1) the synchronous composition $M \| N$ is non-blocking, and (2) the answer to the single-step (resp. multi-step) verification problem $(M \| N, s, \varphi)$ is Yes? This formulation does not distinguish between output and input variables, and thus permits the controller N to arbitrarily constrain the output variables of M, as long as the compound system $M \| N$ is non-blocking. Thus, the "unrestricted synchronous control problem" is not a control problem at all in the traditional sense, because it simply asks for the existence of a transition (in the single-step case) or run (in the multi-step case). The single-step solution amounts to evaluating the boolean Σ_1^p formula $(\exists X'_M)(\tau_M \wedge \varphi')[\![s]\!]$, and like invariant verification, the multi-step problem is complete for PSPACE. Note that if the non-blocking requirement (1) is also dropped, then the appropriate single-step formula is $(\exists X'_M)(\tau_M \rightarrow \varphi')[\![s]\!]$, which permits the controller to block the progress of M.

To introduce a semantic (i.e., non-type-based) distinction between output and input variables, one may define the condition of input universality, that states that a module should not constrain its inputs. Formally, a module M is

input universal if the Π_2^p formula $(\forall X_M)(\forall X_M'^{i})(\exists X_M'^{o})\tau_M$ is true. Input universality is a reasonable requirement, which, in particular, is satisfied by all typed modules. Input universality by itself, however, is not compositional: the composition of two input-universal modules is not necessarily input universal, and may be blocking (cf. Example 1). This is because input universality does not treat the module and its environment symmetrically: it states that a module, when composed with a Moore environment, will not block —but the module itself is not required to be Moore. This mismatch is reflected in the "input-universal control problem," that asks, given an input-universal module M, a state s of M, and a predicate φ over X_M, if there is an input-universal module N composable with M such that (1) the synchronous composition $M\|N$ is non-blocking, and (2) the answer to the single-step (resp. multi-step) verification problem $(M\|N, s, \varphi)$ is Yes. The single-step solution amounts to evaluating the boolean Π_2^p formula $((\forall X_M'^{o})(\exists X_M'^{i})(\tau_M \rightarrow \varphi') \wedge (\exists X_M'^{o})(\exists X_M'^{i})(\tau_M \wedge \varphi'))[\![s]\!]$. The two conjuncts of this formula imply different powers for the controller. The first conjunct states that the controller can look at the next values of the output variables, and propose new values for the input variables that either cause blocking, or achieve control (hence the controller is input universal). The second conjuct gives the controller the additional power of "guessing" the next values of the output variables in order to ensure non-blocking of the compound system. Types solve this mismatch by providing a stronger, compositional condition than input universality, which treats the module and the controller symmetrically.

References

AH98. R. Alur and T. A. Henzinger. *Computer-aided Verification: An Introduction to Model Building and Model Checking for Concurrent Systems.* Draft, 1998. 466

AH99. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999. 460, 462

AL95. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Programming Languages and Systems*, 17:507–534, 1995. 460

BG86. A. Blass and Y. Gurevich. Henkin quantifiers and complete problems. *Ann. Pure and Applied Logic*, 32:1–16, 1986. 461

BG88. G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. Technical Report 842, INRIA, 1988. 460

BL69. J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969. 459

CKS81. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28:114–133, 1981. 468

EJ91. E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus, and determinacy. In *Proc. Symp. on Foundations of Computer Science*, pp. 368–377. IEEE Press, 1991. 459

GH82. Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. Symp. Theory of Computing*, pp. 60–65. ACM Press, 1982. 459

GLV95. G. Gottlob, N. Leone, and H. Veith. Second-order logic and the weak exponential hierarchies. In *Mathematical Foundations of Compuer Science*, LNCS 969, pp. 66–81. Springer-Verlag, 1995. 461, 469

Hal93. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993. 460

Hen61. L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods*, pp. 167–183. Polish Scientific Publishers, 1961. 461, 469

HK97. T. A. Henzinger and P. W. Kopke. Discrete-time control for rectangular hybrid automata. In *Automata, Languages, and Programming*, LNCS 1256, pp. 582–593. Springer-Verlag, 1997. 468

Kur94. R. P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994. 460

Lyn96. N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996. 460

McM97. K. L. McMillan. A compositional rule for hardware design refinement. In *Computer-aided Verification*, LNCS 1254, pp. 24–35. Springer-Verlag, 1997. 460

McN93. R. McNaughton. Infinite games played on finite graphs. *Ann. Pure and Applied Logic*, 65:149–184, 1993. 459

Mil89. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 459

Pap94. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 467

RW87. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25:206–230, 1987. 459

Tho95. W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science*, LNCS 900, pp. 1–13. Springer-Verlag, 1995. 459

TW94. J. G. Thistle and W. M. Wonham. Control of infinite behavior of finite automata. *SIAM J. Control and Optimization*, 32:1075–1097, 1994. 459

Wal70. W. Walkoe. Finite partially-ordered quantification. *J. Symbolic Logic*, 35:535–555, 1970. 461, 469

Typing Non-uniform Concurrent Objects

António Ravara[1] and Vasco T. Vasconcelos[2]

[1] Department of Mathematics, Instituto Superior Técnico, Portugal
[2] Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal

Abstract. Concurrent objects may offer services non-uniformly, constraining the acceptance of messages on the states of objects. We advocate a looser view of communication errors. Safe programmes must guarantee that every message has a chance of being received if it requests a method that may become enabled at some point in the future. We formalise non-uniform concurrent objects in TyCO, a name-passing object calculus, and ensure program safety via a type system. Types are terms of a process algebra that describes dynamic aspects of the behaviour of objects.

1 Introduction

Herein we study non-uniform concurrent objects in TyCO [25,27], an asynchronous name-passing object calculus along the lines of the π-calculus [15,16]. Concurrent objects may offer services non-uniformly according to synchronisation constraints, that is, the availability of a service may depend on the state of the object [18]. Objects exhibiting methods that may be enabled or disabled, according to their internal state, are very common in object-oriented programming (think of a stack, a buffer, an FTP server, a bank account, a cash machine). Nevertheless, the typing of concurrent objects poses specific problems, due to the non-uniform availability of their methods.

The typed λ-calculus is a firm ground to study typing for sequential object-oriented languages, with a large body of research and results, namely on record-types for objects. However, a static notion of typing, like types-as-interfaces, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. Hence, we aim at a type discipline that copes with non-uniform concurrent objects. The interface of an object should describe only those methods that are enabled, and when a client asks for a disabled method the message should not be rejected if the object may evolve to a state where the method becomes available. To achieve this objective, we propose a looser view of communication errors, such that an object-message pair is not an error if the message may be accepted at some time in the future. Therefore, an error-free process guarantees that messages are given a chance of being attended, as a permanently enabled object eventually receives a message target to it. Errors are those processes containing an object that persistently refuses to accept a given message, either because the object is blocked, or because it will never have the right method for the message.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 474–489, 2000.

Traditional type systems assign rigid interface-like types to the names of objects [15,27]. Take the example of a one-place buffer that only allows read operations when it is full, and write operations when it is empty. We like to specify it as follows, showing that the buffer alternates between *write* and *read*.

$$\begin{array}{l}\textbf{def}\ \mathsf{Empty}(b) = b?\{\mathit{write}(u) = \mathsf{Full}[b,u]\}\\ \textbf{and}\ \mathsf{Full}(b,u) = b?\{\mathit{read}(r) = r!\mathit{val}[u] \mid \mathsf{Empty}[b]\}\end{array}$$

The referred type systems reject the example above, since name b alternates its interface. An alternative typable implementation uses the busy-waiting technique to handle non-available operations.

$$\begin{array}{l}\textbf{def}\ \ \mathsf{Buf}(b,v,\mathit{empty}) =\\ \qquad b?\{\mathit{write}(u) = \textbf{if}\ \mathit{empty}\ \textbf{then}\ \mathsf{Buf}[b,u,\mathit{false}]\\ \qquad\qquad\qquad\quad \textbf{else}\ b!\mathit{write}[u] \mid \mathsf{Buf}[b,v,\mathit{false}]\\ \qquad\qquad \mathit{read}(r) = \ \textbf{if}\ \mathit{empty}\ \textbf{then}\ b!\mathit{read}[r] \mid \mathsf{Buf}[b,v,\mathit{true}]\\ \qquad\qquad\qquad\quad \textbf{else}\ r!\mathit{val}[v] \mid \mathsf{Buf}[b,v,\mathit{true}]\}\end{array}$$

In the second implementation, a process containing the redex $\mathsf{Buf}[b,v,\mathit{empty}] \mid b!\mathit{read}[r]$ is not an error, and the presence of a message of the form $b!\mathit{write}[u]$ makes possible the acceptance of the *read* message. Similarly, in the first implementation, a process containing the redex $\mathsf{Empty}[b] \mid b!\mathit{read}[r]$ should not be considered an error, as again, the presence of a message like $b!\mathit{write}[u]$ also makes the reception of the *read* message possible. Nonetheless, notice that a deadlocked process like $\nu b\,(\mathsf{Empty}[b] \mid b!\mathit{read}[r])$ should be considered an error. In conclusion, the implementations behave similarly, in the sense that both accept a *read* message when the buffer is full, afterwards becoming empty, and accept a *write* message when the buffer is empty, afterwards becoming full. A more thorough discussion on non-uniform objects in TyCO, with more complex examples, can be found elsewhere [20].

We have been working on a theory of types able to accommodate this style of programming [21,22]. We adopt a types-as-behaviours approach, such that a type characterises the semantics of a concurrent objects by representing all its possible life-cycles as a state-transition system. Types are terms of a process algebra, fuelled by an higher-order labelled transition system, providing an internal view of the objects that inhabit them. It constitutes a synchronous view, since a transition corresponds to the reception of a message by an object. Hence, the types enjoy the rich algebraic theory of a process algebra, with an operational semantics defined via a labelled transition system, and a notion of equivalence. The equivalence is based on a bisimulation that is incomparable with other notions in the literature, and for which we define an axiomatic system, complete for image-finite types [21]. Therefore, types are partial specifications of the behaviour of objects, able to cope with non-uniform service availability. They are also suitable for specifying communication protocols, like pop3, since a type can represent sequences of requests to a server, and also deals with choice.

Equipped with a more flexible notion of error and with these richer types, we develop a type system which guarantees that typable processes will not run into communication errors.

2 The Calculus of Objects

TyCO (Typed Concurrent Objects) is a name-passing calculus featuring asynchronous communication between concurrent objects via labelled messages carrying names. The calculus is an object-based extension of the asynchronous π-calculus [5,11] where the objects behave according to the principles of the actor model of concurrent computation [1] (with the exception of the uniqueness of actors' names and the fairness assumption).

Syntax. Consider *names* a, b, v, x, y, and *labels* $l, m, n, \ldots$, possibly subscripted or primed, such that the set of names is countable and disjoint from the set of labels. Let $\tilde{v}$ stand for a sequence of names, and $\tilde{x}$ for a sequence of pairwise distinct names; moreover, $\boldsymbol{a}$ denotes a pair of sequences of names, the first identified by $\tilde{a}_i$, and the second by $\tilde{a}_o$, $\boldsymbol{a} = \tilde{a}_i; \tilde{a}_o$. Furthermore, assume a countable set of process variables, $X, Y, \ldots$, disjoint from the previous sets.

The grammar in Table 1 defines the set of processes. *Objects*, of the form $a?M$, and *messages*, of the form $a!l[\tilde{v}]$, are the basic processes in the calculus. An object is an input-guarded labelled sum where the name a is the *location* of the object, and M is a finite collection of labelled methods; each method $l(\tilde{x}) = P$ is labelled by a distinct label l, has a finite sequence of names $\tilde{x}$ as parameters, and has an arbitrary process P for body. An asynchronous message has a name a as target, and carries a *labelled value* $l[\tilde{v}]$ that selects the method l with arguments $\tilde{v}$. The location of an object or the target of a message is the *subject* of the process; arguments of messages are its *objects*. The process $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ denotes a *persistent object*, as it is done in the asynchronous π_1-calculus [2]. It is a recursive definition together with two sequences of arguments: input and output. The original formulation of TyCO used replication to provide for persistent objects. Despite its simplicity, replication is unwieldy. A recent formulation [25] uses 'process-declaration', of which the recursive definition herein is a particular case.

The remaining constructors of the calculus are fairly standard in name-passing process calculi: process $P \mid Q$ denotes the *parallel composition* of processes; process $\nu x\, P$ denotes the *scope restriction* of the name x to the process P (often seen as the creation of a new name, visible only within P); *inaction*, denoted $\mathbf{0}$, is the terminated process. The process $X[\boldsymbol{v}]$ is a recursive call.

In contrast with the actor model, and with most object-oriented languages, in TyCO there can be several objects sharing the same location and locations without associated objects. A *distributed object* is a parallel composition of several objects sharing the same location, possibly with different sets of methods, describing different copies of the same object, each in a different state. The type system ensures that *only the output capability of names may be transmitted*, e.g. in a method $l(\tilde{x}) = P$ the parameters $\tilde{x}$ are not allowed to be locations of objects in the body P. Such a restriction, henceforth called the *locality condition*, is present in object-oriented languages where the creation of a new name and of a new object are tightly coupled (e.g. Java), as well as in recent versions of the asynchronous π-calculus [2,4,13], and in the join-calculus [10]. This restriction is crucial; we would not know how to type non-uniform objects without it.

Notation. In $l(\tilde{x}) = P$, we omit the parentheses when $\tilde{x}$ is the empty sequence, writing $l = P$. Furthermore, we abbreviate to l a method $l = \mathbf{0}$ or a labelled value $l[]$, and abbreviate to $\nu\tilde{x}\,P$ a process $\nu x_1 \cdots \nu x_n\,P$. Also, we consider that the operator 'ν' extends as far to the right as possible. Finally, let $\{\tilde{x}\}$ denote the set of names of the sequence $\tilde{x}$, and $\{\boldsymbol{x}\}$ denote $\{\tilde{x}_i\} \cup \{\tilde{x}_o\}$. Henceforth, we assume the standard convention on names and process variables. Furthermore, we consider that process variables occur only bound in processes, and that in a recursive definition, $(\mathbf{rec}\,X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$, condition $\mathrm{fn}(a?M) \subseteq \{\boldsymbol{x}\}$ holds. Notice that $\{\tilde{x}_i\} \cap \{\tilde{x}_o\}$ is not necessarily empty.

Syntax:

$$P ::= a?M \mid a!l[\tilde{v}] \mid P \mid Q \mid \nu x\,P \mid (\mathbf{rec}\,X(\boldsymbol{x}).a?M)[\boldsymbol{v}] \mid X[\boldsymbol{v}] \mid \mathbf{0}$$

where $M ::= \{l_i(\tilde{x}_i) = P_i\}_{i \in I}$, and I is a non-empty finite indexing set.

Structural Congruence:

$$P \equiv Q, \text{ if } P \equiv_\alpha Q; \qquad a?M \equiv a?M', \text{ if } M \text{ is a permutation of } M';$$

$$P \mid \mathbf{0} \equiv P, \quad P \mid Q \equiv Q \mid P, \text{ and } (P \mid Q) \mid R \equiv P \mid (Q \mid R);$$

$$\nu x\,\mathbf{0} \equiv \mathbf{0}, \quad \nu xy\,P \equiv \nu yx\,P, \text{ and } \nu x\,P \mid Q \equiv \nu x\,(P \mid Q) \text{ if } x \notin \mathrm{fn}(Q).$$

Action Labels: $m ::= \tau \mid a?l[\tilde{v}] \mid \nu\tilde{x}\,a!l[\tilde{v}]$, where $\{\tilde{x}\} \subseteq \{\tilde{v}\} \setminus \{a\}$.

Message Application: $M \bullet l[\tilde{v}] \stackrel{\text{def}}{=} P[\tilde{v}/\tilde{x}]$, if $l(\tilde{x}) = P$ is a method in M.

Asynchronous Transition Relation:

$$\textsc{Out}\ a!l[\tilde{v}] \xrightarrow{a!l[\tilde{v}]} \mathbf{0} \qquad \textsc{In}\ a?M \xrightarrow{a?l[\tilde{v}]} M \bullet l[\tilde{v}] \qquad \textsc{Com}\ a?M \mid a!l[\tilde{v}] \xrightarrow{\tau} M \bullet l[\tilde{v}]$$

$$\textsc{Rin}\ \ (\mathbf{rec}\,X(x\boldsymbol{x}).x?M)[a\boldsymbol{v}] \xrightarrow{a?l[\tilde{v}]]} M[\mathbf{rec}\,X(x\boldsymbol{x}).x?M/X][a\boldsymbol{v}/x\boldsymbol{x}] \bullet l[\tilde{v}]$$

$$\textsc{Rec}\ \ (\mathbf{rec}\,X(x\boldsymbol{x}).x?M)[a\boldsymbol{v}] \mid a!l[\tilde{v}] \xrightarrow{\tau} M[\mathbf{rec}\,X(x\boldsymbol{x}).x?M/X][a\boldsymbol{v}/x\boldsymbol{x}] \bullet l[\tilde{v}]$$

$$\textsc{Par}\ \frac{P \xrightarrow{m} Q}{P \mid R \xrightarrow{m} Q \mid R}\ (\mathrm{bn}(m) \cap \mathrm{fn}(R) = \emptyset) \qquad \textsc{Open}\ \frac{P \xrightarrow{\nu\tilde{x}\,a!l[\tilde{v}]} Q}{\nu x\,P \xrightarrow{\nu x\tilde{x}\,a!l[\tilde{v}]} Q}\ (a \notin \{x\tilde{x}\})$$

$$\textsc{Res}\ \frac{P \xrightarrow{m} Q}{\nu x\,P \xrightarrow{m} \nu x\,Q}\ (x \notin \mathrm{fn}(m) \cup \mathrm{bn}(m)) \qquad \textsc{Struct}\ \frac{P \equiv P' \quad P' \xrightarrow{m} Q' \quad Q' \equiv Q}{P \xrightarrow{m} Q}$$

Table 1. Typed Concurrent Objects.

A Labelled Transition System. Following Milner *et al.* [16], we define the operational semantics of TyCO via two binary relations on processes, a static one — structural congruence — and a dynamic one — a labelled transition relation in an early style — as this formulation expresses more naturally the behavioural aspects of the calculus that we seek.

Definition 1 (Free and Bound Variables and Substitution).

1. *An occurrence of a process variable X in a process P is bound, if it occurs in the part M of a subterm $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ of P; otherwise the occurrence of X is free.*
2. *An occurrence of a name x in a process P is bound if it occurs in the subterm Q of the the part $l(\tilde{w}x\tilde{y}) = Q$, or in the subterm $\nu x\, Q$, or in the subterm $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ where $x \in \{\boldsymbol{x}\}$; otherwise the occurrence of x is free. Accordingly, we define the set $\mathrm{fn}(P)$ of the free names in P, the set $\mathrm{bn}(P)$ of the bound names in P, and the set $\mathrm{n}(P) \stackrel{\mathrm{def}}{=} \mathrm{fn}(P) \cup \mathrm{bn}(P)$ of the names in P.*
3. *Alpha-conversion, $\equiv_\alpha$, affects both bound names and bound process variables.*
4. *The process $P[\tilde{v}/\tilde{x}]$ denotes the simultaneous substitution in P of the names in $\tilde{v}$ for the free occurrences of the respective namess in $\tilde{x}$; it is defined only when $\tilde{x}$ and $\tilde{v}$ are of the same length. Similarly, the process $P[\boldsymbol{v}/\boldsymbol{x}]$ denotes the simultaneous substitution in P of the pair of sequences in $\boldsymbol{v}$ for the respective sequences in $\boldsymbol{x}$. Finally, the process $P[A/X]$ denotes the simultaneous substitution in P of the part A for the free occurrences of X.*

Table 1 defines the action labels. The *silent action* τ denotes internal communication in the process; the *input action* $a?l[\tilde{v}]$ represents the reception on the name a of an l-labelled message carrying names $\tilde{v}$ as arguments; the *output action* $\nu\tilde{x}\, a!l[\tilde{v}]$ represents the emission to a of an l-labelled message carrying names $\tilde{v}$ as arguments, some of them bound (those in $\tilde{x}$; the name a is free to allow the message to be received). In the last two action labels, the name a is the *subject* of the label, and the names $\tilde{v}$ are their *objects*.

Definition 2 (Free and Bound Names in Labels). *An occurrence of a name x in an action label m is bound, if it occurs in the part $a!l[\tilde{v}]$ of $\nu\tilde{y}x\tilde{z}\, a!l[\tilde{v}]$; otherwise the occurrence of x is free. Accordingly, we define the sets $\mathrm{fn}(m)$ and $\mathrm{bn}(m)$ of the free names and of the bound names in an action label m.*

We define the operational semantics of TyCO via an *asynchronous transition relation* that is inspired by the labelled relation defined Amadio *et al.* for the asynchronous π-calculus [3]. The asynchronous transition relation is the smallest relation on processes generated by the respective rules in Table 1, assuming the structural congruence relation inductively defined by the respective rules of Table 1. Notice that rules In, Rin, Com, and Rec assume that message application is defined. Otherwise, the transition does not take place.

3 Error-Free Processes

A communication error in TyCO is an object-message pair, such that message application is not defined. Two different reasons may cause the error: (1) the message requests a method that does not exist in the target object; (2) the message requests a method available in the object, but with a wrong number of arguments. To deal with non-uniform service availability of concurrent objects, this static notion of error is unsuitable. We propose a looser understanding of

what is a process with a communication error where the situations mentioned above are no longer considered errors, if the request may be accepted by the object at some time in the future (after changing its state).

The new notion of 'process without communication errors' needs the auxiliary notion of *redex*. A redex is a object-message pair sharing the same subject. If message application — the contractum of the redex — is defined, then the redex may reduce, and we call it a *good redex*; otherwise, it is a *bad* one. Since an object's location is not unique, there may be several redexes for the same message. To stress the identity of the object and that of the labelled value involved in the redex, we define $al\tilde{v}$-redexes. For the rest of this section, consider that m denotes only input actions. Let $\Longrightarrow$ denote $\xrightarrow{\tau}{}^{*}$, $\overset{m}{\Longrightarrow}$ denote $\Longrightarrow\xrightarrow{m}\Longrightarrow$, and $\overset{\widetilde{m}}{\Longrightarrow}$ denote sequences of $\overset{m}{\Longrightarrow}$. In the following definitions, when we refer to objects, we do not distinguish the ephemeral from the persistent.

Definition 3 (Redexes).

1. *The parallel composition of a distributed object $\Pi_{i\in I}\, a?M_i$ and a message $a!l[\tilde{v}]$ is an $al\tilde{v}$-redex. A process of the form $\nu\tilde{x}\,\Pi_{i\in I}\, a?M_i \mid a!l[\tilde{v}] \mid Q$ has an $al\tilde{v}$-redex, if there is no input action m with subject a such that $Q \overset{m}{\Longrightarrow}$.*
2. *A $al\tilde{v}$-redex is persistent in P if it is present in all the derivatives of P.*
3. *A non-empty $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, if*
   (a) *P has a subterm $x_0!l[\tilde{x}]$ in a method $l(\tilde{w}) = Q$ with $\{x_0, \tilde{x}\} \cap \{\tilde{w}\} = \emptyset$,*
   (b) *$P \overset{\widetilde{m}}{\Longrightarrow} \nu\tilde{z}\, Q[a\tilde{v}/x_0\tilde{x}] \mid Q'$, which has an $al\tilde{v}$-redex.*
4. *A sequence $\widetilde{m}$ generates an $al\tilde{v}$-redex in P, if*
   (a) *there is $x_0\tilde{x}$ such that $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, and*
   (b) *there is $\tilde{n} \neq \widetilde{m}$ such that $\tilde{n}$ makes a $bl\tilde{u}$-redex emerge from P by substituting $x_0\tilde{x}$, and $b\tilde{u} \neq a\tilde{v}$.*

An occasional bad redex is not enough to make the process an error, if further computation can consume (at least) one of the components of the redex. In short, errors are *persistent bad redexes*.

Definition 4 (Error-Free process). *The process P is error-free if, for all $al\tilde{v}$,*

$$\forall_{Q,\widetilde{m}}\, P \overset{\widetilde{m}}{\Longrightarrow} Q \text{ and } Q \text{ has a bad } al\tilde{v}\text{-redex not generated by } \widetilde{m} \text{ implies}$$

$$\exists_{R,\tilde{n}}\, Q \overset{\tilde{n}}{\Longrightarrow} R \text{ and } R \text{ does not have a bad } al\tilde{v}\text{-redex.}$$

The condition on Q ensures that the message participating in a bad $al\tilde{v}$-redex neither comes from, nor is generated by, the environment, since an environment can always interact incorrectly to produce errors. Otherwise, any process with free input-names would be an error.

In conclusion, communication errors are processes containing a message that can never be accepted by a persistent object. We can distinguish two cases: (1) *message-never-understood*, when the object does not have the method requested by the message, and it will never have it; (2) *blocking*, when the object has the method requested by the message, but not in its present interface, and it can never reach a state where it could accept the message.

Example 1. Consider the one-place buffer defined in the introduction[1].

1. Process $Empty[b] \mid b!think$ is an error, since the object b will never have the method *think*. Processes $R \stackrel{\text{def}}{=} Empty[b] \mid b!read[r]$ and $S \stackrel{\text{def}}{=} Empty[b] \mid b!write[u] \mid b!write[v]$ are not errors, since the bad redexes can disappear, if the environment provides the right messages. Processes $\nu b\, R$ and $\nu b\, S$ are erroneous, since in both processes the object b is blocked (as the scope of the object's name is restricted), hence the bad redexes become persistent (cf. an example by Boudol [6]). However, $\nu b\, R \mid a!l[b]$ is not an error, since the name b can be extruded, so the bad redex is not necessarily persistent.
2. Process $(\mathbf{rec}\, X(x;y).x?\{l_1 = y!l_1 \mid x?\{l_2 = y!l_2 \mid X[x;y]\}\})[a;a] \mid a!l_1 \mid a!l_2$ is not an error, although there is always a bad redex, but with messages containing different labels. The bad redex is recurring, but not persistent. Process $(\mathbf{rec}\, X(xy;xy).x?\{l = X[xy;xy] \mid x!l \mid y!m \mid y!n \mid y?n\})[ab;ab]$ is an error, even though the object and the messages participating in the bad *am*-redex are always different.
3. We do not want to reject processes that compute erroneously due only to the incorrect behaviour of the environment.
   (a) Process $Q \stackrel{\text{def}}{=} Empty[b] \mid c?\{l(x) = x!think\}$ is not an error, although there are interactions with it leading to errors (take $Q \xrightarrow{c?l[b]} Empty[b] \mid b!think$).
   (b) Process $(\mathbf{rec}\, X(x;).x?\{l = X[x;]\})[a;] \mid b?\{l(x) = \nu v\, x!l[v] \mid v?\{l(x) = x!l\} \mid c?\{l(x) = x!l[a]\}$ is not an error, although it has derivatives which are errors, depending on the requests coming from the environment.

It is easy to conclude that this notion is undecidable. However, any notion of run-time error of a (Turing complete) language is undecidable, since it can be reduced to the halting problem [26]. A common solution, which should be proved correct, but obviously cannot be complete, is the use of a type system to ensure that well-typed processes do not attain run-time errors.

4 The Algebra of Behavioural Types

We propose a process algebra, the Algebra of Behavioural Types, ABT, where terms denote types that describe dynamic aspects of the behaviour of objects. A type denotes a higher-order labelled transition system where states represent the possible interfaces of an object; state transitions model the dynamic changes of the interfaces by executing a method, thus capturing some causality information. The syntax and operational semantics of ABT are similar to a fragment of CCS [14], the class of Basic Parallel Processes (BPP) of Christensen [7] where communication is not present (parallel composition is merge). However, we need to give a different interpretation to some features of the process algebra, and thus decided to develop ABT. A thorough discussion is presented elsewhere [21].

[1] The translation of $Empty[b]$ into the syntax of this section is the following: $((\mathbf{rec}\, X(x;).x?\{write(u) = x?\{read(r) = r!val[u] \mid X[x;]\}\})[b;]$.

Syntax. Assume a countable set of *method names*, $l, m, n, \ldots$, possibly subscripted, and a countable set of *type variables*, denoted by t. Consider the sets disjoint.

The grammar in Table 2 defines the set of behavioural types. A term of the form $l(\widetilde{\alpha}).\alpha$ is a *method type*. The label l in the prefix stands for the name of a method possessing parameters of type $\widetilde{\alpha}$; the type α under the prefix prescribes the behaviour of the object after the execution of the method l with parameters of type $\widetilde{\alpha}$. A term of the form $\upsilon.\alpha$ is a *blocked type*, the type of an unavailable method. The sum '$\sum$' is a non-deterministic type composition operator that: (1) gathers together several method types to form the type of an object that offers the corresponding collection of methods — the *labelled sum* $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$; (2) associates several blocked types in the *blocked sum* $\sum_{i\in I}\upsilon.\alpha_i$; after being released, the object behaves according to one of the types α_i. The *parallel composition* ('$\|$') is the type of the concurrent composition of several objects located at the same name (interpreted as different copies of the same object, each in a different state), or the type of the concurrent composition of several messages targeted to the same name. Finally, the term $\mu t.\alpha$ (for $\alpha \neq t$) denotes a recursive type, enabling us to characterise the behaviour of persistent objects. Assume the variable convention and that types are equal up to α-conversion.

Definition 5 (Free and Bound Variables). *An occurrence of the variable t in a part α of the type $\mu t.\alpha$ is bound; the occurrence of t in the type α is free. The type $\alpha[\beta/t]$ denotes the substitution in α of β for the free occurrences of t.*

Syntax:

$$\alpha ::= \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \;\mid\; \sum_{i\in I} \upsilon.\alpha_i \;\mid\; \alpha \parallel \alpha \;\mid\; t \;\mid\; \mu t.\alpha$$

where I is a finite indexing set, and each $\widetilde{\alpha}_i$ is a finite sequence of types.

Action Labels: $\pi ::= \upsilon \;\mid\; l(\widetilde{\alpha})$.

Labelled transition relation:

$$\textsc{Act}\quad \sum_{i\in I} \pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I)$$

$$\textsc{Rpar}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \textsc{Lpar}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'} \qquad \textsc{Rec}\quad \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

Table 2. The Algebra of Behavioural Types.

Operational Semantics. A higher-order labelled transition system defines the operational semantics of ABT. Label υ denotes a silent transition that releases a blocked object; label $l(\widetilde{\alpha})$ denotes the invocation of method l with arguments of types $\widetilde{\alpha}$. The rules of Table 2 inductively define the labelled transition relation.

Notation. Let $\mathbf{0}$ denote the sum with the empty indexing set; we omit the sum symbol if the indexing set is singular, and we use the plus operator ('+') to denote binary sums of types. The type $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$. The *interface* of a type α, $\mathrm{int}(\alpha)$, is the set of its observable labels: in a labelled sum it is the set $\{l_i(\widetilde{\alpha}_i)\}_{i\in I}$, in a blocked type it is empty, and in a parallel composition it is the union of the interfaces of the components. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^{*}$, $\overset{\pi}{\Longrightarrow}$ denote $\Longrightarrow\xrightarrow{\pi}\Longrightarrow$, and $\overset{\tilde{\pi}}{\Longrightarrow}$ denote sequences of $\overset{\pi}{\Longrightarrow}$.

Type Equivalence. Label-strong bisimulation, *lsb*, is a *higher-order strong bisimulation on labels* and a *weak bisimulation on unblockings*. We require that if α and β are *label-strong bisimilar*, then: (1) if α offers a method, also β offers that method, and the parameters of the methods are pairwise bisimilar; and (2) if α offers an unblocking transition, β offers zero or more unblocking transitions.

Definition 6 (Bisimilarity on Types).

1. *A symmetric relation $R \subseteq T \times T$ is a bisimulation, if whenever $\alpha R \beta$,*
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists_{\beta',\widetilde{\beta}}\, \beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \widetilde{\alpha} R \beta' \widetilde{\beta}$; [2]
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists_{\beta'}\, \beta \Longrightarrow \beta'$ and $\alpha' R \beta'$.
2. *Two types α and β are label-strong bisimilar, or simply bisimilar, $\alpha \approx \beta$, if there is a label-strong bisimulation R such that $\alpha R \beta$.*

The usual properties of bisimilarities hold, namely $\approx$ is an equivalence relation and a fixed point. The sum of method types and the sum of blocked type preserve bisimilarity, as they are guarded sums. Furthermore, parallel composition and recursion also preserve bisimilarity. Briefly, *lsb* is a higher-order congruence relation. However, it is not known if the relation is decidable.

Type equivalence is a symmetric simulation; the simulation, $\leq$, is a partial order (reflexive, transitive, and anti-symmetric — any two types which simulate each other are equivalent). Thus, if α simulates α', then we say that α is a *subtype* of α'. Moreover, the operators of ABT, as well as *lsb*, preserve *subtyping*.

5 Type Assignment

To ensure that a process is error-free (Definition 4), the interactions within the process should be disciplined in such a way that persistent bad redexes do not appear. The types specify the interactions that can happen on a name and the types of the names carried by it. Thus, a type system assigning types to the names of a process imposes the discipline.

Typings. We need to distinguish the usage of names within a process — either as locations of objects or as targets of messages — since it is their simultaneous presence that may result in communication errors. Therefore, the type system assigns pairs of types to names and pairs of sequences of types to process variables.

[2] We write $\alpha_1 \ldots \alpha_n R \beta_1 \ldots \beta_n$ to denote $\alpha_1 R \beta_1, \ldots,$ and $\alpha_n R \beta_n$.

Definition 7 (Typing Judgements). *Consider* $\gamma \stackrel{\text{def}}{=} (\alpha, \beta)$ *and* $\boldsymbol{\gamma} \stackrel{\text{def}}{=} (\widetilde{\alpha}, \widetilde{\beta})$.

1. *Type assignment to names are formulae* $a{:}\gamma$.
2. *Type assignment to process variables are formulae* $X{:}\boldsymbol{\gamma}$.
3. *Typings,* Γ, Δ, *are finite sets of type assignments — to names or to process variables — where no name and no process variable occurs twice.*
4. *Judgements are formulae* $\Gamma \vdash P$. *Process* P *is typable if such a formula holds.*

Notation. Let $\mathrm{dom}(\Gamma)$ be the set of the names and the process variables in Γ. Then, $\Gamma \cdot a{:}\gamma$ denotes $\Gamma \cup \{a{:}\gamma\}$ if $a \notin \mathrm{dom}(\Gamma)$. Let $\Gamma(a) = \gamma$ if $a{:}\gamma \in \Gamma$, and $\Gamma(a) = (\mathbf{0}, \mathbf{0})$ otherwise; furthermore, $\Gamma(a)_{\mathsf{i}} = \alpha$ and $\Gamma(a)_{\mathsf{o}} = \beta$. Take bisimulation and subtyping on γ-types pairwise defined; hence, $\gamma \leq \gamma'$ if $\gamma_{\mathsf{i}} \leq \gamma'_{\mathsf{i}}$ and $\gamma_{\mathsf{o}} \leq \gamma'_{\mathsf{o}}$, and $\Gamma' \approx \Gamma$ if, for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, $\Gamma(a)_{\mathsf{i}} \approx \Gamma'(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}} \approx \Gamma'(a)_{\mathsf{o}}$.

Capturing the Notion of Communication Error. We refer to two syntactic categories of types, input and output, describing names as object locations and as message targets. An input-type characterises the sequences of messages that an object can accept, and thus the object life-cycle; an output-type characterises the messages sent to an object. We define co-inductively two new higher-order contravariant relations on types, the *agreement* relation and the *compatibility* relation. The former ensures the absence of blockings, the latter the absence of messages-never-understood. If a type denotes a graph, or a rational tree (cf. [22]), an output-type agrees with an input-type if either they share one path, or a path of one of them is a prefix of a path of the other, and, moreover, all paths which have a common prefix are equal (up-to agreement on the parameters, contravariantly). Unblocking occurs only when at least one of the types is blocked, or when both types have the same interface. Furthermore, if one of the types is equivalent to an empty type, then it agrees with all types (i.e. a message or an object, by themselves, are not errors). The compatibility relation is a relaxing of the agreement relation by allowing the environment to "help", providing some transitions. Hence, the compatibility relation considers *projections* of input-types.

Definition 8 (Agreement on Types). *Fix* Γ *and* P *such that* $\Gamma \vdash P$.

1. *A relation* $S \subseteq T \times T$ *is an agreement on* Γ *and* P, *if whenever* $\alpha \; S \; \beta$,
   (a) $\beta \approx \mathbf{0}$ *or* $\alpha \approx \mathbf{0}$, *or else*
   (b) *i.* $\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha', \beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$, *and for all* $a!l[\tilde{v}]$ *subterms of* P *such that* $\Gamma(a) \approx (\alpha\beta)$ *and* $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, *we have* $\widetilde{\alpha}_2 \; S \; \widetilde{\beta}_1$ *and* $\alpha' S \beta'$,
   ii. $\forall_{l,\alpha',\beta',\widetilde{\beta}_1\widetilde{\beta}_2} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha', \; \beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$, *and for all* $a!l[\tilde{v}]$ *subterms of* P *such that* $\Gamma(a) \approx (\alpha\beta)$ *and* $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, *we have* $\widetilde{\alpha}_2 \, S \, \widetilde{\beta}_1$ *implies* $\alpha' S \beta'$,
   iii. if $\mathrm{int}(\alpha) = \emptyset$ *or* $\mathrm{int}(\beta) = \emptyset$ *or* $\mathrm{int}(\alpha) = \mathrm{int}(\beta)$, *then*
   $\alpha \xrightarrow{v} \alpha'$ *implies* $\exists_{\beta'} \beta \Longrightarrow \beta'$ *and* $\alpha' \; S \; \beta'$, *and*
   $\beta \xrightarrow{v} \beta'$ *implies* $\exists_{\alpha'} \alpha \Longrightarrow \alpha'$ *and* $\alpha' \; S \; \beta'$.
2. *A type* α *agrees with a type* β *on a typing* Γ *and a process* P, *written* $\alpha \bowtie^{P}_{\Gamma} \beta$, *if there is an agreement relation* S *on* Γ *and* P *such that* $\alpha \; S \; \beta$.

The relation $\bowtie_\Gamma^P$ is the largest type agreement relation. We define similarly the type compatibility relation $\prec_\Gamma^P$, substituting '$\alpha \equiv \mathbf{0}$' with 'α finite' in condition 1(a), and '$\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$' with '$\alpha \xrightarrow{\widetilde{\pi}} \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$ without occurrences of silent transitions', in condition 1(b).

Type System. The rules in Table 3 inductively define the typing checking system. There is a type rule for each process constructor, and an extra rule ($\approx$) that allows the substitution of a type with a bisimilar one in a formula. Rules VAR and REC check that the types of the arguments of the process variable are equivalent to the types of its parameters; rule MSG checks that the output-type of the message's target has a transition labelled with the message's label (with the correct type parameters); rule OBJ checks that all the parameters of the methods are only used for output, and that the input-type of the object has a (reachable) state which has all its transitions labelled exactly with those of the object's interface. Notice that RES do not discard the bound variable from the typing. This is because we may need its type-pair to verify an agreement.

NIL	$\Gamma \vdash \mathbf{0}$	VAR	$\Gamma \cdot X{:}\boldsymbol{\gamma} \vdash X[\boldsymbol{v}]$ $(\Gamma(\boldsymbol{v}) \approx \boldsymbol{\gamma})$
MSG	$\Gamma \cdot \tilde{v}{:}(\_, \widetilde{\beta}) \vdash a!l[\tilde{v}]$		$(\Gamma(a)_\circ \Longrightarrow \xrightarrow{l(\widetilde{\beta}')}$ and $\widetilde{\beta} \approx \widetilde{\beta}')$
OBJ	$\dfrac{\Gamma \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta}_i) \vdash P_i}{\Gamma \vdash a?\{l_i(\tilde{x}_i) = P_i\}_{i\in I}}$		$(\exists_{\widetilde{\pi},\alpha}\, \Gamma(a)_{\mathrm{i}} \xRightarrow{\widetilde{\pi}} \xrightarrow{l_i(\widetilde{\beta}_i')}, \forall_{i\in I}$ with $\widetilde{\beta}_i \approx \widetilde{\beta}_i')$
REC	$\dfrac{\Gamma \cdot X{:}\boldsymbol{\gamma} \cdot \boldsymbol{x}{:}\boldsymbol{\gamma} \vdash a?M}{\Gamma \cdot \boldsymbol{v}{:}\boldsymbol{\gamma}' \vdash (\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]}$		$(\boldsymbol{\gamma}' \leq \boldsymbol{\gamma}$ and $\forall_{\gamma\in\{\tilde{\gamma}_\circ\}} \gamma \approx \mu t. \sum_{i\in I} v.\beta_i)$

RES $\dfrac{\Gamma \vdash P}{\Gamma \vdash \nu x\, P}$ PAR $\dfrac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}$ $\approx$ $\dfrac{\Gamma \cdot x{:}\gamma \vdash P}{B, \Gamma \cdot x{:}\delta \vdash P}$ $(\gamma \approx \delta)$

Table 3. Typing Checking System.

Example 2 (Typing Objects).

1. One-place boolean buffer: $\{b{:}(\mu t.write(bool).read(val).t, \mathbf{0})\} \vdash Empty[b]$.
2. Internal non-determinism: consider an object with internal non-determinism, $a?\{l = \nu c\, c!m_1 \mid c!m_2 \mid c?\{m_1 = a?l_1, m_2 = a?l_2\}\}$. The input-type of a expresses the choice: $\Gamma(a)_{\mathrm{i}} = l.(v.l_1 + v.l_2)$.
3. Persistent objects (cf. [8]): consider the following three objects:
   (a) $P_1 \stackrel{\text{def}}{=} (\mathbf{rec}\, X(a;).a?\{rep = X[a;] \mid a?\{mess\}\})[a;]$,
   (b) $P_2 \stackrel{\text{def}}{=} (\mathbf{rec}\, X(b;).b?\{quest(x) = X[b;] \mid x!rep\})[b;]$, and
   (c) $P_3 \stackrel{\text{def}}{=} (\mathbf{rec}\, X(c;).c?\{rep = X[c;], mess = X[c;]\})[c;]$.
   Process $P_1 \mid P_2 \mid P_3 \mid b!quest[a] \mid b!quest[c]$ is error-free. It is easy to check that:

   (a) $\Gamma(a)_{\mathsf{i}} = \mu t.rep.(t \| mess) \bowtie^P_\Gamma rep = \Gamma(a)_{\mathsf{o}}$,
   (b) $\Gamma(b)_{\mathsf{i}} = \mu t.quest(rep).t \bowtie^P_\Gamma quest(rep) \| quest(rep) = \Gamma(b)_{\mathsf{o}}$, and
   (c) $\Gamma(c)_{\mathsf{i}} = \mu t.rep.t + mess.t \bowtie^P_\Gamma rep = \Gamma(c)_{\mathsf{o}}$.
4. Blocked objects:
   (a) Let $\Gamma \stackrel{\text{def}}{=} \{b{:}(\mu t.write(bool).read(val).t, read(val))\}$ $\vdash \nu b\, Empty[b] \mid b!read[r]$. The process is an error, and $\Gamma(b)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(b)_{\mathsf{o}}$.
   (b) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(m, \upsilon.n), b{:}(l, \mathbf{0})\} \vdash \nu ab\, b?\{l = a!n\} \mid a?\{m\}$. The process is not an error, but $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.
   (c) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(n.m, \upsilon.n \parallel m), b{:}(l, l)\} \vdash \nu a\, b?\{l = a!n\} \mid a?\{n\} \mid a!m \mid b!l$. The process is neither an error nor a deadlock, still $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.

The type system accepts several type-pairs for each name in a process; not only does it accept bisimilar types, but it also accepts types which are in the subtyping relation. Consider $a?m|a?l$; possible input-types of a are $m \| l$, $l \| m$, $m.l + l.m$, and $l.m + m.l$, all bisimilar. Consider now $a?\{l = a?m\} \mid a!m$; a minimal typing of P is $\{a{:}(l.m, m)\}$, but there are subtypes of $\Gamma(a)$ which also succeed in type-checking with P (like $\{a{:}(l.m, l \parallel m)\}$, even though there is no message l with subject a). Moreover, an error process can type-check with a typing "too generous" for it: $\{a{:}(l.m, l \parallel m)\} \vdash \nu a\, P$, since $\Gamma(a)_{\mathsf{i}} \prec^{\nu a\, P}_\Gamma \Gamma(a)_{\mathsf{o}}$; nevertheless, $\nu a\, P$ is an error.

Definition 9 (Minimal Typing). *Let $\Delta \leq \Gamma$, if $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma)$ and $\Delta(a) \leq \Gamma(a)$, for all $a \in \mathrm{dom}(\Delta)$. We say that Γ is a minimal typing for P, if $\Gamma \vdash P$ and $\forall_\Delta\, \Delta \vdash P$ implies $\Delta \leq \approx\Gamma$.*

Proposition 1 (Basic Properties of Minimal Typings). *A typable process has a minimal typing which is unique up-to type equivalence.*

Proposition 2 (Recursive Output Types). *Take a minimal Γ for P. If, for some name a, $\Gamma(a)_{\mathsf{o}}$ has a subterm of the form $\mu t.\beta$, then β is a blocked sum.*

To be an error is a property of processes which cannot be modularly checked. Thus, we cannot introduce side conditions in the rules to check that input types are compatible or agres with output types. Therefore, after type checking a process, we compute its minimal typing, and check whether the types of the free names are compatible, and whether the types of the bound names agree.

Definition 10 (Well-Typed Processes). *A process P is well-typed, if it is typable and its minimal typing Γ satisfies the following conditions.*

1. *for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{fn}(P)$, $\Gamma(a)_{\mathsf{i}} \prec^P_\Gamma \Gamma(a)_{\mathsf{o}}$;*
2. *for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{bn}(P)$, $\Gamma(a)_{\mathsf{i}} \bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.*

Theorem 1 (Operational Correspondence). *Let $\Gamma \vdash P \xrightarrow{m} Q$ with $\Delta \vdash Q$.*

1. *$m \equiv \nu\tilde{x}\, a!l[\tilde{v}]$, if and only if, $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta})} \approx \Delta(a)_{\mathsf{o}}$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;*
2. *$m \equiv a?l[\tilde{v}]$, if and only if, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \approx \Delta(a)_{\mathsf{i}}$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;*
3. *$m \equiv \tau$, if and only if, $\exists a \in \mathrm{n}(P)$, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta}_1)} \Delta(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta}_2)} \Delta(a)_{\mathsf{o}}$.*

The following theorem characterises the conditions under which the interactions within a process, and between a process and the environment, preserve the typability of the process. Theorem 1 allows to construct the typings for Q. Since the calculus is local,no context may create objects located at extruded names.

Theorem 2 (Subject Reduction). *Consider Γ a well-typing for process P.*

1. *If $P \xrightarrow{\tau} Q$, then Q is well-typed.*
2. *If $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]} Q$, then Q is well-typed.*
3. *If $P \xrightarrow{a?l[\tilde{v}]} Q$ and $\alpha\Gamma(\tilde{v})_i \prec^{Q}_{\Gamma} \Gamma(a)_o\widetilde{\beta}$ with $\Gamma(a)_i \xrightarrow{l(\widetilde{\beta})} \alpha$ then Q is well-typed.*

Corollary 1. *If P is well-typed then P is error-free.*

Notice that the converse of this result is not true. The agreement and compatibility relations do not have causality information, thus there are deadlocks which are not errors and are detected (cf. Example 2.4(b)), but there are interesting processes which are rejected (cf. Example 2.4(c)). The proofs of the above results can be found in the full paper [23].

6 Related Work

In the context of the lazy λ-calculus, Dami proposed a liberal approach to potential errors [9], which is similar to our notion of communication-error. He argues that the common notion of erroneous term is over-restrictive, as some programs, in spite of having error terms in them, do not actually attain a run-time error when executed. Since there is a family of programming languages based on the lazy λ-calculus, Dami proposes a lazy approach to errors: a term is considered erroneous if and only if it always generates an error after a finite number of interactions with its context. Nierstrasz on his work 'Regular types for active objects' [18] discusses on non-uniform service availability of concurrent objects, and proposes notions of behavioural typing and subtyping for concurrent object-oriented programming, notions which take into account dynamic aspects of objects' behaviour. Puntigam [19] starts from Nierstrasz' work, defines a calculus of concurrent objects, a process-algebra of types (with the expressiveness of a non-regular language), and a type system which guarantees that all messages that are sent to an object are accepted, this purpose being achieved by enforcing sequencing of messages. Subtyping is a central issue in his work. It seems quite natural to have a liberal approach to potential errors in non-uniform concurrent objects. It allows a more flexible and behaviour-oriented style of programming and, moreover, detects some deadlocks. In the context of the π-calculus, there is some work on deadlock detection using types. Kobayashi proposes a type system to ensure partial deadlock freedom and partial confluence of programs [12,24] where types have information about the ordering of the use of channels, and also about the reliability of the channels used (with respect to non-determinism). Yoshida defines graph types for monadic π-processes where the nodes denote basic actions and the edges denote the ordering between the

actions [28]. However, TyCO is not π, and making comparisons with π is not a simple task. Although our types are easily adaptable to π — by associating a fix label, e.g. *val*, to each channel name — the definition of error used herein is difficult to express, if indeed possible. On non-uniform types for mobile processes, we know the works by Boudol, by Colaço, Pantel, and Sallé, and by Najm and Nimour. Boudol proposes a dynamic type system for the Blue Calculus, a variant of the π-calculus directly incorporating the λ-calculus [6]. The types are functional, in the style of Curry simple types, and incorporate Hennessy-Milner logic with recursion — modalities interpreted as resources of names. Processes inhabit the types, and this approach captures some causality in the usage of names in a process, ensuring that messages to a name will meet a corresponding offer. Colaço *et al.* [8] propose a calculus of actors and a type system which aims at the detection of "orphan messages", i.e. messages that may never be accepted, either because the requested service is not in the actor's interface, or due to dynamic changes in a actor's interface. Types are interface-like with multiplicities, and the type system requires complex operations on a lattice of types. Najm and Nimour [17] propose several versions of a calculus of objects that features dynamically changing interfaces and distinguishes between private and public objects' interfaces. For each version of the calculus they develop a typing system handling dynamic method offers in private interfaces, and guaranteeing some liveness properties. Types are sets of deterministic guarded equations, equipped with a transition relation that induces an equivalence relation, and a subtyping relation, both based on bissimulation.

Acknowledgements

Special thanks are due to Gérard Boudol, Jean-Louis Colaço, Silvano Dal-Zilio, and Uwe Nestmann for careful reading of previous versions of this work, as well as for very important suggestions and fruitful discussions. We also thank the anonymous referees for their comments and suggestions. The Danish institute BRICS, the ENSEEIHT at Toulouse, and the Franco-Portugais project "Sémantique des objets concurrents" funded visits of the first author. This work was also partially supported by the Portuguese *Fundação para a Ciência e a Tecnologia*, the PRAXIS XXI Projects 2/2.1/TIT/1658/95 LogComp and P/EEI/120598/98 DiCoMo, as well as by the ESPRIT Group 22704 ASPIRE.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, 1986. 476
2. Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, LNCS 1282. Springer-Verlag, 1997. 476
3. Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. 478

4. Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998. 476
5. Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche RR-1702, INRIA Sophia-Antipolis, 1992. 476
6. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Advances in Computing Science*, LNCS 1345, pages 239–253. Springer-Verlag, 1997. 480, 487
7. Søren Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis ECS-LFCS-93-278, Dep. of Computer Science, University of Edinburgh, 1993. 480
8. Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. A set constraint-based analyses of actors. In *Proc. of FMOODS'97*. IFIP, 1997. 484, 487
9. Laurent Dami. Labelled reductions, runtime errors, and operational subsumption. In *Proc. of ICALP'97*, LNCS 1256, pages 782–793. Springer-Verlag, 1997. 486
10. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. of CONCUR'96*, LNCS 1119, pages 406–421. Springer-Verlag, 1996. 476
11. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP'91*, LNCS 512, pages 141–162. Springer-Verlag, 1991. 476
12. Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proc. of LICS'97*, pages 128–139. Computer Society Press, 1997. 486
13. Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proc. of ICALP'98*, LNCS 1443, pages 856–967. Springer-Verlag, 1998. 476
14. Robin Milner. *Communication and Concurrency*. C. A. R. Hoare Series Editor—Prentice-Hall, 1989. 480
15. Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993. 474, 475
16. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992. 474, 477
17. Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *Proc. of FMOODS'97*. IFIP, 1997. 487
18. Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995. 474, 486
19. Franz Puntigam. Coordination Requirements Expressed in Types for Active Objects. In *Proc. of ECOOP'97*, LNCS 1241, pages 367–388. Springer-Verlag, 1997. 486
20. António Ravara and Luís Lopes. Programming and implementation issues in non-uniform TyCO. Research report DCC-99-1, Department of Computer Science, Faculty of Sciences, University of Porto. 475
21. António Ravara, Pedro Resende, and Vasco T. Vasconcelos. An algebra of behavioural types. Research report 26/99 DM-IST, Technical University of Lisbon. 475, 480
22. António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proc. of Euro-Par'97*, LNCS 1300, pages 554–561. Springer-Verlag, 1997. 475, 483
23. António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. Research report 6/00 DM-IST, Technical University of Lisbon. 486
24. Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *Proc. of HLCL'98*, *Electronic Notes in Theoretical Computer Science*, (16), 1998. 486

25. Vasco T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999. 474, 476
26. Vasco T. Vasconcelos and António Ravara. Communication errors in the π-calculus are undecidable. *Information Processing Letters*, 71(5–6):229–233, 1999. 480
27. Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Proc. of ISOTAS'93*, LNCS 742, pages 460–474. Springer-Verlag, 1993. 474, 475
28. Nobuko Yoshida. Graph types for monadic mobile processes. In *Proc. of FST/TCS'96*, LNCS 1180, pages 371-386. Springer-Verlag, 1996. 487

An Implicitly-Typed Deadlock-Free Process Calculus

Naoki Kobayashi, Shin Saito, and Eijiro Sumii

Department of Information Science, University of Tokyo
{koba,shin,sumii}@is.s.u-tokyo.ac.jp

Abstract. We extend Kobayashi and Sumii's type system for the deadlock-free π-calculus and develop a type reconstruction algorithm. Kobayashi and Sumii's type system helps high-level reasoning about concurrent programs by guaranteeing that communication on certain channels will eventually succeed. It can ensure, for example, that a process implementing a function really behaves like a function. However, because it lacked a type reconstruction algorithm and required rather complicated type annotations, applying it to real concurrent languages was impractical. We have therefore developed a type reconstruction algorithm for an extension of the type system. The key novelties that made it possible are generalization of *usages* (which specifies how each communication channel is used) and a *subusage* relation.

1 Introduction

General Background. With increasing opportunities of distributed programming, static guarantee of program safety is becoming extremely important, because (i) distributed programs are inherently concurrent and exhibit more complex behavior than sequential programs, (ii) it is hard to debug the whole distributed systems, and (iii) distributed programs usually involve many entities, some of which may be malicious. Lack of static guarantee results in unsafe or slow (due to expensive run-time check) program execution. Among various issues of program safety such as security, this paper focuses on problems caused by concurrency, in particular, deadlock (in a broad sense).

Traditional type systems are insufficient to guarantee the correctness of concurrent/distributed programs. Consider the following program of CML [14]:

```
fun f n = let val ch=channel() in recv(ch)+n+1 end;
```

The function `f` creates a new channel `ch` (by `channel()`), waits for a value v from the channel (by `recv(ch)`), and returns $v+n+1$. Since there is no sender on the channel `ch`, the application `f(1)` is blocked forever. Thus, `f` actually does not behave like a function, but the type system of CML assigns to `f` a function type $int \to int$.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 489–504, 2000.

Our Previous Type Systems for Deadlock-freedom and Their Problem. To overcome problems like above, a number of type systems [6,11,18] have been studied through π-calculus [9]. Our type systems for deadlock-freedom [5,16] are among the most powerful type systems: They can guarantee partial deadlock-freedom in the sense that communication on certain channels will eventually succeed. In addition to the usual meaning of deadlock-freedom where communications are blocked due to some circular dependencies, they also detect the situation like above, where there exists no communication partner from the beginning. Through the guarantee of deadlock-freedom, they can uniformly ensure that functional processes really behave like functions, that concurrent objects will eventually accept a request for method execution and send a reply, and that binary semaphores are really used like binary semaphores (a process that has acquired a semaphore will eventually release it unless it diverges).

In spite of the attractive features of the deadlock-free type systems, however, their applications to real concurrent programming languages have been limited. The main reason is that there was no reasonable type reconstruction algorithm and therefore programmers had to explicitly annotate programs with rather complex types.

Contributions of This Paper. To solve the above-mentioned problem, this paper develops an *implicitly-typed* version of the deadlock-free process calculus and its type reconstruction algorithm. Programmers no longer need to write complex type expressions; Instead, they just need to declare which communication they want to succeed. (Programmers may still want to partially annotate programs with types for documentation, etc.: Our algorithm can be easily modified to allow such partial type annotation.) For example, a process that sends a request to a function server or a concurrent object can be written as $(\nu r)\,(s![arg, r] \mid r?^{\mathbf{c}}[x]. \ldots)$. Here, (νr) creates a fresh channel r. $s![arg, r]$ sends a pair $[arg, r]$ to the server through channel s, and in parallel to this, $r?^{\mathbf{c}}[x]. \ldots$ waits on channel r to receive a reply from the server. The $\mathbf{c}$ attached to ? indicates that this input from r should eventually succeed, i.e., a reply should eventually arrive on r. If the whole system of processes (including the server process) is judged to be well typed in our type system, then it is indeed guaranteed that the input will eventually succeed, unless the whole system diverges.

Our new technical contributions are summarized as follows. (Those who are unfamiliar with our previous type systems can skip the rest of this paragraph.)

- Generalization of the previous type systems for deadlock-freedom — It is not possible to construct a reasonable type reconstruction algorithm for the previous type systems. So, we generalized them by introducing a subusage relation and new usage constructors such as recursive usages and the greatest lower bound of usages (which roughly correspond to the subtype relation, recursive types, and intersection types in the usual type system). A usage [16] is a part of a channel type and describes for which operations (input or output) and in which order channels can and/or must be used. It can be considered an extension of input/output modes [11] and multiplicities [6].

– Constraint-based type reconstruction algorithm — We have developed a type reconstruction algorithm, which inputs an implicitly-typed process and checks whether it is well typed or not. The algorithm is a non-trivial extension of Igarashi and Kobayashi's type reconstruction algorithm [4] for the linear π-calculus [6], where a principal typing is expressed as a pair of a type environment and a set of constraints on type/usage variables.

Limitations of This Paper. The type system and type reconstruction algorithm described in this paper have the following limitations.

– Incompleteness of the type reconstruction algorithm — The algorithm is sound but incomplete: Although it never accepts ill-typed processes, it rejects some well-typed processes. This is just because we want to reject some well-typed but bad processes that may livelock (i.e., diverge with keeping some process waiting for communication forever). So, our algorithm is actually preferable to a complete algorithm (if there is any).
– Naive treatment of time tags — The treatment of time tags and tag relations, which are key features of the deadlock-free type systems [5,16], is very naive in this paper. As a result, the expressive power is very limited. This is just for clarifying the essence of new ideas of this paper. It is easy to replace the naive treatment of time tags in this paper with the sophisticated one in the previous papers [5,16] and extend the type reconstruction algorithm accordingly. The resulting deadlock-free process calculus is more expressive than the previous calculi [5,16], which have already been shown to be expressive enough to encode the simply-typed λ-calculus with various evaluation strategies, semaphores, and typical concurrent objects.

The Rest of This Paper. Section 2 introduces the syntax and operational semantics of processes, and defines what we mean by deadlock. Section 3 gives a generalized type system. Section 4 describes a type reconstruction algorithm. Section 5 discusses related work, and Section 6 concludes this paper. For the space restriction, we omit proofs, some definitions, and details of the type reconstruction algorithm. They are given in the full version of this paper [7].

2 The Syntax and Operational Semantics of Processes

Our process calculus is a subset of the polyadic π-calculus [8]. Each input/output process can be annotated with the programmer's intention on whether or not the communication should succeed. After introducing its syntax and operational semantics, we define what we mean by deadlock.

2.1 Syntax and Operational Semantics of Processes

We first define the syntax of processes. The metavariables x and y_i range over a countably infinite set of variables.

Definition 1 (processes).

$$\begin{array}{lll} P \text{ (processes)} & ::= & \mathbf{0} \mid x!^b[v_1,\ldots,v_n].\,P \mid x?^b[y_1,\ldots,y_n].\,P \mid (P \mid Q) \mid (\nu x)\,P \\ & & \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid *P \\ v \text{ (values)} & ::= & \mathit{true} \mid \mathit{false} \mid x \\ b \text{ (annotations)} & ::= & \emptyset \mid \mathbf{c} \end{array}$$

Notation 2. We write $\tilde{y}$ for a sequence $y_1,\ldots,y_n$. As usual, $\tilde{y}$ in $x?^b[\tilde{y}].\,P$ and x in $(\nu x)\,P$ are called bound variables. The other variables are called free variables. We assume that α-conversions are implicitly applied so that bound variables are always different from each other and from free variables. $[\tilde{x} \mapsto \tilde{v}]P$ denotes a process obtained from P by replacing all free occurrences of x_i with v_i. We often write $x!^b[\tilde{y}]$ for $x!^b[\tilde{y}].\,\mathbf{0}$. We often omit the empty annotation $\emptyset$ and just write $x![\tilde{y}].\,P$ and $x?[\tilde{y}].\,P$ for $x!^{\emptyset}[\tilde{y}].\,P$ and $x?^{\emptyset}[\tilde{y}].\,P$ respectively.

$\mathbf{0}$ denotes inaction. A process $x!^b[\tilde{v}].\,P$ sends a tuple $[\tilde{v}]$ on x and then (after the tuple is received by some process) behaves like P. The annotation b expresses the programmer's intention: If it is $\mathbf{c}$, then the programmer expects that the output eventually succeeds, i.e., the tuple is received by some process. A process $x?^b[\tilde{y}].\,P$ receives a tuple $[\tilde{v}]$ on x, binds $\tilde{y}$ to $\tilde{v}$, and executes P. $P \mid Q$ executes P and Q in parallel, and $(\nu x)\,P$ creates a fresh channel x and executes P. $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ executes P if v is *true* and executes Q if v is *false*. $*P$ executes infinitely many copies of the process P in parallel.

Remark 3. In an earlier version of this paper [7], we included another annotation $\mathbf{o}$, which means that the annotated input/output operation must be executed. We removed it because it is not so useful and also because it complicates the type system.

The operational semantics is fairly standard: It is defined by using two relations: a structural congruence relation and a reduction relation [8].

Definition 4. *The* structural congruence relation $\equiv$ *is the least congruence relation closed under the rules: (i)* $P \mid \mathbf{0} \equiv P$, *(ii)* $P \mid Q \equiv Q \mid P$, *(iii)* $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, *and (iv)* $(\nu x)\,(P \mid Q) \equiv (\nu x)\,P \mid Q$ *(*x *not free in* Q*). The* reduction relation $\longrightarrow$ *is the least relation closed under the rules in Figure 1.*

2.2 Deadlock

We regard deadlock as a state where (i) processes can no longer be reduced, and (ii) a process is trying to perform an input or output operation annotated with $\mathbf{c}$, but has not succeeded to do so (because there is no corresponding output or input process). The latter condition is formally defined as follows.

Definition 5. *A predicate Waiting on processes is the least unary relation satisfying the following conditions: (i)* $\mathit{Waiting}(x!^{\mathbf{c}}[\tilde{v}].\,P)$, *(ii)* $\mathit{Waiting}(x?^{\mathbf{c}}[\tilde{y}].\,P)$, *and (iii)* $\mathit{Waiting}(P)$ *implies* $\mathit{Waiting}(P \mid Q)$, $\mathit{Waiting}(Q \mid P)$, $\mathit{Waiting}(*P)$, *and* $\mathit{Waiting}((\nu x)\,P)$.

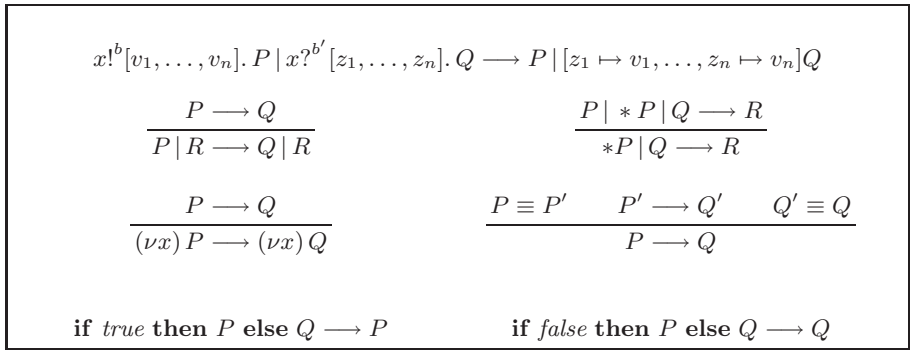


Fig. 1. Reduction Rules

Definition 6 (deadlock). *A process P is* in deadlock *if (i) there exists no P' such that $P \longrightarrow P'$ and (ii) Waiting(P) holds.*

This definition slightly differs from, but subsumes the usual definition of deadlock, which refers to a state where processes are blocked because of circular dependencies.

Example 7. Both $(\nu x)\,(x?^{\mathbf{c}}[\,].\,\mathbf{0})$ and $(\nu x)\,(\nu y)\,(x?^{\mathbf{c}}[\,].\,y![\,] \mid y?[\,].\,x![\,])$ are in deadlock because the input from x is annotated with $\mathbf{c}$, but the input cannot succeed. On the other hand, neither $(\nu x)\,(x?[\,].\,\mathbf{0})$ nor $(\nu x)\,(x![\,] \mid x?^{\mathbf{c}}[\,].\,\mathbf{0})$ is in deadlock.

3 Type System

Now, we give a type system that can guarantee freedom from the deadlock defined in the previous section. The basic idea of the type system is the same as that of our previous type system [16]: We augment ordinary channel types with *usages*, describing for what operations (input or output) and in which order each channel is used. To enable type reconstruction, this paper extends usages with new constructors and a relation between usages.

3.1 Usages

Definition 8 (usages). *The set of usages is given by the following syntax.*

$$\begin{array}{ll} U \text{ (usages)} & ::= 0 \mid \alpha \mid O_a.U \mid I_a.U \mid (U_1||U_2) \mid U_1 \sqcap U_2 \mid \mathbf{rec}\,\alpha.U \mid *U \\ a \text{ (attributes)} & ::= \emptyset \mid \mathbf{c} \mid \mathbf{o} \mid \mathbf{co} \end{array}$$

Here, α ranges over a countably infinite set of variables called usage variables*.*

Notation 9. $\mathbf{rec}\,\alpha.U$ binds α in U. Usage variables that are not bound are called *free* usage variables. $[\alpha \mapsto U']U$ denotes the usage obtained from U by

replacing all free occurrences of α with U'. We give a higher precedence to prefixes ($I_a.$, $O_a.$, $\mathbf{rec}\,\alpha.$, and $*$) than to $||$ and $\sqcap$. We also give a higher precedence to $\sqcap$ than to $||$.

0 is the usage of a channel that cannot be used at all. $O_a.U$ denotes the usage of a channel that can be first used for output, and then used according to U. The attribute $\mathbf{c}$ is called a *capability* and $\mathbf{o}$ an *obligation*. If a contains $\mathbf{c}$ (i.e., if a is $\mathbf{c}$ or $\mathbf{co}$), then the output is guaranteed to succeed. If a contains $\mathbf{o}$, then the channel must be used for output (even though the output may not succeed). Similarly, $I_a.U$ denotes the usage of a channel that can be first used for input with attribute a, and then used according to U. $U_1||U_2$ denotes the usage of a channel that can be used according to U_1 by one process and according to U_2 by another process, possibly in parallel. $U_1 \sqcap U_2$ denotes the usage of a channel that can be used according to either U_1 or U_2. For example, if x is a channel of the usage $I_\emptyset.0 \sqcap O_\emptyset.0$, then x can be used either for input or for output, but not for both. $\mathbf{rec}\,\alpha.U$ denotes the usage of a channel that can be used according to the infinite expansion of $\mathbf{rec}\,\alpha.U$ by $\mathbf{rec}\,\alpha.U = [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$. For example, $\mathbf{rec}\,\alpha.I_\emptyset.\alpha$ denotes the usage of a channel that can be used for input an infinite number of times sequentially. $*U$ denotes the usage of a channel that can be used according to U by infinitely many processes. For example, the usage of a binary semaphore is denoted by $O_{\mathbf{o}}.0||*I_{\mathbf{c}}.O_{\mathbf{o}}.0$, meaning that (i) there must be one initial output, (ii) there can be infinitely many input processes, and (iii) each input is guaranteed to eventually succeed (unless the whole process diverges), and it must be followed by output.

Remark 10. One may think that $*U$ can be replaced by $\mathbf{rec}\,\alpha.(\alpha||U)$. For a subtle technical reason, however, we need to distinguish between them.

Example 11. In the CML program given in Section 1, the usage of the channel `ch` is expressed as $I_a.0$. Because there is no output use, a cannot be $\mathbf{c}$ or $\mathbf{co}$, which implies that `recv(ch)` may be blocked forever. For another example, in the process $x?[\,].(x![\,] \mid x![\,])$, the usage of x can be expressed as $I_{a_1}.(O_{a_2}.0||O_{a_3}.0)$.

The constructors $\sqcap$ and $\mathbf{rec}\,\alpha.U$ are newly introduced in this paper. Although they are not necessary for the previous explicitly-typed calculus [16] (they would only add a little more expressive power), they are crucial for the implicitly-typed calculus in this paper. Suppose that a process P uses a channel x according to a usage U_1 and Q uses x according to U_2. Then, how can we express the usage of x by the process $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q$? With the choice constructor, we can express the most general usage of x as $U_1 \sqcap U_2$. (There is no problem in the case of *type check*: In order to check that $\mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ Q$ uses x according to U, we just need to check that both P and Q use x according to U.) Similarly, we do need a recursive usage, for example, to perform type reconstruction for a process $*x?[y].\,y![\,].\,x![y]$. Suppose that x is used to communicate a channel that should be used according to U. After receiving the channel y on x, the above process uses y for output, and then send it through x. The channel y will then be used according to U again. So, we have an equation $U = O.U$. With the recursive usage constructor, we can express a solution of this equation as $\mathbf{rec}\,\alpha.O.\alpha$.

$$U \succeq U||0 \qquad U_1||U_2 \succeq U_2||U_1 \qquad \frac{U_1 \succeq V_1 \qquad U_2 \succeq V_2}{U_1||U_2 \succeq V_1||V_2}$$

$$U_1 \sqcap U_2 \succeq U_i \qquad (U_1||U_2)||U_3 \succeq U_1||(U_2||U_3) \qquad \frac{U \succeq V}{*U \succeq *V}$$

$$*U \succeq *U||U \qquad \mathbf{rec}\,\alpha.U \succeq [\alpha \mapsto \mathbf{rec}\,\alpha.U]U$$

$$I_{a_1}.U_1||O_{a_2}.U_2||U_3 \longrightarrow U_1||U_2||U_3 \qquad \frac{U_1 \succeq V_1 \qquad V_1 \longrightarrow V_2 \qquad V_2 \succeq U_2}{U_1 \longrightarrow U_2}$$

Fig. 2. Usage Reduction

Usage Reduction. The usage of a channel changes during reduction of a process. For example, a process $x?[\,].x![\,] \,|\, x![\,]$ uses x as $I.O.0||O.0$. After the communication on x, however, the reduced process $x![\,]$ uses x as $O.0$. To express this change of a usage, we introduce a reduction relation on usages. Thus, usages themselves form a small process calculus, which has only one pair of co-actions I and O.

Following the usual reduction semantics of process calculi [8], we define usage reduction by using a structural relation on usages. For technical convenience, however, we do not require that the structural relation is symmetric.

Definition 12. *A usage preorder $\succeq$ and a usage reduction relation $\longrightarrow$ are the least binary relations on usages closed under the rules in Figure 2. $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.*

Usage Reliability. To avoid deadlock, we must require that the usage of each channel must be consistent (called *reliable*) in the sense that each input/output capability is always matched by a corresponding obligation. For example, $I_{\mathbf{c}}.0||O_{\mathbf{o}}.0$ is reliable, but $I_{\mathbf{c}}.0||O_{\mathbf{c}}.0$ is not. To define the reliability of a usage formally, we use the following predicate $ob_{\mathbf{I}}(U)$, which means that the usage U contains an input obligation and that there is no way to discard the obligation.

Definition 13. *Unary predicates $ob_{\mathbf{I}}, ob_{\mathbf{O}} (\subseteq \mathcal{U})$ on usages are defined by:*

$$ob_{\mathbf{I}}(U) \Longleftrightarrow \forall U_1.(U \succeq U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq I_a.U_2||U_3) \wedge (a \in \{\mathbf{o}, \mathbf{co}\})))$$
$$ob_{\mathbf{O}}(U) \Longleftrightarrow \forall U_1.(U \succeq U_1 \Rightarrow \exists a, U_2, U_3.((U_1 \succeq O_a.U_2||U_3) \wedge (a \in \{\mathbf{o}, \mathbf{co}\})))$$

Definition 14 (reliability). *A usage U is* reliable*, written $rel(U)$, if the following conditions hold for every U' with $U \longrightarrow^* U'$.*

1. *If $U' \succeq I_a.U_1||U_2$ and $a \in \{\mathbf{c}, \mathbf{co}\}$, then $ob_{\mathbf{O}}(U_2)$.*
2. *If $U' \succeq O_a.U_1||U_2$ and $a \in \{\mathbf{c}, \mathbf{co}\}$, then $ob_{\mathbf{I}}(U_2)$.*

Remark 15. The reliability of a usage is decidable: It can be reduced to the reachability problem of Petri nets [3].

Subusage. Some usage expresses more general use of a channel than other usages. For example, a channel of the usage $I_\emptyset.0||I_\emptyset.0$ can be used as that of the usage $I_\emptyset.I_\emptyset.0$, because the former usage allows two input operations to be executed in parallel. To express such a relation, we introduce a subusage relation $U_1 \leq U_2$, meaning that a channel of usage U_1 may be used as that of usage U_2.

The introduction of the subusage relation is essential for type reconstruction. For example, the usage of x by a process $x?[\,].x![\,]$ can be expressed both as $I_a.0||O_{\mathbf{c}}.0$ and as $I_a.O_{\mathbf{c}}.0$. Thanks to the subusage relation $I_a.0||O_{\mathbf{c}}.0 \leq I_a.O_{\mathbf{c}}.0$, however, we only need to consider the usage $I_a.O_{\mathbf{c}}.0$ during type reconstruction.

We first introduce a relation $a_1 \leq a_2$ between attributes, which means that a channel that should be used for an input/output operation with the attribute a_1 can be used for the operation with the attribute a_2. Note that $\mathbf{c} \leq \emptyset$ holds but $\mathbf{o} \leq \emptyset$ does not: For deadlock-freedom, it is fine not to use capabilities, but it should be disallowed not to fulfill obligations.

Definition 16 (sub-attribute). *The relation $\leq$ is the least partial order satisfying $\mathbf{c} \leq \emptyset$ and $\mathbf{co} \leq \mathbf{o}$.*

Because we have recursive usages, we need to define the subusage relation coinductively. Because usages themselves are mini-processes, it is natural to define it using a simulation relation.

Definition 17 (usage simulation). *A binary relation $\mathcal{R}(\subseteq \mathcal{U} \times \mathcal{U})$ on usages is called a* usage simulation *if the following conditions are satisfied for each $(U, U') \in \mathcal{R}$:*

1. *If $U' \succeq I_{a'}.U'_1||U'_2$, then there exist U_1, U_2, and a such that (i) $U \succeq I_a.U_1||U_2$, (ii) $U_2 \mathcal{R} U'_2$, (iii) $(U_1||U_2)\mathcal{R}(U'_1||U'_2)$, and (iv) $a \leq a'$.*
2. *If $U' \succeq O_{a'}.U'_1||U'_2$, then there exist U_1, U_2, and a such that (i) $U \succeq O_a.U_1||U_2$, (ii) $U_2 \mathcal{R} U'_2$, (iii) $(U_1||U_2)\mathcal{R}(U'_1||U'_2)$, and (iv) $a \leq a'$.*
3. *$ob_{\mathbf{I}}(U)$ implies $ob_{\mathbf{I}}(U')$, and $ob_{\mathbf{O}}(U)$ implies $ob_{\mathbf{O}}(U')$.*
4. *If $U' \longrightarrow U'_1$, then there exists U_1 such that $U \longrightarrow U_1$ and $U_1 \mathcal{R} U'_1$.*

The first and second conditions mean that in order for U to simulate U', U must allow any input/output operations that U' allows. The third condition means that U' must provide any obligations that U provides. The fourth condition means that such conditions are preserved even after reductions.

Definition 18. *A* subusage relation $\leq$ *on usages is the largest usage simulation.*

Example 19. $I_{\mathbf{c}}.U \leq 0$ and $I_{\mathbf{c}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{c}}.0$ hold, but neither $I_{\mathbf{o}}.U \leq 0$ nor $I_{\mathbf{o}}.0||I_{\mathbf{c}}.0 \leq I_{\mathbf{c}}.I_{\mathbf{o}}.0$ holds. $U_1 \sqcap U_2 \leq U_i$ holds for any U_1 and U_2.

3.2 Types, Type Environments, and Type Judgment

The syntax of types is defined as follows. The metavariable t ranges over a countable set $\mathbf{T}$ of labels called *time tags*.

Definition 20 (types). $\tau ::= \mathit{bool} \mid [\tau_1, \ldots, \tau_n]^t/U$

$[\tau_1, \ldots, \tau_n]^t/U$ denotes the type of a channel that can be used for communicating a tuple of values of types $\tau_1, \ldots, \tau_n$. The channel must be used according to the usage U. As in the previous type systems, the time tag t is used to control the order between communications on different channels. The allowed order is specified by the following tag ordering.

Definition 21. *A* tag ordering*, written $\mathcal{T}$, is a strict partial order (i.e., a transitive and irreflexive binary relation) on* $\mathbf{T}$.

Intuitively, $s\mathcal{T}t$ means that a process can use capabilities to communicate on a channel tagged with s before fulfilling obligations to communicate on a channel tagged with t. For example, if a channel x has type $[bool]^{t_x}/I_{\mathbf{c}}.0$ and y has type $[bool]^{t_y}/O_{\mathbf{o}}.0$, and if $t_x \mathcal{T} t_y$ holds, then a process can wait to receive a boolean on x before fulfilling the obligation to send a boolean on y.

Type Environment. A type environment is a mapping from a finite set of variables to types. We use a metavariable Γ for a type environment. If $\tau_i = bool$ for each i such that $v_i = true$ or $false$, then $v_1 : \tau_1, \ldots, v_n : \tau_n$ denotes the type environment Γ such that $dom(\Gamma) = \{v_1, \ldots, v_n\} \backslash \{true, false\}$ and $\Gamma(v_i) = \tau_i$ for each $v_i \in dom(\Gamma)$. We write $\emptyset$ for the type environment whose domain is empty. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Γ' satisfying $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. $\Gamma \backslash \{x_1, \ldots, x_n\}$ denotes the type environment Γ' such that $dom(\Gamma') = dom(\Gamma) \backslash \{x_1, \ldots, x_n\}$ and $\Gamma'(x) = \Gamma(x)$ for each $x \in dom(\Gamma')$.

Type Judgment. A type judgment is of the form $\Gamma; \mathcal{T} \vdash P$. It means that P uses each channel as specified by Γ, and that P obeys the constraints on the order of communications specified by $\mathcal{T}$. For example, let $\Gamma = x : []^{t_x}/I_{\mathbf{co}}.0, y : []^{t_y}/O_{\mathbf{co}}.0$ and $\mathcal{T} = \{(t_x, t_y)\}$. Then, $\Gamma; \mathcal{T} \vdash x?^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ is a valid judgment, but neither $\Gamma; \mathcal{T} \vdash x!^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ nor $\Gamma; \mathcal{T} \vdash y!^{\mathbf{c}}[\,].\, x?^{\emptyset}[\,].\, \mathbf{0}$ is: The process $x!^{\mathbf{c}}[\,].\, y!^{\emptyset}[\,]$ wrongly uses x for output, and $y!^{\mathbf{c}}[\,].\, x?^{\emptyset}[\,].\, \mathbf{0}$ communicates on x and y in a wrong order.

Operations and Relations on Types and Type Environments. Constructors and relations on usages are extended to operations and relations on types and type environments, as defined in Figure 3. Note that binary operations on types are partial: For example, $bool || [\tilde{\tau}]^t/U$ is undefined.

Intuitively, the type environment $\Gamma_1 || \Gamma_2$ indicates that channels can be used according to Γ_1 by one process and used according to Γ_2 by another process in parallel. So, if P_1 uses channels according to Γ_1 (i.e., P_1 is well typed under Γ_1) and P_2 uses them according to Γ_2, then $P_1 \mid P_2$ uses them according to $\Gamma_1 || \Gamma_2$ in total. Similarly, **if** $true$ **then** P_1 **else** P_2 and $*P_1$ use channels according to $\Gamma_1 \sqcap \Gamma_2$ and $*\Gamma_1$ respectively.

The relation $t\mathcal{R}\Gamma$ means that a process is allowed to use a capability on a channel tagged with t before fulfilling obligations contained in Γ.

– Unary/binary operations ($\mathbf{op} = ||, \sqcap$)

$$*bool = bool \qquad *[\tilde{\tau}]^t/U = [\tilde{\tau}]^t/*U \qquad (*\Gamma)(x) = *(\Gamma(x))$$
$$bool\,\mathbf{op}\,bool = bool \qquad ([\tilde{\tau}]^t/U_1)\,\mathbf{op}\,([\tilde{\tau}]^t/U_2) = [\tilde{\tau}]^t/(U_1\,\mathbf{op}\,U_2)$$
$$(\Gamma_1\,\mathbf{op}\,\Gamma_2)(x) = \begin{cases} \Gamma_1(x)\,\mathbf{op}\,\Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_i(x) & \text{if } x \in dom(\Gamma_i) \backslash dom(\Gamma_{3-i}) \end{cases}$$

– Subtyping

$$bool \leq bool \qquad [\tilde{\tau}]^t/U_1 \leq [\tilde{\tau}]^t/U_2 \text{ if } U_1 \leq U_2$$

– Unary predicates

$$noob(bool) \qquad noob([\tilde{\tau}]^t/U) \text{ if } U \leq 0$$
$$noob(\Gamma) \text{ if } noob(\Gamma(x)) \text{ for each } x \in dom(\Gamma)$$

– Tag ordering

$$t\mathcal{T}\tau \text{ if } noob(\tau) \vee (\tau = [\tilde{\tau'}]^s/U \wedge t\mathcal{T}s)$$
$$t\mathcal{T}\Gamma \text{ if } t\mathcal{T}(\Gamma(x)) \text{ for each } x \in dom(\Gamma)$$

Fig. 3. Operations and Relations on Types and Type Environments

3.3 Typing Rules

The set of typing rules for deriving valid type judgments are given in Figure 4.

In the rule for **0**, we require that the type environment contains no obligation, because **0** does nothing. As explained in the previous subsection, in the rules for $P \,|\, Q$, **if** b **then** P **else** Q, and $*P$, the type environment of the processes are computed by combining the type environments of their sub-processes with operations $||$, $\sqcap$, and $*$. In the rule for $(\nu x)\,P$, we require that x has a channel type and its usage is reliable.

The rule for output processes is a key rule. The premise $\Gamma, x : [\tilde{\tau}]^t/U; \mathcal{T} \vdash P$ implies that P uses x according to U. Because the process $x!^b[\tilde{v}].\,P$ uses x for output before doing so, the total usage of x is expressed by $O_a.U$. The other variables may be used by P or by a receiver on x, possibly in parallel. The former use is expressed by Γ, while the latter use is by $v_1 : \tau_1 || \cdots || v_n : \tau_n$. Thus, the type environment of the whole process is given by $x : [\tilde{\tau}]^t/O_a.U || v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma$. If the annotation b is $\mathbf{c}$, the input must succeed, hence the condition $a \in \{\mathbf{c}, \mathbf{co}\}$. Moreover, we require the conditions $(\neg noob(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)) \Rightarrow a \in \{\mathbf{c}, \mathbf{co}\}$ and $t\mathcal{T}(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)$ to enforce the consistency among different channels. The first condition means that if the process P or the tuple $[v_1, \ldots, v_n]$ contains some obligations, i.e., if $noob(v_1 : \tau_1 || \cdots || v_n : \tau_n || \Gamma)$ does not hold, then the output on x must be guaranteed to succeed (so that it does not block the fulfillment of the obligations). The second condition is required because this output process uses the capability to output on x *before* fulfilling the obligations possibly contained in P and $[v_1, \ldots, v_n]$: Such dependency must be allowed by the tag ordering $\mathcal{T}$. The rule for input processes is similar.

$$\frac{noob(\Gamma)}{\Gamma;\mathcal{T}\vdash \mathbf{0}} \qquad \frac{\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U;\mathcal{T}\vdash P \qquad rel(U)}{\Gamma;\mathcal{T}\vdash (\nu x)\,P}$$

$$\frac{\Gamma_1;\mathcal{T}\vdash P_1 \qquad \Gamma_2;\mathcal{T}\vdash P_2}{\Gamma_1||\Gamma_2;\mathcal{T}\vdash P_1\mid P_2} \qquad \frac{\Gamma_1;\mathcal{T}\vdash P \qquad \Gamma_2;\mathcal{T}\vdash Q}{(\Gamma_1\sqcap\Gamma_2)||v:bool;\mathcal{T}\vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

$$\frac{\Gamma;\mathcal{T}\vdash P}{*\Gamma;\mathcal{T}\vdash *P} \qquad \frac{\Gamma, x:\tau';\mathcal{T}\vdash P \qquad \tau\leq\tau'}{\Gamma, x:\tau;\mathcal{T}\vdash P}$$

$$\frac{\begin{array}{c}\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U;\mathcal{T}\vdash P\\ b=\mathbf{c}\Rightarrow a\in\{\mathbf{c},\mathbf{co}\} \qquad t\mathcal{T}(v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma)\\ (\neg noob(v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma))\Rightarrow a\in\{\mathbf{c},\mathbf{co}\}\end{array}}{x:[\tau_1,\ldots,\tau_n]^t/O_a.U||v_1:\tau_1||\cdots||v_n:\tau_n||\Gamma;\mathcal{T}\vdash x!^b[v_1,\ldots,v_n].\,P}$$

$$\frac{\begin{array}{c}\Gamma, x:[\tau_1,\ldots,\tau_n]^t/U, y_1:\tau_1,\ldots,y_n:\tau_n;\mathcal{T}\vdash P\\ b=\mathbf{c}\Rightarrow a\in\{\mathbf{c},\mathbf{co}\} \qquad t\mathcal{T}\Gamma \qquad (\neg noob(\Gamma))\Rightarrow a\in\{\mathbf{c},\mathbf{co}\}\end{array}}{\Gamma, x:[\tau_1,\ldots,\tau_n]^t/I_a.U;\mathcal{T}\vdash x?^b[y_1,\ldots,y_n].\,P}$$

Fig. 4. Typing Rules

Example 22. A process $P = *f?[x,r].\,r![x]$ implements the identity function, since it just forwards the argument x to the reply address r. We can obtain the following judgment:

$$\emptyset;\{(t_f,t_y)\}\vdash (\nu f)\,(P\mid(\nu y)\,f!^{\mathbf{c}}[true,y].\,y?^{\mathbf{c}}[z].\,\mathbf{0}).$$

The process $f!^{\mathbf{c}}[true,y].\cdots$ calls the function located at f and waits for a reply. t_f and t_y are time tags of channels f and y. The judgment indicates that the caller process can eventually receive a reply.

3.4 Deadlock Freedom Theorem

Theorem 23. *If $\emptyset;\mathcal{T}\vdash P$ and $P\longrightarrow^* Q$, then Q is not in deadlock.*

As in the previous type systems [5,16], this theorem is proved as a corollary of the subject reduction property and lack of immediate deadlock. A proof is given in the full paper [7]. The intuitive reasons why the deadlock-freedom holds are: (i) each rule correctly estimates the usage of each channel, and (ii) the side condition $rel(U)$ of (T-New) guarantees that each channel is consistently used, and (iii) the tag ordering guarantees that there is no cyclic dependency between different channels.

4 Type Reconstruction

Thanks to the generalization of the previous type systems [5,16] made in the last section, it is now possible to develop a type reconstruction algorithm. Type

reconstruction proceeds in a manner similar to Igarashi and Kobayashi's type reconstruction algorithm for linear π-calculus [4]. We first transform the typing rules into syntax-directed typing rules, so that there is only one applicable rule for each process expression. Then, by reading the syntax-directed rules in a bottom-up manner, we obtain an algorithm for extracting a principal typing. Finally, we decide the typability of a process by solving the constraint part of the principal typing. For lack of space, we explain the algorithm only through an example. The concrete description of the algorithm is given in the full paper [7].

The key properties of our new type system that enabled type reconstruction are (i) there is only one rule for each process constructor except for the subsumption rule (the right rule in the third line in Figure 4, which can be merged with other rules), and (ii) a most general typing can be expressed by using the new usage constructors and subusage relation. The property (i) does not hold for our earlier type system [5] and other type systems that guarantee certain deadlock-freedom properties [2,15,18]: They have different rules for input/output on different types of channels.

4.1 Principal Typing

As in Igarashi and Kobayashi's type system [4], a principal typing can be expressed by using constraints. We introduce variables ranging over attributes, usages, and types, and accordingly extend the syntax of attributes, usages, types, and type environments.

A principal typing of a process P is defined as a pair (Γ, C) of an extended type environment and a set of constraints, satisfying the following conditions: (i) Any type judgment obtained by substituting a solution of C for $\Gamma; \mathcal{T} \vdash P$ is valid (i.e., (Γ, C) expresses only valid typings). (ii) Every valid type judgment can be obtained by substituting a solution of C for $\Gamma; \mathcal{T} \vdash P$ (i.e., (Γ, C) expresses all the valid typings). A formal definition is given in the full paper [7].

4.2 Algorithm for Computing a Principal Typing

We can easily eliminate the subsumption rule by combining it with other rules. By reading the resulting syntax-directed rules in a bottom-up manner, we can construct an algorithm for computing a principal typing. For example, we obtain the following rule by combining the rules for $(\nu x)\, P$ and subsumption:

$$\frac{\Gamma; \mathcal{T} \vdash P \qquad (\Gamma(x) = [\tau_1, \ldots, \tau_n]^t / U \wedge rel(U)) \vee x \notin dom(\Gamma)}{\Gamma \backslash \{x\}; \mathcal{T} \vdash (\nu x)\, P}$$

This implies that a principal typing (Γ, C) of $(\nu x)\, P$ can be computed from a principal typing (Γ', C') of P as follows:

$$(\Gamma, C) = \begin{cases} (\Gamma' \backslash \{x\}, C \cup \{rel(\Gamma'(x))\}) & \text{if } x \in dom(\Gamma) \\ (\Gamma', C') & \text{otherwise} \end{cases}$$

Example 24. For the process $(\nu f)\,(P \mid (\nu y)\, f!^{\mathbf{c}}[true, y].\, y?^{\mathbf{c}}[z].\, \mathbf{0})$ given in Example 22, the following pair is a principal typing:

$$\begin{array}{l}(\emptyset, \{\rho_f \le (*([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)||[bool, \rho_r]^{t_f}/O_{a_2}.0), rel(\rho_f), a_2 \in \{\mathbf{c}, \mathbf{co}\}\\ \quad \rho_y \le (\rho_r||[\rho_z]^{t_y}/I_{a_4}.0), rel(\rho_y), a_4 \in \{\mathbf{c}, \mathbf{co}\}\\ \quad \rho_r \le [\rho_x]^{t_y}/O_{a_3}.0, noob(\rho_z), (\neg noob(\rho_r) \vee a_4 \in \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\})\end{array}$$

Here, ρ_f, ρ_x, ρ_y, ρ_z, and ρ_r are type variables representing the types of f, x, y, z, and r respectively. The constraints in the first line are those on the uses of the channel f. Because f is used by P as a value of type $*([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)$ and used by the other process as a value of type $[bool, \rho_r]^{t_f}/O_{a_2}.0$, the type ρ_f of f is constrained by $\rho_f \le *([\rho_x, \rho_r]^{t_f}/I_{a_1}.0)||[bool, \rho_r]^{t_f}/O_{a_2}.0$. The second line shows constraints on the uses of the channel y. The last constraint in the third line comes from the dependency between f and y.

4.3 Constraint Solving

We can decide the typability of a process by reducing the constraints in its principal typing and checking their satisfiability. We reduce the set of constraints on types, those on usages, those on attributes and those on time tags step by step in this order, in a similar (but more complex) manner to Igarashi and Kobayashi's algorithm [4].

The algorithm for reducing constraints on usages is actually incomplete. The completeness is lost in the second step explained in Example 25 below, where a subusage constraint $\alpha \le U$ is replaced by $\alpha = \mathbf{rec}\,\alpha.U$. As mentioned in Section 1, this is because we want to reject some well-typed but bad processes. (So, we do not want to require the completeness.) The other transformation steps are sound and complete: In those steps, constraints can be transformed into simpler, equivalent constraints. For the efficiency reason, however, our current prototype type inference system use an approximate (sound but incomplete) algorithm also in the third step.

Example 25. Consider the constraint set shown in Example 24. Our algorithm roughly proceeds as follows.

1. Reduce constraints on types: In a subtyping constraint $\tau_1 \le \tau_2$ and an expression $\tau_1\,\mathbf{op}\,\tau_2$, τ_1 and τ_2 must be identical except for usages. By instantiating type variables so that this condition is met (which is performed by the first-order unification), we get the following constraint set on usages (ρ_f, ρ_y, ρ_r, ρ_x, and ρ_z were instantiated with $[bool, [bool]^{t_y}/\alpha_r]^{t_f}/\alpha_f$, $[bool]^{t_y}/\alpha_y$, $[bool]^{t_y}/\alpha_r$, $bool$, and $bool$, respectively):

$$\begin{array}{l}\{\alpha_f \le *I_{a_1}.0||O_{a_2}.0, rel(\alpha_f), a_2 \in \{\mathbf{c}, \mathbf{co}\}, \alpha_y \le \alpha_r||I_{a_4}.0, rel(\alpha_y),\\ a_4 \in \{\mathbf{c}, \mathbf{co}\}, \alpha_r \le O_{a_3}.0, (\neg noob(\alpha_r) \vee a_4 \in \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\}\end{array}$$

2. Reduce subusage constraints: From the subusage constraints, we obtain $\alpha_f = *I_{a_1}.0||O_{a_2}.0$, $\alpha_y = O_{a_3}.0||I_{a_4}.0$, and $\alpha_r = O_{a_3}.0$ as a representative solution. By substituting it for the other constraints, we obtain: $\{rel(*I_{a_1}.0||O_{a_2}.0), a_2 \in \{\mathbf{c}, \mathbf{co}\}, rel(O_{a_3}.0||I_{a_4}.0), a_4 \in \{\mathbf{c}, \mathbf{co}\}, (\{a_3, a_4\} \subseteq \{\mathbf{o}, \mathbf{co}\}) \Rightarrow t_f \mathcal{T} t_y\}$.

3. Reduce the other constraints on usages: By reducing the reliability constraints, we obtain: $\{a_1 \notin \{\mathbf{c}, \mathbf{co}\}, a_1 \in \{\mathbf{o}, \mathbf{co}\}, a_2 \in \{\mathbf{c}, \mathbf{co}\}, a_3 \in \{\mathbf{o}, \mathbf{co}\}, a_4 \in \{\mathbf{c}, \mathbf{co}\}, t_f \mathcal{T} t_y\}$.
4. Reduce the constraints on usage attributes: Start with $a_1 = a_2 = a_3 = \emptyset$, and increment the attributes step by step until the whole constraints are satisfied. In this case, we have $a_1 = \mathbf{o}$, $a_2 = \mathbf{c}$, $a_3 = \mathbf{o}$, and $a_4 = \mathbf{c}$ as a solution, and obtain $t_f \mathcal{T} t_y$.
5. Check whether the remaining constraints on the tag ordering is satisfiable. In this case, we have only the constraint $t_f \mathcal{T} t_y$, which is clearly satisfiable. The process is therefore accepted as a well-typed process.

Recursive usages and greatest lower bounds play an important role in solving usage constraints (the step 2 above). For example, given subusage constraints $\alpha \leq O_a.0$ and $\alpha \leq I_{a'}.\alpha$, we can first transform them into $\alpha \leq O_a.0 \sqcap I_{a'}.\alpha$, and then obtain $\alpha = \mathbf{rec}\,\alpha.(O_a.0 \sqcap I_{a'}.\alpha)$ as a representative solution. This is not always possible without recursive usage and greatest lower bound constructors.

5 Related Work

Several type systems guaranteeing certain deadlock-freedom properties have recently been proposed [1,2,5,13,15,16,18]. As far as we know, however, no type reconstruction algorithm has been developed for them so far. One of the main difficulties of type reconstruction for those type systems is that they use different rules for input/output on different types of channels. We solved that problem by generalizing our previous type systems.

One of the key ideas was to use a process-like term to describe the channel-wise behavior of a process. Similar ideas are found in earlier type systems [17,10]: For example, Nierstrasz [10] used CCS-like terms as types of concurrent objects and defined subtyping relations.

Our type reconstruction algorithm can be considered a non-trivial extension of Igarashi and Kobayashi's type reconstruction algorithm [4] for the linear π-calculus [6].

6 Conclusion

We have extended our previous type systems for deadlock-freedom [5,16] and developed its type reconstruction algorithm. A prototype type inference system is available at http://www.yl.is.s.u-tokyo.ac.jp/~shin/pub/.

There remain a number of issues in applying our type system and algorithm to real concurrent programming languages [12,14], such as whether the type system is expressive enough, how to make the algorithm efficient, and how to present the result of type reconstruction to programmers. We plan to perform experiments using existing CML or Pict programs to answer these questions.

Acknowledgment

We would like to thank Atsushi Igarashi for useful comments.

References

1. G. Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings of ASIAN'97*, LNCS 1345, pages 239–253, 1997. 502
2. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, LNCS 1782, pp.180–199, 2000. 500, 502
3. J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30(3):143–160, 1994. 495
4. A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*. To appear. A preliminary summary appeared in Proceedings of SAS'97, LNCS 1302, pp.187-201. 491, 500, 501, 502
5. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. 490, 491, 499, 500, 502
6. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999. A preliminary summary appeared in Proceedings of POPL'96, pp.358-371. 490, 491, 502
7. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. Technical Report TR00-01, Dept. Info. Sci., Univ. of Tokyo, January 2000. Available at http://www.yl.is.s.u-tokyo.ac.jp/~koba/publications.html. 491, 492, 499, 500
8. R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993. 491, 492, 495
9. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I, II. *Information and Computation*, 100:1–77, September 1992. 490
10. O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995. A preliminary version appeared in Proceedings of OOPSLA'93, pp.1-15. 502
11. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. 490
12. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. To appear in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, MIT Press, 2000 502
13. F. Puntigam. Coordination requirements expressed in types for active objects. In *Proceedings of ECOOP'97*, LNCS 1241, pages 367–388, 1997. 502
14. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of PLDI'91*, pages 293–305, 1991. 489, 502
15. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2), pages 457–493, 1999. 500, 502
16. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *Proc. of Workshop on High-Level Concurrent Language (HLCL'98)*, *ENTCS* 16(3), pages 55–77, 1998. 490, 491, 493, 494, 499, 502

17. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings of PARLE'94*, LNCS 817, pages 398–413, 1994. 502
18. N. Yoshida. Graph types for monadic mobile processes. In *FST/TCS'16*, LNCS 1180, pages 371–387, 1996. 490, 500, 502

Typed Mobile Objects*

Michele Bugliesi[1], Giuseppe Castagna[2], and Silvia Crafa[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia
{michele,silvia}@dsi.unive.it
http://www.dsi.unive.it/~{michele,silvia}
[2] C.N.R.S., Département d'Informatique, École Normale Supérieure, Paris
Giuseppe.Castagna@ens.fr
http://www.di.ens.fr/~castagna

Abstract. We describe a general model for embedding object-oriented constructs into calculi of mobile agents. The model results from extending agents with methods and primitives for message passing. We then study an instance of the model based on Cardelli and Gordon's Mobile Ambients. We define a type system for the resulting calculus, give a subject reduction theorem, and discuss the rôle of the type system for static detection of run-time type errors and for other program verification purposes.

1 Introduction

Calculi of mobile agents are receiving increasing interest in the programming language community as advances in computer communications and hardware enhance the development of large-scale distributed programming. Independently of the new trends in communication technology, object-oriented programming has established itself as the de-facto standard for a principled design of complex software systems.

Drawing on our preliminary work on the subject [2], in this paper we develop a general model for integrating object-oriented constructs into calculi of mobile agents. The resulting framework can be looked at in different ways: (*i*) as a *concurrent* object calculus where objects have explicit names, in the style of [9], (*ii*) as a *distributed* object calculus where objects are stored at different named locations, or (*iii*), more ambitiously, as a model for distributed computation, where conventional client-server technology based on (remote) exchange of message between agents, and mobile agents coexist.

The model results from extending the structure of *named* agents with method definitions and primitives for dealing with message passing and self denotations. The extension has interesting payoffs, as it leads to a principled approach to structuring agents. In particular, introducing methods and message passing as primitives, rather than encoding them on top of the underlying calculus of agents

* Work partially supported by MURST Project 9901403824_003 and by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet".

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 504–520, 2000.

leads to a rich and precise notion of agent interface and type. Furthermore, it opens the way to reusing the advances in type system of object-oriented programming and static analysis.

After giving an outline of the general model, we describe an instance of that model, named MA^{++}, based on the calculus of *Mobile Ambients* (henceforth MA) of [3,5]. We illustrate the calculus with a number of variated examples. Specifically, we show that it is possible to encode primitives like method overriding distinctive of object calculi; we also show that various forms of process communication can be encoded (here, and throughout the paper, we use the terms "encode" and "encoding" in somewhat loose sense: we should in fact use "simulate" and "simulation" as we don't claim these encodings to be "atomic" – i.e. free of interferences – in all possible contexts.).

Then we study the type theory of our calculus. The motivation for doing that is twofold: (*i*) a sound type system can statically detect run-time type errors such as the classical "message not understood" errors distinctive of object-oriented languages and calculi; (*ii*) an expressive type system eases the definition of program equivalences and their proof (see for example [13]). The present paper gives an in-depth account of the former aspect, and only hints at the latter (see Section 6), leaving a detailed treatment for future work.

Several proposals of formalisms for foundational study of concurrent object-oriented programming can be found in the literature (e.g., [9,12,15,16,18,7,14]): however, to our knowledge, no previous work directly embeds methods into calculi of mobile agents.

2 Outline of the Model

The definition of the model is given parametrically on the underlying calculus of processes and agents so that it can be adapted to formalisms such as Mobile Ambients [5], Safe Ambients [13], or the Seal Calculus [17]. The minimal requirement we assume for the underlying calculus of mobile agents is the existence of the following constructs: $\mathbf{0}$ denoting the inactive process, $P \mid Q$, denoting the parallel composition of two processes P and Q, $a[P]$, denoting the process (or agent) named a running the process P, $(\nu x)P$, that restricts the name x to P, and finally $A.P$, that performs the action described by the expression A and then continues as P. Clearly, the actions will eventually include primitives for moving agents over the locations of the distributed system, but we may disregard those primitives at this stage. What instead is central to the model is the naming mechanism for processes. Named processes are abstractions of both agents and locations: furthermore, since naming allows nesting, locations may be structured hierarchically [3]. As a consequence, named processes model both the nodes of the distributed system and the agents over that system.

2.1 Syntax

The generic model of mobile objects results from generalizing the structure of named agents to include method definitions, as in $a[M\,;\,P]$, where P is a process

and M a set (rather, a list) of method definitions. The syntax of agents is defined by the productions in Figure 1.

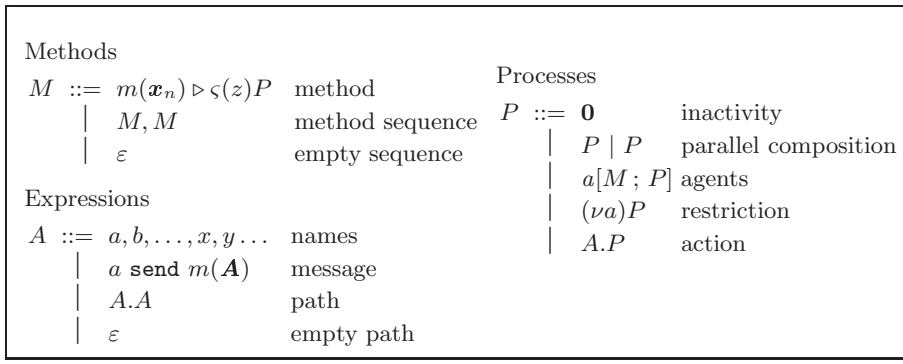


Fig. 1. Syntax of Agents

Notation. In the following we use P, Q, R to range over processes, L, M, N to range over (possibly empty) method sequences, and lower case letters to range over generic names, reserving $a, b, \ldots$ for agent names, and $x, y, \ldots$ for (method) parameters. Method names, denoted by m and n range over a disjoint alphabet and have a different status: they are fixed labels that may not be restricted, abstracted, substituted, nor passed as values (they are similar to field labels in record-based calculi). We omit trailing or isolated $\mathbf{0}$ processes and empty method sequences, using A, $a[M\,;\,]$, $a[P]$, and $a[\,]$ to abbreviate $A.\mathbf{0}$, $a[M\,;\,\mathbf{0}]$, $a[\varepsilon\,;\,P]$, and $a[\varepsilon\,;\,\mathbf{0}]$ respectively.

We sometimes write $a[(m_i(\boldsymbol{x}_{k_i}) \triangleright \varsigma(z_i)P_i)_{i\in[1..n]}\,;\,P]$ as a shorthand for the agent $a[m_1(\boldsymbol{x}_{k_1}) \triangleright \varsigma(z_1)P_1, \ldots, m_n(\boldsymbol{x}_{k_n}) \triangleright \varsigma(z_n)P_n\,;\,P]$. Similarly we write $\boldsymbol{x}_n$ as a shorthand for $x_1, \ldots, x_n$ and omit the subscript n when there is no risk of ambiguity. Finally, $P\{\boldsymbol{x}_n := \boldsymbol{y}_n\}$ denotes the term resulting from substituting every free occurrence of x_i by y_i in P; equivalent notations that we also use are $P\{\boldsymbol{x}_{n-1} := \boldsymbol{y}_{n-1}, x_n := y_n\}$ and $P\{\boldsymbol{x}_{n-1}, x_n := \boldsymbol{y}_{n-1}, y_n\}$.

Methods. A method definition has the form $m(\boldsymbol{x}_n) \triangleright \varsigma(z)P$, where the m is the name of the method, $\boldsymbol{x}_n$ the list of its formal parameters, and $\varsigma(z)P$ its body. As in the Object Calculi of [1], the ς-bound variable z is the "internal" name of the host agent: the scope of the ς-binder is the method body P. Since agents are named, explicit names could be used to invoke methods from any agent, including sibling methods from the same agent: however, as we shall see, the *late binding* semantics associated with *self* ensures a smooth and elegant integration of method invocation and agent mobility. The syntax of methods is abbreviated to $m(\boldsymbol{x}_n) \triangleright P$ when z does not occur free in P, or when the presence of the binder is irrelevant to the context in question.

Expressions. The definition and associated behavior of expressions is what distinguishes different calculi for mobility. In the Seal Calculus, for instance, expressions that perform mobility use channels synchronization, and act on agents that are passive with respect to mobility. In Mobile Ambients, instead, mobility expressions are exercised by the moving agents themselves, without intervention by the surrounding environment (Safe Ambients add a synchronization mechanism to Mobile Ambients). Since we wish our extension to be independent of the underlying primitives for mobility, at the present stage we define expressions to be agent *names*, messages-sends, or *paths* that define composite expressions. Message-sends are denoted by a `send` $m(\boldsymbol{A}_n)$, where a is the name of an agent, and $m(\boldsymbol{A}_n)$ invokes the m method found in a with arguments $\boldsymbol{A}_n$. Unlike [1], the format of a message-send requires that the recipient be the *name* of an agent rather than the agent (the object, in [1]) itself. The semantics of message sends is discussed in detail in Section 2.3.

2.2 Structural Congruence

Structural congruence for agents is defined in terms of a relation of equivalence over method sequences, given in Figure 2. The intention is to allow method suites to be reordered without affecting the behavior of the enclosing agent.

$(L, M), N \leftrightharpoons L, (M, N)$	(Eq Meth Assoc)
$n \neq m \Rightarrow M, m(x) \triangleright P, n(y) \triangleright Q \leftrightharpoons M, n(y) \triangleright Q, m(x) \triangleright P$	(Eq Meth Comm)
$M, m(x) \triangleright P, m(x) \triangleright Q, M \leftrightharpoons M, m(x) \triangleright Q$	(Eq Meth Over)
$M \leftrightharpoons M$	(Eq Meth Refl)
$M \leftrightharpoons N \Rightarrow N \leftrightharpoons M$	(Eq Meth Symm)
$L \leftrightharpoons M, M \leftrightharpoons N \Rightarrow L \leftrightharpoons N$	(Eq Meth Trans)

Fig. 2. Equivalence for Methods

Definitions for methods with different name and/or arity may be freely permuted; instead, if the same method has multiple definitions, then the rightmost definition overrides the remaining ones. Similar notions of equivalence between method suites can be found in the literature on objects: in fact, our definition is directly inspired by the bookkeeping relation introduced in [8].

The relation structural congruence of processes is defined as the smallest congruence on processes that forms a commutative monoid with product | and unit **0**, and is closed under the rules in Figure 3. These rules are parametric in the definition of expressions and *free names*, which vary depending on the specific calculus at issue (for MA^{++}, expressions are given in Section 3, while free names are defined by a standard extension of the definition in [3]).

The first block of clauses are standard (they are the rules of the π-calculus). The rule (Struct Path Assoc) is a structural equivalence rule for the Ambient

$(\nu x)\mathbf{0} \equiv \mathbf{0}$	(Struct Res Dead)
$x \neq y \Rightarrow (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	(Struct Res Res)
$x \notin fn(P) \Rightarrow (\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$	(Struct Res Par)
$(A.A').P \equiv A.A'.P$	(Struct Path Assoc)
$x \notin fn(M) \cup \{a\} \Rightarrow (\nu x)a[M\,;\,P] \equiv a[M\,;\,(\nu x)P]$	(Struct Res Agent)
$\varepsilon.P \equiv P$	(Struct Empty Path)
$M \leftrightharpoons N \Rightarrow a[M\,;\,P] \equiv a[N\,;\,P]$	(Struct Cong Agent Meth)

Fig. 3. Structural Congruence for Agents

Calculus, while the rule (Struct Res Agent) modifies the rule for agents in the Ambient and Seal calculi to account for the presence of methods. The last rule establishes agent equivalence up to reordering of method suites. In addition, we identify processes up to renaming of bound variables: $(\nu p)P = (\nu q)P\{p := q\}$ if $q \notin fn(P)$. The behavior of agents is now defined in terms of a reduction relation which obeys the structural rules in Figure 4, plus specific rules for each of the construct in the calculus.

(**Red Struct**)	$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \quad \Rightarrow \quad P' \rightarrow Q'$
(**Red Amb**)	$P \rightarrow Q \quad \Rightarrow \quad a[M\,;\,P] \rightarrow a[M\,;\,Q]$
(**Red Res**)	$P \rightarrow Q \quad \Rightarrow \quad (\nu x)P \rightarrow (\nu x)Q$
(**Red Par**)	$P \rightarrow Q \quad \Rightarrow \quad P \mid R \rightarrow Q \mid R$

Fig. 4. Structural Rules for Reduction

2.3 Messages and Method Invocation

The semantics of method invocation is based on the idea of *self-substitution* distinctive of the Object Calculi of [1]: since agents are named, what gets substituted for the *self* variable is the *name* of the agent rather than the agent itself. Several choices can then be made as to (i) where the method invocation should occur, and (ii) where the body of the invoked method should be executed. Below, we discuss two possible modes for invocation and, within each mode we illustrate two possible locations where the invoked process can be activated.

Remote Invocation. This mode arises in a situation where the sender and the receiver of the message are siblings, and the message is sent by the process enclosed within the sender. Remote invocation is consistent with the "subjective" model of mobility, in which the moves of an object are regulated from within the object, by its controlling process: similarly, the object delegates to its controlling process the ability to communicate with its siblings.

Once the message is sent, the selected method can be activate either on the sender or on the receiver. There is a close analogy between these two possibilities, and two common protocols in distributed systems. Activating the method in the sender corresponds to the protocol known as *code on demand*.

(Code on Demand)
$$a[M\,;\, b\ \texttt{send}\ m(\boldsymbol{A}).P] \mid b[N, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R] \\ \rightarrow\ a[M\,;\, Q\{z := b, \boldsymbol{x} := \boldsymbol{A}\} \mid P] \mid b[N, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R]$$

Having requested a service, the sender takes the load of executing the corresponding process: this is a common practice for modern distributed systems, and specifically, for the Web. Upon receiving a request, a server spawns a new process (e.g., a Java *applet*) authorizing the client to execute it: the process is activated on the client side to not overburden the server with computations loads pertaining to its clients.

The alternative is to activate the method body on the receiver side, thus mimicking a *remote procedure call* (or more precisely a *remote method invocation*).

(Remote Procedure Call)
$$a[M\,;\, P \mid b\ \texttt{send}\ m(\boldsymbol{A}).P'] \mid b[M', m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R] \\ \rightarrow\ a[M\,;\, P \mid P'] \mid b[M', m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, Q\{z := b, \boldsymbol{x} := \boldsymbol{A}\} \mid R]$$

This alternative is just as reasonable. There is, however, a technical argument in favor of our first solution: activating the method body within the receiver makes it difficult to give a uniform reduction for messages to siblings and messages to *self*. Consider the following example:

$$a[M\,;\, b\ \texttt{send}\ m_1.P] \mid b[m_1 \triangleright \varsigma(z)z\ \texttt{send}\ m_2, m_2 \triangleright Q\,;\,]$$

If the method body is activated on the sender, the configuration reduces to $a[M\,;\, P \mid Q]$, as expected, in two steps. If instead, the body of m_1 is activated on the receiver, then one reduction leads to the configuration

$$a[M\,;\, P] \mid b[m_1 \triangleright \varsigma(z)z\ \texttt{send}\ m_2, m_2 \triangleright Q\,;\, b\ \texttt{send}\ m_2].$$

At this stage is no sibling object b to which the message m_2 may be sent. Although the problem may easily be remedied using special syntax (and reduction) for local method calls, our first solution is formally simpler and more elegant.

Local Invocation In this mode, the message is sent from an agent to one of its children: the invocation is *local* in that it does not extrude the scope of the sender. This mode is consistent with the "objective" model of mobility, in which the moves of an object are regulated by the enclosing environment: similarly, the environment requests the services provided by its enclosed objects.

The activation of the selected method may take place either at the same level as the invocation, or in the receiver. In the first case, the reduction is

$$b\ \texttt{send}\ m(\boldsymbol{A}).P \mid b[M; m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R] \\ \rightarrow\ P \mid Q\{z := b, \boldsymbol{x} := \boldsymbol{A}\} \mid b[M; m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, R]$$

Having requested a service to a child, the environment takes the load of executing

the corresponding process. As for the case of remote invocation, the method body could be activated within the receiver, but again the definition of reduction would not be uniform for messages to children and messages to *self*.

This concludes the discussion on the general model. In the next section we illustrate an instance of the model, based on Mobile Ambients [3,5].

3 MA^{++}

As we anticipated, our mobile objects combine the functionalities of Mobile Ambients and objects. We will use the terms object and ambient interchangeably to refer to agents in MA^{++}: the context will prevent any source of confusion between our agents and agents from MA.

Processes and methods are defined as in Figure 1, while the (now complete) syntax of expressions is given by the productions below:

$$A ::= a, \ldots, x \ldots \mid a\ \texttt{send}\ m(\boldsymbol{A}) \mid \varepsilon \mid A.A \mid \texttt{in}\ a \mid \texttt{out}\ a \mid \texttt{open}\ a$$

The `in` and `out` expressions provide objects with the same mobility capabilities as ambients, while `open` allows an object to break through the boundary of its enclosed objects and incorporate their body. As anticipated, the calculus does not include any construct or resource for synchronization or input/output among processes: objects only communicate via messages, and message sends are synchronous. This makes MA^{++} a superset of the combinatorial core of Mobile Ambients. Instead, Mobile Ambients and MA^{++} differ in their communication primitives. However, as we shall illustrate process communication can be encoded in terms of the existing constructs of MA^{++}.

3.1 Reduction Semantics

The definition of structural congruence and structural reduction are inherited directly from the general model. In Figure 5, we give the reduction rules distinctive of ambients in MA^{++}.

(in)	$a[M\,;\,\texttt{in}\ \ b.P \mid Q] \mid b[N\,;\,R] \ \rightarrow\ b[N\,;\,R \mid a[M\,;\,P \mid Q]]$
(out)	$a[M\,;\,b[N\,;\,\texttt{out}\ \ a.P \mid Q] \mid R] \ \rightarrow\ b[N\,;\,P \mid Q] \mid a[M\,;\,R]$
(open$_1$)	$\texttt{open}\ \ a.P \mid a[Q] \ \rightarrow\ P \mid Q$
(open$_2$)	$b[M\,;\,\texttt{open}\ \ a.P \mid a[N\,;\,Q] \mid R] \ \rightarrow\ b[M,N\,;\,P \mid Q \mid R]$ for $N \neq \varepsilon$
(send)	$a[M\,;\,P \mid b\ \texttt{send}\ m(\boldsymbol{A}).R] \mid b[N, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\,S]$ $\rightarrow\ a[M\,;\,P \mid Q\{z, \boldsymbol{x} := b, \boldsymbol{A}\} \mid R] \mid b[N, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\,S]$

Fig. 5. Reduction Rules for Ambients

The reduction rule **(send)** implements the remote mode for code-on-demand model we discussed in the previous section. The reduction rules for the `in` and `out` actions are defined exactly as in the Ambient Calculus. The action `in` $a.P$ instructs the ambient surrounding `in` $a.P$ to enter a sibling ambient named a. If no sibling named a exists, the operation blocks. The action `out` $a.P$ instructs the ambient surrounding `out` $a.P$ to exit its parent ambient named a: if the parent is not named a, the operation blocks until a time when such a parent exists.

The reduction for `open` depends on whether the method suite of the opened ambient is empty or not. If it is empty, the reduction is exactly the same as in the Ambient Calculus: `open` a dissolves the boundary of an ambient named a located at the same level as `open`, unleashing the process enclosed in a. If instead a contains a nonempty method suite, `open` a may only be reduced within an enclosing ambient and its effect is twofold: besides unleashing the process contained in a, it also merges the method suites of the opening and opened ambients. In both cases, if no ambient named a exists, `open` a blocks. As we show below, this behavior of open allows an elegant encoding of the operations of method update available in object calculi.

4 Examples

Parent-Child Messages. Having chosen the remote mode for method invocation, it is often useful for an ambient to be able to send messages to its parent or its children. We denote the two forms of communication as follows:

$a^{\downarrow}$ `send` $m(\boldsymbol{A}).P$ send $m(\boldsymbol{A})$ to child a

$a^{\uparrow}$ `send` $m(\boldsymbol{A}).P$ send $m(\boldsymbol{A})$ to parent a

Both these invocation modes can be derived with the existing constructs. Parent-to-child invocation can be defined as follows:

$$a^{\downarrow}\ \texttt{send}\ m(\boldsymbol{A}).P \triangleq (\nu p, q)(p[a\ \texttt{send}\ m(\boldsymbol{A}).q[\texttt{out}\ p]] \mid \texttt{open}\ q.\texttt{open}\ p.P)$$

where $p, q \notin fn(\boldsymbol{A}) \cup fn(P)$. Then, it is easy to verify that:

$$a^{\downarrow}\,\texttt{send}\,m(\boldsymbol{A}).P \mid a[m(\boldsymbol{x})\bowtie\varsigma(z)Q;\, R] \ \rightarrow^*\ P \mid Q\{z{,}\boldsymbol{x} := a{,}\boldsymbol{A}\} \mid a[m(\boldsymbol{x})\bowtie\varsigma(z)Q;\, R]$$

Child-to-parent invocation can be defined similarly, as follows:

$$a^{\uparrow}\,\texttt{send}\ m(\boldsymbol{A}).P \triangleq (\nu p, q)(\texttt{open}\ q.\texttt{open}\ p \mid p[\texttt{out}\ a.a\ \texttt{send}\ m(\boldsymbol{A}).\texttt{in}\ a.q[\texttt{out}\ p.P]])$$

where $p, q \notin fn(\boldsymbol{A}) \cup fn(P)$. Then $a[M, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, a^{\uparrow}\ \texttt{send}\ m(\boldsymbol{A}).P \mid R]$ reduces after some steps to $(\nu p, q)(a[M, m(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;\, Q\{z, \boldsymbol{x} := a, \boldsymbol{A}\} \mid P \mid R])$ as expected.

Replication. The behavior of replication in concurrent calculi is typically defined by a structural equivalence rule establishing that $!P \equiv !P \mid P$. With ambients, we can encode a similar construct relying upon the implicit form of recursion inherent in the reduction of method invocation. The coding is as follows:

$$!P \triangleq (\nu p)(p[bang \triangleright \varsigma(z)z^{\downarrow}\ \texttt{send}\ bang \mid P\,;\,] \mid p^{\downarrow}\ \texttt{send}\ bang)$$

where $p \notin fn(P)$. Using the derived reduction rule for downward method invocation, for every P, $!P$ reduces in one (encoded) step to $P \mid !P$ as desired. Similarly we can encode guarded replication $!A.P$ –where replication is performed only after the consumption of A– as follows:

$$(\nu p)(A.p^{\downarrow}\ \texttt{send}\ bang \mid p[bang \triangleright \varsigma(z)A.z^{\downarrow}\ \texttt{send}\ bang \mid P\,;\,])$$

Method update. Following the standard definition of method override [1,8] method updates for ambients can be formulated, informally, as follows: given the ambient $a[M; m(\boldsymbol{x}) \triangleright P\,;\, Q]$ we wish to replace the current definition P of $m(\boldsymbol{x})$ by the new definition P' to form the ambient $a[M; m(\boldsymbol{x}) \triangleright P'\,;\, Q]$.

Updates can be coded using a distinguished ambient as "updater". The updater carries the new method body and enters the updatable ambient a, while the updatable ambient is coded as an ambient whose controlling process opens the updater thus allowing updates on its own methods. The coding is defined precisely below. We give two different versions: in the first we have a form of concurrent update, where updates are processes; in the second, updates are "sequential" and coded as expressions.

Updates as processes: Update processes are denoted by $a \cdot m(\boldsymbol{y}) \triangleright \varsigma(s)P$, read "the m method at a gets definition P ". We define their behavior as follows: let first

$$a \cdot m(\boldsymbol{x}) \triangleright \varsigma(s)P \ \stackrel{\triangle}{=}\ \text{UPD}[m(\boldsymbol{x}) \triangleright \varsigma(s)P\,;\, \texttt{in}\ \ a].$$

Then define an updatable ambient as follows

$$a^{\star}[M\,;\,P] \ \stackrel{\triangle}{=}\ a[M\,;\,!(\texttt{open}\ \ \text{UPD}) \mid P].$$

Now, if we form the composition $a \cdot m(\boldsymbol{x}) \triangleright \varsigma(z)P' \mid a^{\star}[M, m(\boldsymbol{x}) \triangleright P\,;\, Q]$, the reduction for `open` enforces the expected behavior:

$$a \cdot m(\boldsymbol{x}) \triangleright P' \mid a^{\star}[M, m(\boldsymbol{x}) \triangleright P\,;\, Q] \quad \rightarrow^{*} \quad a^{\star}[M, m(\boldsymbol{x}) \triangleright P'\,;\, Q]$$

Multiple updates for the same method may occur in parallel, in which case their relative order is established nondeterministically. The coding works well also with "self inflicted" updates: for example, the configuration

$$a^{\downarrow}\ \texttt{send}\ m.P \mid a^{\star}[m \triangleright \varsigma(z)z \cdot n(\boldsymbol{x}) \triangleright Q', n(\boldsymbol{x}) \triangleright Q\,;\, R]$$

reduces to

$$P \mid a^{\star}[m \triangleright \varsigma(z)z \cdot n(\boldsymbol{x}) \triangleright Q', n(\boldsymbol{x}) \triangleright Q'\,;\, R]$$

as expected. With an appropriate use of restrictions it is possible to establish update permissions: for example, the ambient $(\nu \text{UPD})a[M\,;\,P \mid !\texttt{open}\ \ \text{UPD}]$ allows only self-inflicted updates.

Updates as expressions: Sequential updates are defined similarly to update processes. In this case, $(a \cdot m(\boldsymbol{x}) \triangleright \varsigma(s)P).Q$ first updates m at a and then continues as Q. This behavior can be accounted for by instrumenting the encoding we just described with a "locking" mechanism that blocks Q until the update is completed. An example of how this locking can be implemented is described below:

$$(a \cdot m(\boldsymbol{y}) \triangleright \varsigma(z)P).Q \triangleq (\nu p)(\text{UPD}[m(\boldsymbol{y}) \triangleright \varsigma(z)P\,;\ \texttt{in}\ \ a.p[\,;\ \texttt{out}\ a.Q]] \mid \texttt{open}\ p)$$

where $p \notin fn(P \mid Q)$. Then we have:

$$(a \cdot m(\boldsymbol{x}) \triangleright P').Q \mid a^\star[M, m(\boldsymbol{x}) \triangleright P\,;\ R] \quad \rightarrow^* \quad Q \mid a^\star[M, m(\boldsymbol{x}) \triangleright P'\,;\ R]$$

Process communication. The next example shows that synchronous and asynchronous communication primitives between processes can be encoded. We first give an encoding of synchronous communication. A similar model of (asynchronous) channel-based communication is presented in [5] and it is based on the more primitive form of local and *anonymous* communication defined for the Ambient Calculus: here, instead, we rely on the ability, distinctive of our ambients, to exchange values between methods.

A channel n is modeled by a (parallel composition of) an updatable ambient n, and two locks n^i, and n^o. The ambient n contains a method *msg*: a process willing to read from n installs itself as the body of this method, whereas a process willing to write on n invokes *msg* passing along the argument of the communication.

$$\begin{aligned}(ch\ n) &\triangleq n^\star[msg(x) \triangleright \mathbf{0}] \mid n^i[\,]\\ n\langle A\rangle.Q &\triangleq \texttt{open}\ n^o.n^\downarrow\ \texttt{send}\ msg(A).(n^i[\,] \mid Q)\\ n(x).P &\triangleq \texttt{open}\ n^i.n \cdot msg(x) \triangleright \varsigma(z)P.n^o[\,] \qquad (z \notin fn(P))\end{aligned}$$

The communication protocol is as follows: A process $n\langle A\rangle.Q$ writing A on n first attempts to grab the output lock n^o, then sends the message $msg(A)$ to n, and finally continues as Q releasing the input lock n^i. At the start of the protocol there are no output locks: hence the process writing on n blocks. A process $n(x).P$ reading from n first grabs the input lock n^i provided by the channel, then installs itself as the body of the *msg* method in n, and finally releases the output lock. Now the writing process resumes its computation: it sends the message thus unleashing P, and then releases the input lock and continues as Q.

Asynchronous communications are obtained directly from the coding above, by a slight variation of the definition of $n\langle A\rangle.Q$. We simply need a different parenthesizing:

$$n\langle A\rangle.Q \triangleq (\texttt{open}\ n^o.n^\downarrow\ \texttt{send}\ msg(A).(n^i[])) \mid Q$$

Based on the this technique, we can encode the synchronous (and similarly, the asynchronous) π-calculus in ways similar to what is done in [6]. Each name n in the π-calculus becomes a triple of names in our calculus: the name n of the ambient dedicated to the communication, and the names n^i and n^o of the two locks. Therefore, communication of a π-calculus name becomes the communication of a triple of ambient names.

The initialization of the *msg* method in the ambient that encodes the channel n could be safely omitted, without affecting the operational properties of encoding. However, as given, the definition scales smoothly to the case of a typed encoding, preserving well-typing.

$$
\begin{array}{rcl}
[\![(\nu n)P]\!] & \triangleq & (\nu n, n^i, n^o)(n^i[\,] \mid n^\star[msg(x, x^i, x^o) \triangleright \mathbf{0}] \mid [\![P]\!]) \quad n^i, n^o \notin fn([\![P]\!]) \\
[\![n\langle y\rangle.Q]\!] & \triangleq & \texttt{open}\ n^o.n^\downarrow\ \texttt{send}\ msg(y, y^i, y^o).(n^i[\,] \mid Q) \\
[\![n(x).P]\!] & \triangleq & \texttt{open}\ n^i.n \cdot msg(x, x^i, x^o) \triangleright \varsigma(z)P.n^o[\,] \\
[\![P \mid Q]\!] & \triangleq & [\![P]\!] \mid [\![Q]\!] \\
[\![!P]\!] & \triangleq & (\nu n)(n[bang \triangleright \varsigma(z)z^\downarrow\ \texttt{send}\ bang \mid [\![P]\!]\ ;\,] \mid n^\downarrow\ \texttt{send}\ bang) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n \notin fn([\![P]\!])
\end{array}
$$

Fig. 6. Encoding of the synchronous π-calculus

5 Types and Type Systems

The typing of ambients inherits ideas from existing type systems for Mobile Ambients: however, as we anticipated, the presence of methods enables a more structured (and informative) characterization of their enclosing ambient's interfaces. The productions defining the set of types are given below:

Signatures	$\Sigma ::=$	$m(\mathscr{W}) \triangleright \mathscr{P}, \Sigma \quad \mid \quad \varepsilon$
Ambients	$\mathscr{A} ::=$	$\texttt{Amb}[\Sigma]$
Capabilities	$\mathscr{C} ::=$	$\texttt{Cap}[\Sigma]$
Processes	$\mathscr{P} ::=$	$\texttt{Proc}[\Sigma]$
Values	$\mathscr{W} ::=$	$\mathscr{A} \mid \mathscr{C}$

Signatures convey information about the interface of an ambient, by listing the ambient's method names, input type as well as the type of the method bodies. The intuitive reading of ambient, capability and process types is as follows: the type $\texttt{Amb}[\Sigma]$ is the type of ambients with methods declared in Σ; the type $\texttt{Cap}[\Sigma]$ is the type of capabilities[1] whose enclosing ambient (if any) has a signature which contains at least the methods included in Σ; the type $\texttt{Proc}[\Sigma]$ is the type of processes whose enclosing ambient (if there is any) contains at least all the methods declared in Σ.

The essential novelty over previous type systems for Mobile Ambients [6,4,13] is that our ambient and capability types are associated to method signatures: in [6], instead, ambient (and capability) types expose the type of values that can be exchanged as a result of local process communication. This difference reflects the different communication primitives in the two calculi: specifically, communication is accomplished via message sends in our calculus, whereas it relies on explicit input/output primitives in MA.

[1] *Capability* is the term used in [5] to refer to "actions": capabilities can be transmitted over channels, and transmitting a capability corresponds to transmit the *capability* of performing the corresponding action. The same intuition justifies the use of the term in MA^{++}.

5.1 Type System

The typed syntax of the calculus is defined by the following productions[2] :

Methods	$M ::= m(x{:}\mathscr{W}) \triangleright \varsigma(z{:}\mathscr{A})P \mid M, M \mid \varepsilon$
Processes	$P ::= \mathbf{0} \mid P \mid P \mid a[M\,;\,P] \mid (\nu a{:}\mathscr{A})P \mid A.P$
Expressions	$A ::= a, x \mid a\ \mathtt{send}\ m(\boldsymbol{A}) \mid \mathtt{in}\ a \mid \mathtt{out}\ a \mid \mathtt{open}\ a \mid A.A \mid \varepsilon$

The type system derives three kinds of judgments (where E denotes generic expressions and processes): $\Gamma \vdash \diamond$ (well-formed type environment), $\Gamma \vdash E : T$ (typing), and $\Gamma \vdash T_1 \leq T_2$ (subtyping). The typing and subtyping rules presented in Figure 7 are discussed below.

Method signatures, associated with ambient types, are traced by the types `Cap`, of capabilities, to allow an adequate typing of messages and mobility: specifically, the rule (OPEN) establishes that opening an ambient $a : \mathtt{Amb}[\Sigma]$ is legal under the condition that the signature of the opening ambient is equal to (in fact, contains, given the presence of subtyping) the signature of the ambient being opened. This condition is necessary, as subject reduction would otherwise fail: as a consequence, opening an ambient may only update existing methods of the opening ambient, and the update must preserve the types of the original methods.

Signatures are not traced when typing expressions involving moves or messages: for the latter, see rule (MESSAGE), the capability type has the same signature as the process type of the body of the invoked method. Of course, in order for the expression to type check the message argument and the method parameters must have the same type[3].

The typing of processes is standard (cf. [6,13]), with the only exception of the rule (AMB) which defines the types of ambients. Ambients are typed similarly to objects in the object calculi of [1]: each method is typed under the assumptions that (i) the self parameter has the same type of the enclosing ambient, and (ii) that method parameters have the declared type. The condition $j \in \mathrm{LAST}(I)$ (where $\mathrm{LAST}(I)$ denotes the set $\{i \in I \mid \forall j > i, m_j \neq m_i\}$) ensures that only the rightmost definition of a method is considered when typing an ambient[4]. Finally, no constraint is imposed on the signature Σ', associated with the process type in the conclusion of the rule, as that signature is (a subset of) the signature of the ambient enclosing a (if any). As for the subtyping relation, non-trivial subtyping is defined for capability and process types: specifically, a capability (resp. process) type $\mathtt{Cap}[\Sigma]$ (resp. $\mathtt{Proc}[\Sigma]$) is a subtype of any capability (resp. process) type whose associated signature (set theoretically) contains Σ. The

[2] The other typed versions Ambients [6,4,13] allow restrictions on variables of type $\mathscr{W}$ (rather than just $\mathscr{A}$), but we do not see the purpose of such a generalization.

[3] In fact, since capability and ambient types can be subtyped, the type of the arguments can be subtypes of the type of the formal parameters.

[4] Technically, we need this restriction to ensure the subject reduction property: without it, a well-typed term could be structurally equivalent (and, therefore, reduction equivalent) to an ill-typed one.

Type environments

$$\frac{}{\varnothing \vdash \diamond}\ (\textsc{Env-empty}) \qquad \frac{\Gamma \vdash \diamond \quad x \notin Dom(\Gamma)}{\Gamma, x : \mathscr{W} \vdash \diamond}\ (\textsc{Env-name})$$

Expressions

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)}\ (\textsc{name/var}) \qquad \frac{\Gamma \vdash A_1 : \mathtt{Cap}[\Sigma] \quad \Gamma \vdash A_2 : \mathtt{Cap}[\Sigma]}{\Gamma \vdash A_1.A_2 : \mathtt{Cap}[\Sigma]}\ (\textsc{path})$$

$$\frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma]}{\Gamma \vdash \mathtt{open}\ a : \mathtt{Cap}[\Sigma]}\ (\textsc{open}) \qquad \frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma] \quad (A' \in \{\mathtt{in}\ a, \mathtt{out}\ a\})}{\Gamma \vdash A' : \mathtt{Cap}[\Sigma']}\ (\textsc{inout})$$

$$\frac{\Gamma \vdash a : \mathtt{Amb}[\Sigma] \quad \Gamma \vdash A' : \mathscr{W} \quad (m(\mathscr{W}) \triangleright \mathtt{Proc}[\Sigma'] \in \Sigma)}{\Gamma \vdash a\ \mathtt{send}\ m(A') : \mathtt{Cap}[\Sigma']}\ (\textsc{Message})$$

Processes

$$\frac{\Gamma \vdash A : \mathtt{Cap}[\Sigma] \quad \Gamma \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash A.P : \mathtt{Proc}[\Sigma]}\ (\textsc{pref}) \qquad \frac{\Gamma \vdash P : \mathtt{Proc}[\Sigma] \quad \Gamma \vdash Q : \mathtt{Proc}[\Sigma]}{\Gamma \vdash P \mid Q : \mathtt{Proc}[\Sigma]}\ (\textsc{par})$$

$$\frac{\Gamma, a : \mathscr{A} \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash (\nu a : \mathscr{A})P : \mathtt{Proc}[\Sigma]}\ (\textsc{restr}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \mathtt{Proc}[\Sigma]}\ (\textsc{dead})$$

$$\frac{\Gamma \vdash a : \mathscr{A} \qquad \Gamma, z : \mathscr{A}, x_j : \mathscr{W}_j \vdash P_j : \mathtt{Proc}[\Sigma_j] \qquad \Gamma \vdash P : \mathtt{Proc}[\Sigma]}{\Gamma \vdash a[(m_i(x_i : \mathscr{W}_i) \triangleright \varsigma(z : \mathscr{A})P_i)_{i \in I}\ ;\ P] : \mathtt{Proc}[\Sigma']}\ (\textsc{Amb}) \quad (\Sigma = (m_j(\mathscr{W}_j) \triangleright \mathtt{Proc}[\Sigma_j])_{j \in \textsc{last}(I)},\ \mathscr{A} = \mathtt{Amb}[\Sigma],\ j \in \textsc{last}(I))$$

Subsumption

$$\frac{\Gamma \vdash E : \mathscr{W} \qquad \mathscr{W} \leq \mathscr{W}'}{\Gamma \vdash E : \mathscr{W}'}\ (\textsc{subs})$$

Subtyping

$$\frac{\Sigma \subseteq \Sigma'}{\mathtt{Cap}[\Sigma] \leq \mathtt{Cap}[\Sigma']}\ (\textsc{Subcap}) \qquad \frac{\Sigma \subseteq \Sigma'}{\mathtt{Proc}[\Sigma] \leq \mathtt{Proc}[\Sigma']}\ (\textsc{Subproc})$$

Fig. 7. Typing and Subtyping Rules

resulting notion of subtyping corresponds to the contravariant subtyping in *width* distinctive of variant types. The covariant width subtyping typical of object and record types must be disallowed over ambient types to ensure sound uses of the `open` capability: intuitively, when opening an enclosed ambient, we need *exact* knowledge of the contents of that ambient, (specifically, of its method suite) so as to ensure that all the overriding that takes place upon exercising the capability be traced in the types.

As customary, the subtyping relation is endowed in the type system via a subsumption rule.

5.2 Subject Reduction and Type Soundness

We conclude the description of the type system with a subject reduction theorem and a discussion on type soundness. The, rather standard, proof is only outlined here due to the lack of space.

Lemma 1 (Substitution). *Assume* $\Gamma, x{:}\mathscr{W} \vdash P : \mathtt{Proc}[\Sigma]$ *and* $\Gamma \vdash A{:}\mathscr{W}$. *Then* $\Gamma \vdash P\{x := A\} : \mathtt{Proc}[\Sigma']$ *with* $\Sigma' \subseteq \Sigma$.

Proposition 1 (Subject Congruence).
1. If $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $P \equiv Q$ *then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma]$.
2. If $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $Q \equiv P$ *then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma]$.

Theorem 1 (Subject Reduction). *Assume* $\Gamma \vdash P : \mathtt{Proc}[\Sigma]$ *and* $P \rightarrow Q$. *Then* $\Gamma \vdash Q : \mathtt{Proc}[\Sigma']$ *with* $\Sigma' \subseteq \Sigma$.

Besides being interesting as a meta-theoretical property of the type system, subject reduction may be used to derive a soundness theorem ensuring the absence of run-time (type) errors for well-typed programs. As we anticipated, the errors we wish to detect are those of the kind "message not understood" distinctive of object calculi. With the current definition of the reduction relation such errors do not arise, as not-understood messages simply block: this is somewhat unrealistic, however, as the result of sending a message to an object (a server) which does not contain a corresponding method should be (and indeed is, in real systems) reported as an error. We thus introduce a new reduction to account for these situations:

$$a[M\,;\,P \mid b\ \mathtt{send}\ m(\boldsymbol{A}).Q] \mid b[N\,;\,R] \rightarrow a[M\,;\,P \mid \mathtt{ERR}] \mid b[N\,;\,R] \qquad (m \notin N)$$

The intuitive reading of the reduction is that a not-understood message causes a local error –for the sender of that message– rather than a global error for the entire system. The reduction is meaningful also in the presence of multiple ambients with equal name, as our type system (like those of [6,4,13]) ensures that ambients with the same name have also the same type. Hence, if a method m is absent from a given ambient b, it will also be absent from all ambients named b. If we take `ERR` to be a distinguished process, with no type, it is easy to verify that no system containing an occurrence of `ERR` can be typed in our type system. Absence of run-time errors may now be stated follows:

Theorem 2. *Let P be a well-typed MA^{++} process. Then, there exist no context* $\mathbf{C}[-]$ *such that* $P \rightarrow^* \mathbf{C}[\mathtt{ERR}]$.

6 Extensions

There are several extensions desirable both for MA^{++} and for its type system. The most natural is the ability to treat method names as ordinary names. This would allow one to define private methods, and to give a formal account of dynamic messages. Both the extensions can be accommodated for free in the untyped calculus. For the typed version, instead, the extension is subtler. It is possible (and relatively easy) to extend the syntax and allow method names to be restricted. Instead, allowing method names as values is more critical. The reason is that method names occur in the signatures of ambient (capability and process) types: consequently, allowing methods to be passed would be possible but it would make our types (first-order) dependent types (see [10] for similar restrictions).

A further extension has to do with Safe Ambients. In [13] the authors describe an extension of the calculus of Mobile Ambients, called Safe Ambients, where entering, exiting and opening an ambient requires a corresponding co-action by the ambient that undergoes the action. The use of co-actions allows (*i*) a more fine-grained control on when actions take place, and (*ii*) the definition of a refined type system where types can be used to essentially "serialize" the activities of the parallel processes controlling the moves of an ambient. As shown in [13], the combination of these features makes it possible to define a rich algebraic theory for the resulting calculus. The idea of co-actions and of single-threaded types can be incorporated in the type system we have described in the previous sections rather smoothly: besides the co-actions related to mobility, we simply need a co-action for messages, and a modified reduction rule for message sends that requires the receiver to be *listening* (i.e. to exercise the co-action corresponding to `send`) in order to reduce the message. We leave this as subject of future work.

Finally, it would be interesting to include linear types to ensure (local) absence of ambients with the same name: in fact, while the possibility of there being more than one ambient that is willing to receive a given message provides useful forms of nondeterminism, ensuring linearity of ambient names could be useful to prevent what [13] defines "grave interferences" and thus to prove interesting behavioral properties of method invocation.

7 Conclusions

One of the main purposes, as well as of the challenges, for a foundational formalism for distributed systems is to establish an adequate setting where formal proofs of behavioral properties for processes and agents can be carried out.

Viewed from this perspective, the work on MA^{++} should be understood as a first step to define a computation model for distributed applications, where

conventional technology –based on remote exchange of messages between static sites– and mobile agents coexist and can be integrated in a uniform way. This attempt appears to be well motivated by the current –rather intense– debate on the role of mobility in wide-area distributed applications; a debate in which even proponents and developers of mobile agents offer that "we probably shouldn't expect purely mobile applications to replace other structuring techniques" [11].

More work is clearly needed to evaluate the adequacy of the calculus as a formal tool for modeling realistic applications, to develop a reasonable algebraic theory for the calculus, and to study techniques of program static analysis other than typing.

All these aspects are current topic of research for the two calculi –Ambients and Seals– we had in mind when developing the general model. The recent papers on typed formulations of Ambients [6,4,13] provide rather interesting and useful insight into how an algebraic theory for mobile objects could be defined as well as into the rôle of types in proving behavioral properties.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996. 506, 507, 508, 512, 515
2. M. Bugliesi and G. Castagna. Mobile objects. In *FOOL'7 Proc. of the 7th Int. Workshop on Foundations of Object Oriented Languages*. 2000. Electronic Proceedings. 504
3. L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in LNCS, pages 51–94. Springer, 1999. 505, 507, 510
4. L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, number 1644 in LNCS, pages 230–239. Springer, 1999. 514, 515, 517, 519
5. L. Cardelli and A. Gordon. Mobile ambients. In *POPL'98*. ACM Press, 1998. 505, 510, 513, 514
6. L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999. 513, 514, 515, 517, 519
7. P Di Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR'96*, number 1119 in LNCS, pages 655–670. Springer, 1996. 505
8. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. 507, 512
9. A. Gordon and P. D Hankin. A concurrent object calculus: reduction and typing. *In Proceedings HLCL'98, Elsevier ENTC*, 1998. Also Technical Report 457, University of Cambridge Computer Laboratory, February 1999. 504, 505
10. Ms Hennessy and J. Riely. Resource access control in systems of mobile agents (extended abstract). In *Proc. of 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*. 1998. 518
11. D. Johansen. Trend wars. *IEEE Concurrency*, 7(3), Sept 1999. 519
12. J. Kleist and Sangiorgi D. Imperative objects and mobile processes. Unpublished manuscript. 505
13. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'2000*, pages 352–364. ACM Press, 2000. 505, 514, 515, 517, 518, 519

14. U Nestmann, H. Huttel, J. Kleist, and M. Merro. Aliasing models for object migration. In *Proceedings of Euro-Par'99*, number 1685 in LNCS, pages 1353–1368. Springer, 1999. 505
15. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming, Sendai, Japan (Nov. 1994)*, number 907 in LNCS, pages 187–215. Springer-Verlag, April 1995. 505
16. V. T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, number 821 in LNCS, pages 100–117. Springer, 1994. 505
17. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in LNCS. Springer, 1999. 505
18. D. J. Walker. Objects in the π calculus. *Information and Computation*, 116(2):253–271, 1995. 505

Synthesizing Distributed Finite-State Systems from MSCs*

Madhavan Mukund[1], K. Narayan Kumar[1], and Milind Sohoni[2]

[1] Chennai Mathematical Institute, Chennai, India.
{madhavan,kumar}@smi.ernet.in
[2] Indian Institute of Technology Bombay, Mumbai, India
sohoni@cse.iitb.ernet.in

Abstract. Message sequence charts (MSCs) are an appealing visual formalism often used to capture system requirements in the early stages of design. An important question concerning MSCs is the following: how does one convert requirements represented by MSCs into state-based specifications? A first step in this direction was the definition in [9] of *regular* collections of MSCs, together with a characterization of this class in terms of finite-state distributed devices called message-passing automata. These automata are, in general, nondeterministic. In this paper, we strengthen this connection and describe how to directly associate a *deterministic* message-passing automaton with each regular collection of MSCs. Since real life distributed protocols are deterministic, our result is a more comprehensive solution to the synthesis problem for MSCs. Our result can be viewed as an extension of Zielonka's theorem for Mazurkiewicz trace languages [6,19] to the setting of finite-state message-passing systems.

1 Introduction

Message sequence charts (MSCs) are an appealing visual formalism often used to capture system requirements in the early stages of design. They are particularly suited for describing scenarios for distributed telecommunication software [16]. They have also been called timing sequence diagrams, message flow diagrams and object interaction diagrams and are used in a number of software engineering methodologies [4,7,16]. In its basic form, an MSC depicts the exchange of messages between the processes of a distributed system along a single partially-ordered execution. A collection of MSCs is used to capture the scenarios that a designer might want the system to exhibit (or avoid).

Given the requirements in the form of a collection of MSCs, one can hope to do formal analysis and discover errors at an early stage. A standard way to generate a collection of MSCs is to use a High Level Message Sequence Chart

* This work has been supported in part by Project DRD/CSE/98-99/MS-4 between the Indian Institute of Technology Bombay and Ericsson (India), Project 2102-1 of the Indo-French Centre for Promotion of Advanced Research and NSF grant CDA9805735.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 521–535, 2000.

(HMSC) [12]. An HMSC is a finite directed graph in which each node is labelled, in turn, by an HMSC. The HMSCs labelling the vertices may not refer to each other. The collection of MSCs represented by an HMSC consists of all MSCs obtained by tracing a path in the HMSC from an initial vertex to a terminal vertex and concatenating the MSCs that are encountered along the path.

In order to analyze the collection of MSCs represented by an HMSC, one desirable property is that these MSCs correspond to the behaviour of a finite-state system. This property would be violated if the specification were to permit an unbounded number of messages to accumulate in a channel. A sufficient condition to rule out such *divergence* in an HMSC is described in [3]. Subsequently, it has been observed that HMSCs can also violate the finite-state property by exhibiting nonregular behaviour over causally independent bounded channels [2]. To remedy this, a stronger criterion is established in [2] which suffices to ensure that the behaviour described by an HMSC can be implemented by a (global) finite-state system. This leads to a more general question of when a collection of MSCs should be called regular. A robust notion of regularity has been proposed in [9]. As shown in [8], this notion strictly subsumes the finite-state collections generated by HMSCs. It turns out that the collections defined by HMSCs correspond to the class of finitely-generated regular collections of MSCs.

One of the main contributions of [9] is a characterization of regular collections of MSCs in terms of (distributed) finite-state devices called message-passing automata. This addresses the important synthesis problem for MSCs, first raised in [5]; namely, how to convert requirements as specified by MSCs into distributed, state-based specifications. The message-passing automata associated with regular collections of MSCs in [9] are, in general, nondeterministic. In this respect the solution to the synthesis problem in [9] is not completely satisfactory, since real life distributed protocols are normally deterministic.

In this paper, we strengthen the result of [9] by providing a technique for decomposing a sequential automaton accepting a regular collection of MSCs into a *deterministic* message-passing automaton. Our result can be viewed as the message-passing analogue of the celebrated theorem of Zielonka from Mazurkiewicz trace theory establishing that regular trace languages precisely correspond to the trace languages accepted by deterministic asynchronous automata [19].

In related work, a number of studies are available which are concerned with individual MSCs in terms of their semantics and properties [1,10]. A variety of algorithms have been developed for HMSCs in the literature—for instance, pattern matching [11,14,15] and detection of process divergence and non-local choice [3]. A systematic account of the various model-checking problems associated with HMSCs and their complexities is given in [2].

The paper is organized as follows. In the next section we introduce MSCs and regular MSC languages. In Section 3 we define message-passing automata and state the problem. Section 4 describes a time-stamping result for message-passing systems from [13] which is crucial for proving our main result. In Section 5 we then introduce the notion of residues and show how the ability to locally compute residues would solve the decomposition problem. The next section describes

a procedure for locally updating residues. This procedure is formalized as a message-passing automaton in Section 7.

2 Regular MSC Languages

Let $\mathcal{P} = \{p, q, r, \ldots\}$ be a finite set of processes which communicate with each other through messages. We assume that messages are never inserted, lost or modified—that is, the communication medium is reliable. However, there may be an arbitrary delay between the sending of a message and its receipt. We assume that messages are delivered in the order in which they are sent—in other words, the buffers between processes behave in a FIFO manner.

For each process $p \in \mathcal{P}$, we fix $\Sigma_p = \{p!q \mid p \neq q\} \cup \{p?q \mid p \neq q\}$ to be the set of communication actions in which p participates. The action $p!q$ is to be read as p *sends to* q and the action $p?q$ is to be read as p *receives from* q. We shall not be concerned with the actual messages that are sent and received—we are primarily interested in the pattern of communication between agents. We will also not deal with the internal actions of the agents. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$ and let a, b range over Σ.

For $a \in \Sigma$ and $u \in \Sigma^*$, $\#_a(u)$ denotes the number of occurrences of a in u. A word $u \in \Sigma^*$ is a *proper* communication sequence of the processes if for each prefix v of u and each pair of processes $p, q \in \mathcal{P}$, $\#_{p!q}(v) \geq \#_{q?p}(v)$—that is, at any point in the computation, at most as many messages have been received at q from p as have been sent from p to q. We say that u is a *complete* communication sequence if u is a proper communication sequence and for each pair of processes $p, q \in \mathcal{P}$, $\#_{p!q}(u) = \#_{q?p}(u)$—in other words, at the end of u, all messages that have been sent have also been received. We shall often say that u is proper (respectively, complete) to mean that u is a proper (respectively, complete) communication sequence over Σ.

Let $u = a_0 a_1 \ldots a_n \in \Sigma^*$ be proper. We can associate a natural Σ-labelled partial order $M_u = (E_u, \leq, \lambda)$ with u where:

- $E = \{(i, a_i) \mid i \in \{1, 2, \ldots, n\}$.
- $\lambda((i, a_i)) = a_i$. (If $\lambda(e) \in \Sigma_p$, we say that e is a p-event.)
- For $p, q \in \mathcal{P}$, we define relations $<_{pq} \subseteq E \times E$ as follows:
  - For $p \in \mathcal{P}$, $(i, a_i) <_{pp} (j, a_j)$ if $a_i, a_j \in \Sigma_p$, $i < j$ and there is no $i < k < j$ such that $a_k \in \Sigma_p$.
  - For $p, q \in \mathcal{P}$, $p \neq q$, $(i, a_i) <_{pq} (j, a_j)$ if $a_i = p!q$, $a_j = q?p$ and the sets $\{(k, a_k) \mid k < i, a_k = p!q\}$ and $\{(k, a_k) \mid k < j, a_k = q?p\}$ are of the same cardinality. Since messages are assumed to be read in FIFO fashion, $(i, a_i) <_{pq} (j, a_j)$ implies that the message read at the receive event (j, a_j) is the one sent at the send event (i, a_i).
- The partial order $\leq$ is the reflexive, transitive closure of the relations $\bigcup_{p,q \in \mathcal{P}} <_{pq}$.

We shall call the structure M_u generated from a *complete* communication sequence u a Message Sequence Chart (MSC).[1] The partial order between events in M_u is a more faithful representation of the causality between events in u than the sequential order induced by writing u as a string.

Henceforth, we shall implicitly associate with each proper word u the corresponding structure $M_u = (E_u, \leq, \lambda)$. In particular, E_u always refers to the set of events associated with the structure M_u generated from a proper word u. Abusing terminology, we refer to M_u as an MSC even if u is not complete.

Let $M_u = (E_u, \leq, \lambda)$ be an MSC. For $e \in E_u$, $e{\downarrow}$ denotes, as usual, the set $\{f \in E_u \mid f \leq e\}$. For $X \subseteq E_u$, $X{\downarrow}$ is defined to be $\bigcup_{e \in X} e{\downarrow}$.

We define a context-sensitive independence relation $I \subseteq \Sigma^* \times (\Sigma \times \Sigma)$ as follows: $(u, a, b) \in I$ provided that u is proper, $a \in \Sigma_p$ and $b \in \Sigma_q$ for distinct processes p and q, and, further, if $a = p!q$ and $b = q?p$ then $\#_a(u) > \#_b(u)$. Observe that if $(u, a, b) \in I$ then $(u, b, a) \in I$.

Let $\Sigma^\circ = \{u \in \Sigma^* \mid u \text{ is complete}\}$. We define $\sim \subseteq \Sigma^\circ \times \Sigma^\circ$ to be the least equivalence relation such that if $u = u_1abu_2$ and $u' = u_1bau_2$ and $(u_1, a, b) \in I$ then $u \sim u'$. It is important to note that $\sim$ is defined over Σ° (and not Σ^*).

The following simple observation shows that each MSC corresponds to a $\sim$-equivalence classes of complete communication sequences over Σ.

Proposition 2.1. *Let $u, v \in \Sigma^\circ$. Then, v is a linearization of M_u iff $u \sim v$.*

We define $L \subseteq \Sigma^*$ to be a *MSC language* if every member of L is complete and L is $\sim$-closed (that is, for each $u \in L$, if $u \in L$ and $u \sim v$ then $v \in L$.) We say that an MSC language L is a *regular* if L is a regular subset of Σ^*.

Given a regular subset $L \subseteq \Sigma^*$, we can decide whether L is a regular MSC language. We say that a state s in a finite-state automaton is *live* if there is a path from s to a final state. We then have the following result from [9].

Lemma 2.2. *Let $\mathcal{A} = (S, \Sigma, s_{in}, \delta, F)$ be the minimal DFA representing L. Let $Chan = \{(p, q) \mid p, q \in \mathcal{P}, p \neq q\}$ denote the set of* channels. *L is a regular MSC language iff we can associate with each live state $s \in S$, a channel-capacity function $\mathcal{K}_s : Chan \to \mathbb{N}$ which satisfies the following conditions.*

- *(i) If $s \in \{s_{in}\} \cup F$ then $\mathcal{K}_s(c) = 0$ for every $c \in Ch$.*
- *(ii) If s, s' are live states and $\delta(s, p!q) = s'$ then $\mathcal{K}_{s'}((p, q)) = \mathcal{K}_s((p, q))+1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p, q)$.*
- *(iii) If s, s' are live states $\delta(s, q?p) = s'$ then $\mathcal{K}_s((p, q)) > 0$, $\mathcal{K}_{s'}((p, q)) = \mathcal{K}_s((p, q))-1$ and $\mathcal{K}_{s'}(c) = \mathcal{K}_s(c)$ for every $c \neq (p, q)$.*
- *(iv) Suppose $\delta(s, a) = s_1$ and $\delta(s_1, b) = s_2$ with $a \in \Sigma_p$ and $b \in \Sigma_q$, $p \neq q$. If it is not the case that $a = p!q$ and $b = q?p$, or it is the case that $\mathcal{K}_s((p, q)) > 0$, there exists s'_1 such that $\delta(s, b) = s'_1$ and $\delta(s'_1, a) = s_2$.*

[1] Our definition captures the standard partial-order semantics associated with MSCs [1,16]. See [9] for an equivalent definition of MSCs in terms of labelled partial orders.

Observe that the conditions described in the lemma can be checked in time linear in the size of δ.

Item (iv) of the lemma has useful consequences. As usual, we extend δ to words and let $\delta(s_{in}, u)$ denote the (unique) state reached by $\mathcal{A}$ on reading an input u. Let u be a proper word and let a, b be communication actions such that (u, a, b) belongs to the context-sensitive independence relation defined earlier. Item (iv) guarantees that $\delta(s_{in}, uab) = \delta(s_{in}, uba)$. From this, we can conclude that if v, w are complete words such that $v \sim w$, then $\delta(s_{in}, v) = \delta(s_{in}, w)$.

3 Message-Passing Automata

We now define distributed automata which accept MSC languages.

Message-passing automaton A *message-passing automaton* over Σ is a structure $\mathcal{A} = (\{\mathcal{A}_p\}_{p \in \mathcal{P}}, \mathcal{M}, s_{in}, \mathcal{F})$ where

- $\mathcal{M}$ is a finite alphabet of messages.
- Each component $\mathcal{A}_p$ is of the form $(S_p, \rightarrow_p)$ where
  - S_p is a finite set of p-local states.
  - $\rightarrow_p \subseteq (S_p \times \Sigma_p \times \mathcal{M} \times S_p)$ is the p-local transition relation.
- $s_{in} \in \prod_{p \in \mathcal{P}} S_p$ is the global initial state.
- $\mathcal{F} \subseteq \prod_{p \in \mathcal{P}} S_p$ is the set of global final states.

The local transition relation $\rightarrow_p$ specifies how the process p sends and receives messages. The transition $(s, p!q, m, s')$ specifies that when p is in the state s, it can send the message m to q (by executing the communication action $p!q$) and go to the state s'. The message m is, as a result, appended to the queue of messages in the channel (p, q). Similarly, the transition $(s, p?q, m, s')$ signifies that at the state s, the process p can receive the message m from q by executing the action $p?q$ and go to the state s'. The message m is removed from the head of the queue of messages in the channel (q, p).

We say that $\mathcal{A}$ is *deterministic* if the local transition relation $\rightarrow_p$ for each component $\mathcal{A}_p$ satisfies the following conditions:

- $(s, p!q, m_1, s'_1) \in \rightarrow_p$ and $(s, p!q, m_2, s'_2) \in \rightarrow_p$ imply $m_1 = m_2$ and $s'_1 = s'_2$.
- $(s, p?q, m, s'_1) \in \rightarrow_p$ and $(s, p?q, m, s'_2) \in \rightarrow_p$ imply $s'_1 = s'_2$.

In other words, when a component $\mathcal{A}_p$ of a deterministic automaton $\mathcal{A}$ executes a send action, the current state of $\mathcal{A}_p$ uniquely determines the message sent as well as the new state of $\mathcal{A}_p$, and when $\mathcal{A}_p$ executes a receive action, the current state of $\mathcal{A}_p$ and the nature of the message at the head of the queue uniquely determine the new state of $\mathcal{A}_p$.

The set of global states of $\mathcal{A}$ is given by $\prod_{p \in \mathcal{P}} S_p$. For a global state s, we let s_p denote the pth component of s. A *configuration* is a pair (s, χ) where s is a global state and $\chi : \mathit{Chan} \rightarrow \mathcal{M}^*$ is the *channel state* which specifies the queue of messages currently residing in each channel c. The *initial configuration*

of $\mathcal{A}$ is $(s_{in}, \chi_\varepsilon)$ where $\chi_\varepsilon(c)$ is the empty string ε for every channel c. The set of *final configurations* of $\mathcal{A}$ is $\mathcal{F} \times \{\chi_\varepsilon\}$.

We now define the set of reachable configurations $Conf_{\mathcal{A}}$ and the global transition relation $\Rightarrow \subseteq Conf_{\mathcal{A}} \times \Sigma \times Conf_{\mathcal{A}}$ inductively as follows:

- $(s_{in}, \chi_\varepsilon) \in Conf_{\mathcal{A}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{A}}$, (s', χ') is a configuration and $(s_p, p!q, m, s'_p) \in \rightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.
  - $\chi'((p,q)) = \chi((p,q)) \cdot m$ and for $c \neq (p,q)$, $\chi'(c) = \chi(c)$.

  Then $(s, \chi) \stackrel{p!q}{\Longrightarrow} (s', \chi')$ and $(s', \chi') \in Conf_{\mathcal{A}}$.
- Suppose $(s, \chi) \in Conf_{\mathcal{A}}$, (s', χ') is a configuration and $(s_p, p?q, m, s'_p) \in \rightarrow_p$ such that the following conditions are satisfied:
  - $r \neq p$ implies $s_r = s'_r$ for each $r \in \mathcal{P}$.
  - $\chi((q,p)) = m \cdot \chi'((q,p))$ and for every channel $c \neq (q,p)$, $\chi'(c) = \chi(c)$.

  Then $(s, \chi) \stackrel{p?q}{\Longrightarrow} (s', \chi')$ and $(s', \chi') \in Conf_{\mathcal{A}}$.

Let $u \in \Sigma^*$. A run of $\mathcal{A}$ on u is a map $\rho : Pre(u) \rightarrow Conf_A$ (where $Pre(u)$ is the set of prefixes of u) such that $\rho(\varepsilon) = (s_{in}, \chi_\varepsilon)$ and for each $\tau a \in Pre(u)$, $\rho(\tau) \stackrel{a}{\Longrightarrow} \rho(\tau a)$. The run ρ is *accepting* if $\rho(u)$ is a final configuration. Let $L(\mathcal{A}) = \{u \mid \mathcal{A} \text{ has an accepting run on } u\}$. It is easy to see that every member of $L(\mathcal{A})$ is complete and $L(\mathcal{A})$ is $\sim$-closed—that is, $u \in L(\mathcal{A})$ and $u \sim u'$ implies $u' \in L(\mathcal{A})$.

Unfortunately, $L(\mathcal{A})$ need not be regular. Consider, for instance, a message-passing automaton for the canonical producer-consumer system in which the producer p sends an arbitrary number of messages to the consumer q. Since we can reorder all the $p!q$ actions to be performed before all the $q?p$ actions, the queue in channel (p,q) is unbounded. Hence, the reachable configurations of this system are not bounded and the corresponding language is not regular.

For $B \in \mathbb{N}$, we say that a configuration (s, χ) of the message-passing automaton $\mathcal{A}$ is *B-bounded* if for every channel $c \in Chan$, it is the case that $|\chi(c)| \leq B$. We say that $\mathcal{A}$ is a B-bounded automaton if every reachable configuration $(s, \chi) \in Conf_{\mathcal{A}}$ is B-bounded.

Proposition 3.1. *Let $\mathcal{A}$ be a B-bounded automaton over Σ. Then $L(\mathcal{A})$ is a regular MSC language.*

This result follows easily from the definitions. Our goal is to prove the converse, which may be stated as follows.

Theorem 3.2. *Let L be a regular MSC language over Σ. Then, there is a* deterministic *B-bounded message-passing automaton $\mathcal{A}$ over Σ such that $L(\mathcal{A}) = L$.*

Our strategy to prove this result is as follows. For a regular MSC language L, we consider the minimal DFA $\mathcal{A}_L$ for L. We construct a message-passing automaton $\mathcal{A}$ which simulates the behaviour of $\mathcal{A}_L$ on each complete word $u \in \Sigma^*$. The catch is that no single component of $\mathcal{A}$ is guaranteed to see all of u. Thus, from

the partial information available in each component about u, we have to reconstruct the behaviour of $\mathcal{A}_L$ on all of u. To achieve this, we need to time-stamp events so that components can keep track of each others' information about the computation.

4 Bounded Time-Stamps

Partial computations Let $u \in \Sigma^*$ be proper. A set of events $I \subseteq E_u$ is called an *(order) ideal* if I is closed with respect to $\leq$—that is, $e \in I$ and $f \leq e$ implies $f \in I$ as well.

Ideals denote consistent partial computations of u—notice that any linearization of an ideal forms a proper communication sequence.

p-views For an ideal I, the $\leq$-maximum p-event in I is denoted $\max_p(I)$, provided $\#_{\Sigma_p}(I) > 0$. The p-view of I, $\partial_p(I)$, is the ideal $\max_p(I)\downarrow$. Thus, $\partial_p(I)$ consists of all events in I which p can "see". (By convention, if $\max_p(I)$ is undefined—that is, if there is no p-event in I—the p-view $\partial_p(I)$ is empty.) For $P \subseteq \mathcal{P}$, we use $\partial_P(I)$ to denote $\bigcup_{p \in P} \partial_p(I)$.

Latest information Let $I \subseteq E_u$ be an ideal and $p, q \in \mathcal{P}$. Then $latest(I)$ denotes the set of events $\{\max_p(I) \mid p \in \mathcal{P}\}$. For $p \in \mathcal{P}$, we let $latest_p(I)$ denote the set $latest(\partial_p(I))$. A typical event in $latest_p(I)$ is of the form $\max_q(\partial_p(I))$ and denotes the $\leq$-maximum q-event in $\partial_p(I)$. This is the latest q-event in I that p knows about. For convenience, we denote this event $latest_{p \leftarrow q}(I)$. (As usual, if there is no q-event in $\partial_p(I)$, the quantity $latest_{p \leftarrow q}(I)$ is undefined.)

It is clear that for $q \neq p$, $latest_{p \leftarrow q}(I)$ will always correspond to a send action from Σ_q. However $latest_{p \leftarrow q}(I)$ need not be of the form $q!p$; the latest information that p has about q in I may have been obtained indirectly.

Message acknowledgments Let $I \subseteq E_u$ be an ideal and $e \in I$ an event of the form $p!q$. Then, e is said to have been *acknowledged* in I if the corresponding receive event f such that $e <_{pq} f$ exists and, moreover, belongs to $\partial_p(I)$. Otherwise, e is said to be *unacknowledged* in I.

Notice that it is not enough for a message to have been received in I to deem it to be acknowledged. We demand that the event corresponding to the receipt of the message be "visible" to the sending process.

For an ideal I and a pair of processes p, q, let $unack_{p \to q}(I)$ be the set of unacknowledged $p!q$ events in I.

B-bounded computations Let $u \in \Sigma^*$ be proper and let $M_u = (E_u, \leq, \lambda)$. We say that u is B-bounded, for $B \in \mathbb{N}$, if for every pair of processes p, q and for every ideal $I \subseteq E$, $unack_{p \to q}(I)$ contains at most B events.

The following result is immediate.

Proposition 4.1. *Let $u \in \Sigma^*$ be proper. The word u is B-bounded iff for every linearization v of M_u, for every prefix w of v and for every pair of processes p, q, $\#_{p!q}(w) - \#_{q?p}(w) \leq B$.*

It is easy to see that during the course of a B-bounded computation, none of the message buffers ever contains more than B undelivered messages, regardless of how the events are sequentialized. Thus, if each component $\mathcal{A}_p$ of a message-passing automaton is able to keep track of the sets $\{unack_{p\to q}(E_u)\}_{q\in\mathcal{P}}$ for each word u, this information can be used to inhibit sending messages along channels which are potentially saturated. This would provide a mechanism for constraining an arbitrary message-passing automaton to be B-bounded.

Primary information Let $I \subseteq E$ be an ideal. The *primary information* of I, $primary(I)$, consists of the following events in I:

- The set $latest(I) = \{\max_p(I) \mid p \in \mathcal{P}\}$.
- The collection of sets $unack(I) = \{unack_{p\to q}(I) \mid p, q \in \mathcal{P}\}$.

For $p \in \mathcal{P}$, we denote $primary(\partial_p(I))$ by $primary_p(I)$. Thus, $primary_p(I)$ reflects the primary information of p in I. Observe that for B-bounded computations, the number of events in $primary(I)$ is bounded.

In [13], a protocol is presented for processes to keep track of their primary information during the course of an arbitrary computation.[2] This protocol involves appending a bounded amount of information to each message in the underlying computation, provided the computation is B-bounded. To ensure that the message overhead is bounded, the processes use a distributed time-stamping mechanism which consistently assigns "names" to events using a bounded set of labels.

Consistent time-stamping Let $\mathcal{L}$ be a finite set of labels. For a proper communication sequence u, we say that $\tau : E_u \to \mathcal{L}$ is a *consistent time-stamping* of E_u by $\mathcal{L}$ if for each pair of (not necessarily distinct) processes p, q and for each ideal I the following holds: if $e_p \in primary_p(I)$, $e_q \in primary_q(I)$ and $\tau(e_p) = \tau(e_q)$ then $e_p = e_q$.

In the protocol of [13], whenever a process p sends a message to q, it first assigns a time-stamp to the new message from a finite set of labels. Process p then appends its primary information to the message being sent. Notice that the current send event will form part of the primary information since it is the latest p-event in $\partial_p(E_u)$. When q receives the message, it can consistently update its primary information to reflect the new information received from p.

The two tricky points in the protocol are for p to decide when it is safe to reuse a time-stamp, and for q to decide whether the information received from p is really new. In order to solve these problems, the protocol of [13] requires processes to also maintain additional time-stamps, corresponding to secondary information. Though we do not need the details of how the protocol works, we will need to refer to secondary information in the proof of our main theorem.

Secondary information Let I be an ideal. The *secondary information* of I is the collection of sets $primary(e{\downarrow})$ for each event e in $primary(I)$. This collection

[2] In [13], the primary information of an ideal I is defined to include more events than just $latest(I) \cup unack(I)$. However, for our purposes, it suffices to treat events in $latest(I) \cup unack(I)$ as primary.

of sets is denoted $secondary(I)$. As usual, for $p \in \mathcal{P}$, $secondary_p(I)$ denotes the set $secondary(\partial_p(I))$.

In our framework, the protocol of [13] can now be described as follows.

Theorem 4.2. *For any $B \in \mathbb{N}$, we can construct a* deterministic *B-bounded message-passing automaton $\mathcal{A}^B = (\{\mathcal{A}_p^B\}_{p\in\mathcal{P}}, \mathcal{M}^B, s_{in}^B, \mathcal{F}^B)$ such that for every B-bounded proper communication sequence u, $\mathcal{A}^B$ inductively generates a consistent time-stamping τ of E_u. Moreover, for each component $\mathcal{A}_p^B$ of $\mathcal{A}^B$, the local state of $\mathcal{A}_p^B$ at the end of u records the information $primary_p(E_u)$ and $secondary_p(E_u)$ in terms of the time-stamps assigned by τ.*

5 Residues and Decomposition

As we mentioned earlier, our strategy to prove our main theorem is to construct a message-passing automaton $\mathcal{A}$ which simulates the behaviour of the minimal DFA for L, $\mathcal{A}_L = (S, \Sigma, s_{in}, \delta, F)$, on each complete communication sequence u. In other words, after reading u, the components in $\mathcal{A}$ must be able to decide whether $\delta(s_{in}, u) \in F$. Unfortunately, after reading u each component in $\mathcal{A}$ only has partial information about $\delta(s_{in}, u)$—the component $\mathcal{A}_p$ only "knows about" those events from E_u which lie in the p-view $\partial_p(E_u)$. We have to devise a scheme to recover the state $\delta(s_{in}, u)$ from the partial information available with each process after reading u.

Another complication is that processes can only maintain a finite amount of information. We need a way of representing arbitrary words in a bounded, finite way. This can be done by recording for each word w, its "effect" as dictated by the minimal automaton $\mathcal{A}_L$. We associate with each word u a function $f_u : S \to S$, where S is the set of states of $\mathcal{A}_L$, such that $f_u(s) = \delta(s, u)$. The following observations follow from the fact that $\mathcal{A}_L$ is the minimal DFA recognizing L.

Proposition 5.1. *Let $u, w \in \Sigma^*$. Then:*

(i) $\delta(s_{in}, u) = f_u(s_{in})$.
(ii) $f_{uw} = f_w \circ f_u$, *where $\circ$ denotes function composition.*

Clearly the function $f_w : S \to S$ corresponding to a word w has a bounded representation. For an input u, if the components in $\mathcal{A}$ could compute the function f_u they would be able to determine whether $\delta(s_{in}, u) \in F$—by part (i) of the preceding proposition, $\delta(s_{in}, u) = f_u(s_{in})$. As the following result demonstrates, for any input u, it suffices to compute f_v for some linearization v of the MSC M_u.

Proposition 5.2. *Let $\hat{L}$ be a regular MSC language. For complete sequences $u, v \in \Sigma^*$, if $u \sim v$ then $f_u = f_v$.*

Proof: Follows from the structural properties of $\mathcal{A}_L$ described in Lemma 2.2. □

Before proceeding, we need a convention for representing the subsequence of communication actions generated by a subset of the events in an MSC.

Partial words Let $u = a_1 a_2 \ldots a_n$ be proper and let $X \subseteq E_u$ be given by $\{(i_1, a_{i_1}), (i_2, a_{i_2}), \ldots, (i_k, a_{i_k})\}$, where $i_1 < i_2 < \cdots < i_k$. Then, $u[X]$ denotes the subsequence $a_{i_1} a_{i_2} \ldots a_{i_k}$ (which need not be proper).

The following fact, analogous to standard results in Mazurkiewicz trace theory, will be used several times in our construction. We omit the proof.

Lemma 5.3. *Let u be proper and let $I, J \subseteq E_u$ be ideals such that $I \subseteq J$. Then $u[J] \sim u[I]u[J \setminus I]$.*

Corollary 5.4. *Let u be a word and $I_1 \subseteq I_2 \subseteq \cdots \subseteq I_k \subseteq E_u$ be a sequence of nested ideals. Then $u[I_k] \sim u[I_1]u[I_2 \setminus I_1] \cdots u[I_k \setminus I_{k-1}]$.*

Returning to our problem, suppose that $\mathcal{P}$ consists of m processes $\{p_1, p_2, \ldots, p_m\}$. Consider a complete word u. We wish to compute f_v for some $v \sim u$. Suppose we construct a chain of subsets of processes $\emptyset = Q_0 \subset Q_1 \subset Q_2 \subset \cdots \subset Q_m = \mathcal{P}$ such that for $j \in \{1, 2, \ldots, m\}$, $Q_j = Q_{j-1} \cup \{p_j\}$. From Corollary 5.4, we then have

$$u = u[\partial_{Q_m}(E_u)] \sim u[\partial_{Q_0}(E_u)]u[\partial_{Q_1}(E_u) \setminus \partial_{Q_0}(E_u)] \cdots u[\partial_{Q_m}(E_u) \setminus \partial_{Q_{m-1}}(E_u)]$$

Observe that $\partial_{Q_j}(E_u) \setminus \partial_{Q_{j-1}}(E_u)$ is the same as $\partial_{p_j}(E_u) \setminus \partial_{Q_{j-1}}(E_u)$. Thus, we can rewrite the expression above as

$$u = u[\partial_{Q_m}(E_u)] \sim u[\emptyset]u[\partial_{p_1}(E_u) \setminus \partial_{Q_0}(E_u)] \cdots u[\partial_{p_m}(E_u) \setminus \partial_{Q_{m-1}}(E_u)] \quad (\diamondsuit)$$

The word $u[\partial_{p_j}(E_u) \setminus \partial_{Q_{j-1}}(E_u)]$ is the portion of u which p_j has seen but which the processes in Q_{j-1} have not seen. This is a special case of what we call a residue.

Residues Let u be proper, $I \subseteq E_u$ an ideal and $p \in \mathcal{P}$ a process. $\mathcal{R}(u, p, I)$ denotes the word $u[\partial_p(E_u) \setminus I]$ and is called the *residue* of u at p with respect to I. Observe that any residue of the form $\mathcal{R}(u, p, I)$ can equivalently be written $\mathcal{R}(u, p, \partial_p(E_u) \cap I)$.

Using the notation of residues, we can write the word $u[\partial_{p_j}(E_u) \setminus \partial_{Q_{j-1}}(E_u)]$ as $\mathcal{R}(u, p_j, \partial_{Q_{j-1}}(E_u))$. A residue of this form is called a *process residue*: $\mathcal{R}(u, p, I)$ is a process residue if $\mathcal{R}(u, p, I) = \mathcal{R}(u, p, \partial_P(E_u))$ for some $P \subseteq \mathcal{P}$. We say that $\mathcal{R}(u, p, \partial_P(E_u))$ is the P-residue of u at p.

Unfortunately, a process residue at p may change due to an action of another process. For instance, if we extend a word u by an action $a = q?p$, it is clear that $\mathcal{R}(u, p, \partial_q(E_u))$ will not be the same as $\mathcal{R}(ua, p, \partial_q(E_{ua}))$ since q will get to know about more events from $\partial_p(u)$ after receiving the message via the action a. On the other hand, since p does not move on an action of the form $q?p$, p has no chance to update its q-residue when the action $q?p$ occurs.

However, it turns out that each process can maintain a set of residues based on its primary information such that these *primary residues* subsume the process residues. The key technical fact which makes this possible is the following.

Lemma 5.5. *For any non-empty ideal I, and $p, q \in \mathcal{P}$, the maximal events in $\partial_p(I) \cap \partial_q(I)$ lie in $primary_p(I) \cap primary_q(I)$.*

Proof: We show that for each maximal event e in $\partial_p(I) \cap \partial_q(I)$, either $e \in latest(\partial_p(I)) \cap unack(\partial_q(I))$ or $e \in unack(\partial_p(I)) \cap latest(\partial_q(I))$.

First suppose that $\partial_p(I) \setminus \partial_q(I)$ and $\partial_q(I) \setminus \partial_p(I)$ are both nonempty. Let e be a maximal event in $\partial_p(I) \cap \partial_q(I)$. Suppose e is an r-event, for some $r \in \mathcal{P}$. Since $\partial_p(I) \setminus \partial_q(I)$ and $\partial_q(I) \setminus \partial_p(I)$ are both nonempty, it follows that $r \notin \{p, q\}$. The event e must have $\leq$-successors in both $\partial_p(I)$ and $\partial_q(I)$. However, observe that any event f can have at most two immediate $\leq$-successors—one "internal" successor within the process and, if f is a send event, one "external" successor corresponding to the matching receive event.

Thus, the maximal event e must be a send event, with a $<_{rr}$ successor e_r and a $<_{rs}$ successor e_s, corresponding to some $s \in \mathcal{P}$. Assume that $e_r \in \partial_q(I) \setminus \partial_p(I)$ and $e_s \in \partial_p(I) \setminus \partial_q(I)$. Since the r-successor of e is outside $\partial_p(I)$, $e = \max_r(\partial_p(I))$. So e belongs to $latest(\partial_p(I))$. On the other hand, e is an unacknowledged $r!s$ event in $\partial_q(I)$. Thus, $e \in unack_{r \to s}(\partial_q(I))$, which is part of $unack(\partial_q(I))$.

Symmetrically, if $e_r \in \partial_p(I) \setminus \partial_q(I)$ and $e_s \in \partial_q(I) \setminus \partial_p(I)$, we find that e belongs to $unack(\partial_p(I)) \cap latest(\partial_q(I))$.

We still have to consider the case when $\partial_p(I) \subseteq \partial_q(I)$ or $\partial_q(I) \subseteq \partial_p(I)$. Suppose that $\partial_p(I) \subseteq \partial_q(I)$, so that $\partial_p(I) \cap \partial_q(I) = \partial_p(I)$. Let $e = \max_p(\partial_q(I))$. Clearly, $\partial_p(I) = e{\downarrow}$. Consider any r-event f in $\partial_p(I)$, where $r \notin \{p, q\}$. Since $f < e$, f cannot be maximal in $\partial_p(I)$. Thus, the only maximal event in $\partial_p(I)$ is the p-event e. Since e has a successor in $\partial_q(I)$, e must be a send event and is hence in $unack(\partial_p(I))$. Thus, $e \in unack(\partial_p(I)) \cap latest(\partial_q(I))$. Symmetrically, if $\partial_q(I) \subseteq \partial_p(I)$, the unique maximal event e in $\partial_q(I)$ belongs to $latest(\partial_p(I)) \cap unack(\partial_q(I))$. □

Let us call $\mathcal{R}(u, p, I)$ a *primary residue* if I is of the form $X{\downarrow}$ for some subset $X \subseteq primary_p(E_u)$. Clearly, for $p, q \in \mathcal{P}$, $\mathcal{R}(u, p, \partial_q(E_u))$, can be rewritten as $\mathcal{R}(u, p, \partial_p(E_u) \cap \partial_q(E_u))$. So, by the previous result the q-residue $\mathcal{R}(u, p, \partial_q(E_u))$ is a primary residue $\mathcal{R}(u, p, X{\downarrow})$ for some $X \subseteq primary(\partial_p(E_u))$. Further, the set X can be effectively computed from the primary information of p and q. In fact, it turns out that *all* process residues can be effectively described in terms of primary residues.

We begin with a simple observation, whose proof we omit.

Proposition 5.6. *Let $u \in \Sigma^*$ be proper and $p \in \mathcal{P}$. For ideals $I, J \subseteq E_u$, let $\mathcal{R}(u, p, I)$ and $\mathcal{R}(u, p, J)$ be primary residues such that $\mathcal{R}(u, p, I) = \mathcal{R}(u, p, X_I{\downarrow})$ and $\mathcal{R}(u, p, J) = \mathcal{R}(u, p, X_J{\downarrow})$ for $X_I, X_J \subseteq primary_p(E_u)$. Then $\mathcal{R}(u, p, I \cup J)$ is also a primary residue and $\mathcal{R}(u, p, I \cup J) = \mathcal{R}(u, p, (X_I \cup X_J){\downarrow})$.*

Our claim that all process residues can be effectively described in terms of primary residues can then be formulated as follows.

Lemma 5.7. *Let $u \in \Sigma^*$ be proper, $p \in \mathcal{P}$ and $Q \subseteq \mathcal{P}$. Then $\mathcal{R}(u, p, \partial_Q(E_u))$ is a primary residue $\mathcal{R}(u, p, X\downarrow)$ for p. Further, the set $X \subseteq primary_p(E_u)$ can be effectively computed from the information in $\bigcup_{q \in \{p\} \cup Q} primary_q(E_u)$.*

Proof: Let $Q = \{q_1, q_2, \ldots, q_k\}$. We can rewrite $\mathcal{R}(u, p, \partial_Q(E_u))$ as $\mathcal{R}(u, p, \bigcup_{i \in [1..k]} \partial_{q_i}(E_u))$. From Lemma 5.5 it follows that for each $i \in \{1, 2, \ldots, k\}$, p can compute a set $X_i \subseteq primary_p(E_u)$ from the information in $primary_p(E_u) \cup primary_{q_i}(E_u)$ such that $\mathcal{R}(u, p, \partial_{q_i}(E_u)) = \mathcal{R}(u, p, X_i\downarrow)$. From Proposition 5.6, it then follows that $\mathcal{R}(u, p, \partial_Q(E_u)) = \mathcal{R}(u, p, \bigcup_{i \in \{1,2,\ldots,k\}} \partial_{q_i}(E_u)) = \mathcal{R}(u, p, X\downarrow)$ where $X = \bigcup_{i \in \{1,2,\ldots,k\}} X_i$. □

6 Updating Residues

We now describe how, while reading a word u, each process p maintains the functions f_w for each primary residue w of u at p.

Initially, at the empty word $u = \varepsilon$, every primary residue from $\{\mathcal{R}(u, p, X\downarrow)\}_{p \in \mathcal{P}, X \subseteq primary(\partial_p(E_u))}$ is just the empty word ε. So, all primary residues are represented by the identity function $Id : \{s \mapsto s\}$.

Let $u \in \Sigma^*$ and $a \in \Sigma$. Assume inductively that every $p \in \mathcal{P}$ has computed at the end of u the function f_w for each primary residue $w = \mathcal{R}(u, p, X\downarrow)$, where $X \subseteq primary(\partial_p(E_u))$. We want to compute for each p the corresponding functions after the word ua.

Suppose a is of the form $p!q$ and $X \subseteq primary_p(E_{ua})$. Let e_a denote the event corresponding to the new action a. If $e_a \in X$, then $\mathcal{R}(ua, p, X\downarrow) = \varepsilon$, so we represent the residue by the identity function Id. On the other hand, if $a \notin X$, then $X \subseteq primary_p(E_u)$, so we already have a residue of the form $\mathcal{R}(u, p, X)$. We then set $\mathcal{R}(ua, p, X\downarrow)$ to be $f_a \circ \mathcal{R}(u, p, X\downarrow)$. For $r \neq p$, the primary residues are unchanged when going from u to ua.

The case where a is of the form $p?q$ is more interesting. As before, the primary residues are unchanged for $r \neq p$. We show how to calculate all the new primary residues for p using the information obtained from q. This will use the following result.

Lemma 6.1. *Let $u \in \Sigma^*$ be proper. Let $p, q \in \mathcal{P}$ and $e \in E_u$ such that $e \in primary_q(E_u)$ but $e \notin \partial_p(E_u)$. Then $\mathcal{R}(u, p, e\downarrow)$ is a primary residue $\mathcal{R}(u, p, X\downarrow)$ for p. Further, the set $X \subseteq primary(\partial_p(E_u))$ can be effectively computed from the information in $primary_p(E_u)$ and $secondary_q(E_u)$.*

Proof: Let e be an r-event, $r \in \mathcal{P}$ and let $J = \partial_p(E_u) \cup e\downarrow$. By construction, $\max_p(J) = \max_p(E_u)$. On the other hand, $\max_r(J) = e$, since e is an r-event and we assumed that $e \notin \partial_p(E_u)$.

By Lemma 5.5, the maximal events in $\partial_p(J) \cap \partial_r(J)$ lie in $primary_p(J) \cap primary_r(J)$. Since $\max_p(J) = \max_p(E_u)$, $primary_p(J) = primary_p(E_u)$. On the other hand, $primary_r(J) = primary(e\downarrow)$, which is a subset of $secondary_q(E_u)$, since $e \in primary_q(E_u)$.

Thus, the set of maximal events in $\partial_p(J) \cap \partial_r(J)$, which is the same as $\partial_p(E_u) \cap e{\downarrow}$, is contained in $primary_p(E_u) \cap primary(e{\downarrow})$. These events are available in $primary_p(E_u) \cup secondary_q(E_u)$. □

Suppose that $X \subseteq primary_p(E_{ua})$. Suppose that $X = \{x_1, x_2, \ldots, x_k\}$.

We first argue that for each $x_i \in X$, $\mathcal{R}(u, p, x_i{\downarrow})$ is a primary residue $\mathcal{R}(u, p, Y_i{\downarrow})$, where $Y_i \subseteq primary_p(E_u)$. If $x_i \in primary_p(E_u)$, then $\mathcal{R}(u, p, x_i{\downarrow})$ is already a primary residue, so we can set $Y_i = \{x_i\}$. If, however, $x_i \notin primary_p(E_u)$, then x_i must have been contributed from $primary_q(u)$ through the message received at the action a. We have $x_i \in primary_q(E_u)$ but $x_i \notin \partial_p(E_u)$. Thus, appealing to Lemma 6.1, we can identify $Y_i \subseteq primary_p(E_u)$ such that $\mathcal{R}(u, p, \{x_i\}{\downarrow}) = \mathcal{R}(u, p, Y_i{\downarrow})$.

Since $X = \bigcup_{i \in \{1,2,\ldots,k\}} x_i$, we can appeal to Proposition 5.6 to argue that $\mathcal{R}(u, p, X{\downarrow})$ is the primary residue $\mathcal{R}(u, p, Y{\downarrow})$ where $Y = \bigcup_{i \in \{1,2,\ldots,k\}} Y_i$. We can then set $\mathcal{R}(ua, p, X{\downarrow}) = f_a \circ \mathcal{R}(u, p, Y{\downarrow})$.

Thus, after each action that is performed, the process performing the action can effectively update its primary residues using the primary and secondary information available to it.

7 A Deterministic Message-Passing Automaton for L

We can now construct a deterministic B-bounded message-passing automaton corresponding to a given regular MSC language L. We first observe that there is a bound $B \in \mathbb{N}$ such that every word in L is B-bounded.

Proposition 7.1. *Let $L \subseteq \Sigma^*$ be a regular MSC language. There is an effectively computable bound $B \in \mathbb{N}$ such that every word in L is B-bounded.*

Proof: Let $\mathcal{A}_L = (S, \Sigma, s_{in}, \delta, F)$ be the minimal DFA for L. Recall that we can associate with each live state in S and each channel $(p, q) \in Chan$ a channel-capacity function $\mathcal{K}_s : Chan \to \mathbb{N}$. Let B be the maximum value of $\mathcal{K}_s((p, q))$ over all live states s and all channels (p, q).

We know that for any word u in L, the run of $\mathcal{A}_L$ on u visits only live states. Thus, while processing u, no channel's capacity ever exceeds the bound B. Moreover, since L is $\sim$-closed, every interleaving v of M_u belongs to L and the run of $\mathcal{A}_L$ on each such interleaving also respects this bound. From Proposition 4.1, we can conclude that in every ideal $I \subseteq E_u$, for any pair p, q of processes, the set $unack_{p \to q}(I)$ contains at most B-events. Thus, u is B-bounded. □

We now construct a B-bounded message-passing automaton $\mathcal{A} = (\{\mathcal{A}_p\}_{p \in \mathcal{P}}, \mathcal{M}, s_{in}, \mathcal{F})$ for L, where B is the bound derived from the minimal DFA $\mathcal{A}_L$ for L as described in the preceding proposition.

Recall that $\mathcal{A}^B = (\{\mathcal{A}_p^B\}_{p \in \mathcal{P}}, \mathcal{M}^B, s_{in}^B, \mathcal{F}^B)$ is the time-stamping automaton for B-bounded computations, where the state of each component records the primary and secondary information of the component in terms of a consistent set of time-stamps.

- The message alphabet of $\mathcal{A}$ is the alphabet $\mathcal{M}^B$ used by the time-stamping automaton $\mathcal{A}^B$.
- In $\mathcal{A}$, a typical state of a component $\mathcal{A}_p$ is a pair (s_B, s_R) where s_B is a state drawn from $\mathcal{A}_p^B$ and s_R is the collection $\{f_X : S \to S\}_{X \subseteq primary_p(E_u)}$ of primary residues of $\mathcal{A}_p$ at the end of a word u.
- The local transition relation $\rightarrow_p$ of each component $\mathcal{A}_p$ is as follows:
  - For a of the form $p!q$, the tuple $((s_B, s_R), a, m, (s'_B, s'_R)) \in \rightarrow_p$ provided $(s_B, a, m, s'_B) \in \rightarrow_p^B$ and the residues in s'_R are derived from the residues in s_R using the time-stamping information in s_B, as described in Section 6.

    Moreover, according to the primary information in s_B, it should be the case that $|unack_{p \to q}(E_u)| < B$ for the word u read so far. Otherwise, this send action is disabled.
  - For a of the form $p?q$, the tuple $((s_B, s_R), a, m, (s'_B, s'_R)) \in \rightarrow_p$ provided $(s_B, a, m, s'_B) \in \rightarrow_p^B$ and the residues in s'_R are derived from the residues in s_R using the time-stamping information in s_B and the message m, as described in Section 6.
- In the initial state of $\mathcal{A}$, the local state of each component $\mathcal{A}_p$ is of the form $(s^p_{B,in}, s^p_{R,in})$ where $s^p_{B,in}$ is the initial state of $\mathcal{A}_p^B$ and $s^p_{R,in}$ records each residue to be the identity function Id.
- The global state $\{(s^p_B, s^p_R)\}_{p \in \mathcal{P}}$ belongs to the set $\mathcal{F}$ of final states if the primary residues stored in the global state record that $\delta(s_{in}, u) \in F$ for the word u read so far. (This is achieved by evaluating the expression $(\diamondsuit)$ in Section 5.)

From the analysis of the previous section, it is clear that $\mathcal{A}$ accepts precisely the language L. The last clause in the transition relation $\rightarrow_p$ for send actions ensures that $\mathcal{A}$ will not admit a run in which $unack_{p \to q}(E_u)$ grows beyond B events for any input u and any pair of processes p, q. This ensures that every reachable configuration of $\mathcal{A}$ is B-bounded. Finally, we observe that $\mathcal{A}$ is deterministic because the time-stamping automaton $\mathcal{A}^B$ is deterministic and the update procedure for residues described in Section 6 is also deterministic.

We have thus succeeded in proving the main result we were after (Theorem 3.2)—namely, that for every regular MSC language L over Σ, there is a *deterministic* B-bounded message-passing automaton $\mathcal{A}$ over Σ such that $L(\mathcal{A}) = L$.

We conclude by providing an upper bound for the size of $\mathcal{A}$, which can be computed by estimating the number of bits required to record the time-stamps and residues which form the local state of a process.

Proposition 7.2. *Let n be the number of processes in the system, m be the number of states of the minimal DFA $\mathcal{A}_L$ for L and B the bound computed from the channel-capacity functions of $\mathcal{A}_L$. Then, the number of local states of each component $\mathcal{A}_p$ is at most $2^{(2^{O(Bn^2)} m \log m)}$.*

References

1. Alur, R., Holzmann, G. J., and Peled, D.: An analyzer for message sequence charts. *Software Concepts and Tools*, **17(2)** (1996) 70–77. 522, 524
2. Alur, R., and Yannakakis, M.: Model checking of message sequence charts. *Proc. CONCUR'99*, LNCS **1664**, Springer Verlag (1999) 114–129. 522
3. Ben-Abdallah, H., and Leue, S.: Syntactic detection of process divergence and non-local choice in message sequence charts. *Proc. TACAS'97*, LNCS **1217**, Springer-Verlag (1997) 259–274. 522
4. Booch, G., Jacobson, I., and Rumbaugh, J.: *Unified Modeling Language User Guide*. Addison Wesley (1997). 521
5. Damm, W., and Harel, D.: LCS's: Breathing life into message sequence charts. *Proc. FMOODS'99*, Kluwer Academic Publishers (1999) 293–312. 522
6. Diekert, V., and Rozenberg, G. (Eds.): *The book of traces*. World Scientific (1995). 521
7. Harel, D., and Gery, E.: Executable object modeling with statecharts. *IEEE Computer*, July 1997 (1997) 31–42. 521
8. Henriksen, J. G., Mukund, M., Narayan Kumar K., and Thiagarajan, P. S.: On message sequence graphs and finitely generated regular MSC languages, to appear in *Proc. ICALP 2000*, LNCS, Springer-Verlag (2000). 522
9. Henriksen, J. G., Mukund, M., Narayan Kumar K., and Thiagarajan, P. S.: Regular collections of message sequence charts, to appear in *Proc. MFCS 2000*, LNCS, Springer-Verlag (2000). 521, 522, 524
10. Ladkin, P. B., and Leue, S.: Interpreting message flow graphs. *Formal Aspects of Computing* **7**(5) (1975) 473–509. 522
11. Levin, V., and Peled, D.: Verification of message sequence charts via template matching. *Proc. TAPSOFT'97*, LNCS **1214**, Springer-Verlag (1997) 652–666. 522
12. Mauw, S., and Reniers, M. A.: High-level message sequence charts, *Proc. SDL '97*, Elsevier (1997) 291–306. 522
13. Mukund, M., Narayan Kumar, K., and Sohoni, M.: Keeping track of the latest gossip in message-passing systems. *Proc. Structures in Concurrency Theory (STRICT)*, Workshops in Computing Series, Springer-Verlag (1995) 249–263. 522, 528, 529
14. Muscholl, A.: Matching Specifications for Message Sequence Charts. *Proc. FOSSACS'99*, LNCS **1578**, Springer-Verlag (1999) 273–287. 522
15. Muscholl, A., Peled, D., and Su, Z.: Deciding properties for message sequence charts. *Proc. FOSSACS'98*, LNCS **1378**, Springer-Verlag (1998) 226–242. 522
16. Rudolph, E., Graubmann, P., and Grabowski, J.: Tutorial on message sequence charts. In *Computer Networks and ISDN Systems—SDL and MSC*, Volume 28 (1996). 521, 524
17. Thomas, W.: Automata on infinite objects. In van Leeuwen, J. (Ed.), *Handbook of Theoretical Computer Science, Volume B*, North-Holland, Amsterdam (1990) 133–191.
18. Thomas, W.: Languages, Automata, and Logic. In Rozenberg, G., and Salomaa, A. (Eds.), *Handbook of Formal Language Theory, Vol. III*, Springer-Verlag, New York (1997) 389–455.
19. Zielonka, W.: Notes on finite asynchronous automata. *R. A. I. R. O.—Inf. Théor. et Appl.*, **21** (1987) 99–135. 521, 522

Emptiness Is Decidable for Asynchronous Cellular Machines

Dietrich Kuske

Institut für Algebra, Technische Universität Dresden,
D-01062 Dresden, Germany
kuske@math.tu-dresden.de

Abstract. We resume the investigation of asynchronous cellular automata. Originally, these devices were considered in the context of Mazurkiewicz traces, and later generalized to run on arbitrary pomsets without autoconcurrency by Droste and Gastin [3]. While the universality of the accepted language is known to be undecidable [11], we show here that the emptiness is decidable. Our proof relies on a result due to Finkel and Schnoebelen [7] on well-structured transition systems.

1 Introduction

In a distributed system, some events may occur concurrently, meaning that they may occur in any order or simultaneously or even that their executions may overlap. This is the case in particular when two events use independent resources. On the other hand, some events may causally depend on each other. For instance, the receiving of a message must follow its sending. Therefore, a distributed behavior may be abstracted as a directed acyclic graph (dag), that is a set of events together with an edge relation which describes causal dependencies of events and with a labeling function. A typical example are pomsets without autoconcurrency: The edge relation can be chosen to be the covering relation and concurrent events must have different labels. These pomsets are called semiwords in [14]. For studies how general pomsets can be used to represent parallel processes and how they can be composed, we refer the reader e.g. to [13,9].

Dependence graphs (known from the theory of Mazurkiewicz traces [5]) are another example of dags that model concurrent behaviors: A dependence graph is a dag whose edges are dictated by a static dependence relation between the actions of the system. Asynchronous cellular automata (ACAs for short) are a computational model that faithfully reflects the concurrency in a dependence graph. Zielonka [15] showed that ACAs are suitable to describe the behavior of a finite state system in a distributed way. In particular, it was shown that the expressive power of ACAs on Mazurkiewicz traces coincides with that of monadic second order logic MSO (cf. [6]), i.e. a set of traces can be accepted by an ACA iff it is definable in MSO. Since ACAs generalize finite automata, this extends Büchi's Theorem [1] from finite words to dependence graphs. Furthermore, it is decidable whether two ACAs accept the same set of traces ("equivalence") [15].

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 536–551, 2000.

This implies in particular that it is decidable whether an asynchronous cellular automaton accepts all traces ("universality") or no trace at all ("emptiness").

In [3], the model of ACAs was extended to run on the Hasse-diagram of an arbitrary pomset without autoconcurrency. Droste and Gastin defined the notion of CROW-pomsets and extended the results from the context of traces to that of CROW-pomsets. In particular, they showed that a set of CROW-pomsets can be accepted by an ACA iff it is definable in MSO. They also showed that the equivalence of ACAs when restricted to CROW-pomsets is decidable.

In [11], so called k-pomsets were introduced that generalize CROW-pomsets. It was shown that the results mentioned above on ACAs for CROW-pomsets hold for k-pomsets, too. In particular, the expressive power of ACAs on k-pomsets coincides with that of MSO, and the equivalence of ACAs when restricted to k-pomsets is decidable. On the other hand, it was shown that the universality (and therefore the equivalence) for the unrestricted class of all pomsets is undecidable.

In common approaches to model checking [2], the decidability of the emptiness is of the utmost importance. Since, as was also shown in [11], an ACA can in general not be complemented, the decidability status of this question was left open for general pomsets. Here, we show that the emptiness is decidable in the context of Σ-dags that slightly differs from [3,11,4] where essentially Hasse-diagrams were considered: Essentially, a *Σ-dag* is a Σ-labeled dag (V, E, λ) where (for any action $a \in \Sigma$) any node $y \in V$ has at most one in-node and at most one out-node labeled by a (i.e. there is at most one a-labeled node x and one a-labeled node z with $(x, y) \in E$ and $(y, z) \in E$). For Mazurkiewicz traces, a Σ-dag is an intermediate structure between its dependence graph and its Hasse-diagram, but not all Σ-dags arise this way. By [12], the results from [11] mentioned above can be transfered to Σ-dags; in particular, the universality and therefore the equivalence is undecidable. Here, we show the decidability of the question whether an ACA accepts some Σ-dag. Moreover, we show the decidability of the emptiness not only for finite-state ACAs, but allow them to have infinitely many local states that carry a well quasi order.

The main result of the paper is shown using a result of Finkel and Schnoebelen [7]: A well-structured transition system is a transition system whose set of states carries a well quasi ordering where any transition "lifts" to a larger state (see below for a formal definition). Finkel and Schnoebelen showed several decidability results in this context; here, we use the uniform decidability of the "covering problem". To this aim, we first have to construct from an ACA a well-structured transition system that faithfully reflects the behavior of the ACA. This construction uses in particular Higman's Theorem.

2 Basic Definitions

Recall that a *quasi order* is a transitive and reflexive relation $\preceq \subseteq S \times S$ on a set S. For $A \subseteq S$, let $\uparrow A = \{s \in S \mid \exists a \in A : a \preceq s\}$ denote the generated filter. We abbreviate $\uparrow\{a\}$ by $\uparrow a$. A quasi ordered set $(S, \preceq)$ is a *well quasi order* or *wqo* if for any infinite sequence $(s_i)_{i\in\mathbb{N}}$ of elements of S there exist $1 \leq i < j$

with $s_i \preceq s_j$. If $\preceq$ is a partial order, this is equivalent to the requirement that there are no infinite antichains and no infinite decreasing chains. We say that B is a *basis* of A if $\uparrow A = \uparrow B$. In a wqo, any subset A has a *finite* basis.

Let Σ be an alphabet and let $D \subseteq \Sigma^2$ be a reflexive and symmetric relation. A *Mazurkiewicz trace* over (Σ, D) is a Σ-labeled dag (V, E, λ) such that $(\lambda(x), \lambda(y)) \in D$ iff $x = y$ or x and y are connected by an edge from E.

For a function $f : A \to B$, let $\mathrm{dom}(f) = A$ be its *domain* and $\mathrm{im}(f) = \{f(a) \mid a \in A\}$ its *image* (which can in general be a proper subset of B).

Throughout this paper, alphabets are always assumed to be finite and non-empty. Furthermore, if not stated otherwise, let Σ be some fixed alphabet.

3 Σ-Dags and Asynchronous Cellular Machines

We start with the definition of a Σ-dag. These directed acyclic graphs generalize an aspect of dependence graphs known from trace theory: As defined above, a dependence graph is a Σ-labeled dag (V, E, λ). Then $E^\star$ is a partial order on the set of vertices V. Since D is assumed to be reflexive, the set $\lambda^{-1}(a) \subseteq V$ of a-labeled vertices is linearly ordered w.r.t. $E^\star$. Furthermore, a dependence graph contains lots of redundant edges: If, e.g., $(a, b) \in D$, then any a-labeled vertex is connected with all b-labeled vertices. Let y be some a-labeled vertex. Performing this vertex y, an asynchronous cellular automaton as considered by Zielonka reads only the maximal b-labeled vertex x satisfying $(x, y) \in E$. So let $E' \subseteq E$ consist of all edges $(x, y) \in E$ such that $(x', y') \in E$, $\lambda(x) = \lambda(x')$, $\lambda(y) = \lambda(y')$, $xE^\star x'$ and $y'E^\star y$ imply $x = x'$ and $y = y'$. Then (V, E', λ) is a Σ-dag:

Definition 1. *Let Σ be an alphabet. A* Σ-dag *is a triple (V, E, λ) where (V, E) is a finite directed acyclic graph and $\lambda : V \to \Sigma$ is a labeling function such that*

1. *$\lambda^{-1}(a)$ is linearly ordered with respect to $E^\star$ for any $a \in \Sigma$, and*
2. *for any $(x, y), (x', y') \in E$ with $\lambda(x) = \lambda(x')$, $\lambda(y) = \lambda(y')$, we have $(x, x') \in E^\star$ if and only if $(y, y') \in E^\star$.*

By $\mathbb{D}(\Sigma)$, we denote the set of all Σ-dags.

As usual, we will identify isomorphic Σ-dags. So let $(V, E, \lambda) \in \mathbb{D}(\Sigma)$. Since (V, E) is acyclic, $E^\star$ is a partial order on V. By the first requirement, vertices that carry the same label are comparable with respect to the partial order $E^\star$. In particular, the width of the partially ordered set $(V, E^\star)$ is bounded by $|\Sigma|$ and there is a natural covering of a Σ-dag by the $|\Sigma|$ chains $\lambda^{-1}(a)$ for $a \in \Sigma$. Any edge connects two such chains. The second clause ensures in particular that two edges connecting the same chains (in the same direction) cannot "cross". More precisely, let $(x, y), (x', y') \in E$ with $\lambda(x) = \lambda(x')$ and $\lambda(y) = \lambda(y')$, i.e. these two edges connect the same chains in the same direction. Then, by the first requirement, x and x' are comparable with respect to $E^\star$, say $(x, x') \in E^\star$. Then the second requirement forces $(y, y') \in E^\star$. In particular, if $(x, y), (x, y') \in E$ with $\lambda(y) = \lambda(y')$, then $y = y'$ and dually if $(x, y), (x', y) \in E$ with $\lambda(x) = \lambda(x')$ then x and x' are forced to be equal.

Example 1. 1. Let $\Sigma = \{a, b, c\}$. Then the labeled directed acyclic graph depicted on the left in Figure 1 is a Σ-dag.

2. Let $(V, \leq)$ be a finite partial order and $\lambda : V \to \Sigma$ be a mapping. Then the triple $(V, \leq, \lambda)$ is a *pomset without autoconcurrency* if, for any $a \in \Sigma$, the set $\lambda^{-1}(a)$ is linearly ordered by $\leq$ (The middle dag in Figure 1 is a pomset without autoconcurrency). Note that $(V, E^\star, \lambda)$ is a pomset without autoconcurrency for any Σ-dag (V, E, λ). Conversely, a pomset without autoconcurrency is not a Σ-dag for it may violate the second requirement. Now let $x \lessdot y$ denote that $x < y$ and there is no element properly between x and y (We say that x *is covered by* y). The *Hasse-diagram* $\mathrm{Ha}(V, \leq, \lambda)$ of $(V, \leq, \lambda)$ is the labeled directed acyclic graph $(V, \lessdot, \lambda)$. It is easily checked that this Hasse-diagram is a Σ-dag whenever $(V, \leq, \lambda)$ is a pomset without autoconcurrency (cf. the right dag in Figure 1).

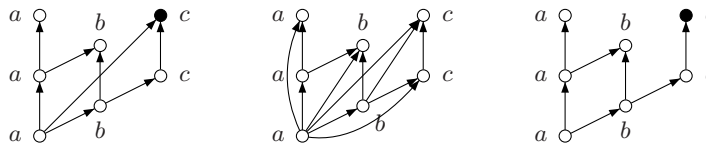


Fig. 1. A Σ-dag, a pomset without autoconcurrency and its Hasse-diagram

Let $t = (V, E, \lambda) \in \mathbb{D}(\Sigma)$ be a Σ-dag and $x \in V$. Then the *reading domain* $\mathrm{R}(x)$ of x is the set of all letters a from Σ that satisfy

$$\exists y \in V : \lambda(y) = a \text{ and } (y, x) \in E,$$

i.e. $R(x)$ is the set of labels of those nodes $y \in V$ that are connected with x by an edge (y, x). Informally, these nodes can be seen as the lower neighbors of x in the dag (V, E, λ) (but not necessarily in the partial order $(V, E^\star, \lambda)$). For $a \in \mathrm{R}(x)$, let $\partial_a(x)$ denote the (unique) element y of $\lambda^{-1}(a)$ with $(y, x) \in E$. Thus, $\{\partial_a(x) \mid a \in \mathrm{R}(x)\}$ is the set of lower neighbors of x in the Σ-dag t.

Example 1 (continued) Let x denote the element of the left Σ-dag from Figure 1 depicted by a solid circle. Since x is the target of edges whose source is labeled by a and by c, respectively, the reading domain $\mathrm{R}(x)$ is $\{a, c\}$. Differently, the solid circle in the Σ-dag from Figure 1 is the target of only one edge whose source is labeled by c. Hence, for this Σ-dag, $\mathrm{R}(x) = \{c\}$.

Next we define asynchronous cellular machines, the model of parallel systems that we are going to investigate. An *asynchronous cellular machine over* Σ or Σ*-ACM* is a tuple $\mathcal{A} = ((Q_a, \sqsubseteq_a)_{a \in \Sigma}, (\delta_{a,J})_{a \in \Sigma, J \subseteq \Sigma}, F)$ where

1. $(Q_a, \sqsubseteq_a)$ is an at most countable, well-quasi ordered set of local states for any $a \in \Sigma$,
2. $\delta_{a,J} : \prod_{b \in J} Q_b \to 2^{Q_a}$ is a nondeterministic transition function for any $a \in \Sigma$, $J \subseteq \Sigma$, and
3. $F \subseteq \bigcup_{J \subseteq \Sigma} \prod_{b \in J} Q_b$ is a finite set of accepting states.

One can think of a Σ-ACM as a Σ-tuple of sequential devices. The device with index a performs the a-labeled events of an execution. Doing so, it reads states from other constituents of the Σ-ACM. But it changes its own state, only (see below for a formal definition of a run). The set of all local states $\bigcup_{a\in\Sigma} Q_a$ will be denoted by Q.

Example 2. Let Σ be an alphabet. For $a \in \Sigma$, let $Q_a := \mathbb{N}^\Sigma$ be the set of all functions $\Sigma \to \{0, 1, 2, \dots\}$. The local wqos $\sqsubseteq_a$ are defined by $f \sqsubseteq_a g$ iff $f(b) \le g(b)$ for any $b \in \Sigma$. Next, we define the transition function by

$$\delta_{a,J}((f_c)_{c\in J}) := \begin{cases} \emptyset & \text{if there exist } b, c \in J \text{ with } b \neq c \text{ and } f_b(c) \ge f_c(c) \\ \{g\} & \text{otherwise} \end{cases}$$

where the function $g : \Sigma \to \mathbb{N}$ is given by

$$g(b) := \begin{cases} \sup\{f_c(b) \mid c \in J\} & \text{if } a \neq b \\ 1 + \sup\{f_c(b) \mid c \in J\} & \text{if } a = b. \end{cases}$$

Furthermore, F is the set of all tuples $(f_c)_{c\in J}$ for $J \subseteq \Sigma$ with $f_c(b) \in \{0, 1\}$ for all $b \in \Sigma$ and $f_c(c) = 0$ for $c \in J$. We will return to this example later and show that $\mathcal{A}$ accepts the set of all Hasse-diagrams of pomsets without autoconcurrency.

A Σ-ACM is called *asynchronous cellular automaton* over Σ (Σ-*ACA* for short) provided the sets of local states Q_c are finite for $c \in \Sigma$. Usually, for an ACA we will assume the wqos $\sqsubseteq_c$ to be the trivial reflexive relation Δ_{Q_c} on Q_c.

Next we define how a Σ-ACM can run on a Σ-dag and when it accepts a Σ-dag. Let $t = (V, E, \lambda)$ be a Σ-dag and $\mathcal{A}$ a Σ-ACM. Let $r : V \to \bigcup_{a\in\Sigma} Q_a$ be a mapping and $x \in V$ be a node of t. Then r *satisfies the run condition of* $\mathcal{A}$ *at* x *(relative to* t*)* if

$$r(x) \in \delta_{\lambda(x),\mathrm{R}(x)}((r\partial_b(x))_{b\in\mathrm{R}(x)}).$$

The mapping r is a *run of* $\mathcal{A}$ *on* t if it satisfies the run condition for any $x \in V$. Note that, for a run r and $x \in V$, we always have $r(x) \in Q_{\lambda(x)}$ since the image of $\delta_{\lambda(x),\mathrm{R}(x)}$ belongs to $Q_{\lambda(x)}$.

Although the transitions of a Σ-ACM $\mathcal{A}$ are modeled by functions $\delta_{a,J}$, we can think of them as tuples $(q, (p_b)_{b\in J})$ with $q \in \delta_{a,J}((p_b)_{b\in J})$. Such a tuple can be understood as a directed acyclic graph with node set $\{q, p_b \mid b \in J\}$ and edges from p_b to q for $b \in J$. Furthermore, the nodes are labeled by elements of $\Sigma \times Q$ where q gets the label (a, q) and p_b is labeled by (b, p_b). Note that on the other side a run r on a Σ-dag $t = (V, E, \lambda)$ defines a $(\Sigma \times Q)$-labeled dag by $t' = (V, E, \lambda \times r)$. Then r is a run iff for any $y \in V$, the restriction of t' to y and its lower neighbors is a transition, i.e. if t' can be "tiled" by the transitions.

Now let r be a run on the Σ-dag $t = (V, E, \lambda)$. It is *successful* provided there exists a tuple $(q_a)_{a\in\lambda(V)} \in F$ with

$$r(\max(\lambda^{-1}(a))) \sqsupseteq_a q_a \text{ for all } a \in \lambda(V),$$

i.e. if the "global final state" $(r(\max\{\lambda^{-1}(a)\}))_{a\in\lambda(V)}$ dominates some accepting state in the direct product of the wqos $\sqsubseteq_a$. Let $L(\mathcal{A}) \subseteq \mathbb{D}(\Sigma)$ denote the set

of all Σ-dags that admit a successful run of $\mathcal{A}$. Let M be a set of Σ-dags and $L \subseteq M$. Then we say that L *can be accepted by a Σ-ACM relative to M* if there is a Σ-ACM $\mathcal{A}$ with $L(\mathcal{A}) \cap M = L$. Sometimes we will omit the term "relative to M" if the set M is clear from the context or if M is the set of all Σ-dags.

Example 2 (continued) Let Ha denote the set of Hasse-diagrams of pomsets without autoconcurrency. Then Ha $\subseteq \mathbb{D}(\Sigma)$. Furthermore, let $\mathcal{A}$ denote the Σ-ACM defined above. We show that $L(\mathcal{A}) =$ Ha: For a Hasse-diagram $(V, \multimap, \lambda) \in$ Ha, let $r(x)(a)$ be the number of a-labeled elements below x (possibly including x, cf. Figure 2 for an example where a tuple (x, y, z) denotes the function $\{(a, x), (b, y), (c, z)\}$). For $x \in V$, the reading domain $\mathrm{R}(x)$ is the set of labels of vertices covered by x. Thus, the vertices $\partial_c(x)$ for $c \in \mathrm{R}(x)$ are mutually incomparable. Hence, for $c \in \mathrm{R}(x)$, the vertex $\partial_c(x)$ dominates the largest number of c-labeled vertices among $\{\partial_d(x) \mid d \in \mathrm{R}(x)\}$. Hence $r(\partial_c(x))(c) > r(\partial_d(x))(c)$ for $d \in \mathrm{R}(x) \setminus \{c\}$, i.e. r is a run of $\mathcal{A}$ on $(V, \multimap, \lambda)$. Since any tuple $(g_c)_{c \in J}$ dominates some state from F, it is accepting, i.e. Ha $\subseteq L(\mathcal{A})$. Conversely, let r be a successful run of $\mathcal{A}$ on the Σ-dag (V, E, λ). Let $\sqsubseteq := E^\star$. Then, for any $x \in V$, $c \in \mathrm{R}(x)$ and $a \in \Sigma$, we have $r(\partial_c(x))(a) \leq r(x)(a)$, i.e. $h_a : (V, \sqsubseteq) \to \mathbb{N}$ defined by $h_a(x) := r(x)(a)$ is monotone with respect to the partial order $(V, \sqsubseteq)$. Furthermore, by the definition of $\delta_{a,J}$, for any $x \in V$ and $c, d \in \mathrm{R}(x)$ with $c \neq d$ we have $h_c(\partial_c(x)) > h_c(\partial_b(x))$. Hence $\partial_c(x) \not\sqsubseteq \partial_b(x)$. Since we can similarly infer $\partial_b(x) \not\sqsubseteq \partial_c(x)$, the elements $\partial_c(x)$ and $\partial_b(x)$ are incomparable. Hence (V, E, λ) is a Hasse-diagram. Thus, the set of Hasse-diagrams can be accepted by a Σ-ACM with infinitely many states. It is not known whether finitely many states suffice. On the contrary, one can show (cf. [12, Lemma 4.1.3]) that the set of *not*-Hasse-diagrams can be accepted by a Σ-ACA, i.e. by a Σ-ACM with only finitely many states.

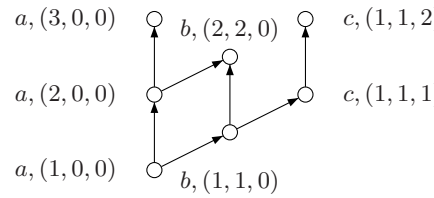


Fig. 2. A run of $\mathcal{A}$ (cf. Example 2)

Example 3. Let L be the set of all Σ-dags t satisfying "The number of d-labeled vertices of t is even for any $d \in \Sigma$". This set can be accepted by a Σ-ACM that differs from the ACM $\mathcal{A}$ from Example 2 only in the wqos $\sqsubseteq_a$: Here, we define $f \sqsubseteq_a g$ iff $f(b) \leq g(b)$ for $b \in \Sigma$ and $f(a) \equiv g(a) \mod 2$. Then a tuple $(f_c)_{c \in \Sigma}$ dominates some accepting state iff $f_c(c)$ is even for all $c \in J$.

Next, we define when a Σ-ACM is monotone. Intuitively, this means that increasing the input of a transition does not disable the transition and increases its output. A Σ-ACM is *monotone* if, for any $a \in \Sigma$, $J \subseteq \Sigma$, $p_b, p'_b \in Q_b$ for $b \in J$ and $q \in Q_a$, we have

$$q \in \delta_{a,J}((p_b)_{b\in J}) \text{ and } p_b \sqsubseteq_b p'_b \text{ for } b \in J \Longrightarrow \exists q' \in \delta_{a,J}((p'_b)_{b\in J}) : q \sqsubseteq_a q'.$$

Since the local wqos $\sqsubseteq_a$ of an asychronous cellular automaton are the diagonal relation Δ_{Q_a}, any Σ-ACA is monotone.

Example 4. In this example, we consider monotone ACMs that run on words over Σ. To do this, we identify a word over Σ with the Hasse-diagram of a linearly ordered Σ-labeled poset. In this sense, we can show that the non-regular "word-language" $\{a^m b^n \mid 1 \leq n < m\}$ can be accepted by a monotone ACM: The local states of $\mathcal{A}$ are the natural numbers. In a run, the ACM $\mathcal{A}$ first counts the occurrences of a in the a-component. Once a b is read, the ACM copies the a-counter into its b-component and starts to count down. If at the end the b-counter contains at least the value 1, the run is accepting. Otherwise, the b-counter has the value 0 and the ACM rejects. This in particular implies that monotone ACMs are more powerful than finite automata since ACMs can have infinite sets of states.

On the other hand, the set $L = \{b^n a^m \mid 1 \leq n < m\}$ cannot be accepted by a monotone Σ-ACM: Suppose $\mathcal{A}$ accepts L among all linearly ordered Σ-dags. For $n \in \mathbb{N}$, let q_n denote the state that is reached in the b-component by a successful run of $\mathcal{A}$ on $b^n a^{n+1}$ after performing the b-prefix. Since $\sqsubseteq_b$ is a wqo, there are $n < m$ with $q_n \sqsubseteq_b q_m$. Since $\mathcal{A}$ is monotone, one can extend the restriction of its successful run on $b^m a^{m+1}$ to the b-prefix to a successful run on $b^m a^{n+1} \notin L$. Thus, the set of languages acceptable by a monotone ACM is not closed under reversal. This might be surprising at first glance, but it is not really so since the notion of well quasi ordering as well as that of monotonicity are not symmetric.

4 Notational Conventions

Let $\mathbb{N}^+ = \{1, 2, \dots\}$. Nevertheless, an expression $\sup(M)$ for $M \subseteq \mathbb{N}^+$ will be understood in the structure $(\mathbb{N}, \leq)$. The useful effect of this convention is that $\sup(M) = 0$ for $M \subseteq \mathbb{N}^+$ if and only if M is empty.

Let A be a set. Then, a *word* is a mapping $w : M \to A$ where M is a finite subset of $\mathbb{N}^+$. If $M = \{n_1, n_2, \dots, n_k\}$ with $n_1 < n_2 < \cdots < n_k$, the finite sequence $w(n_1)w(n_2)\dots w(n_k)$ is a word in the usual sense. Two words $v : M \to A$ and $w : N \to A$ are *isomorphic* (and we will identify them) if there is an order isomorphism (with respect to the usual linear order of the natural numbers) $\eta : M \to N$ with $v = w \circ \eta$. By $A^\star$ we denote the set of all words over A. Furthermore, for $w \in A^\star$ and $a \in A$, let wa denote the word $v : \operatorname{dom} w \cup \{n\} \to A$ with $n > \operatorname{dom} w$, $v \restriction \operatorname{dom} w = w$ and $v(n) = a$. By ε, we denote the empty word, i.e. the mapping $\varepsilon : \emptyset \to A$.

Recall that we identify isomorphic Σ-dags. Hence, we can impose additional requirements on the carrier set V as long as they can be satisfied in any isomorphism class. It turns out that in our considerations, it will be convenient to assume that for any Σ-dag (V, E, λ)

$$V \subseteq \mathbb{N}^+ \text{ and } E^\star \text{ is contained in the usual linear order on } \mathbb{N}^+.$$

Note that on the set $\lambda^{-1}(a)$ we have two linear orders: $E^\star$ and the order $\leq$ of the natural numbers. Since $\leq$ extends $(V, E^\star)$, these two linear orders on $\lambda^{-1}(a)$ coincide. Hence, for any run r of some Σ-ACM on a Σ-dag $t = (V, E, \lambda)$, the mapping $r \restriction \lambda^{-1}(a) : \lambda^{-1}(a) \to Q_a$ is a word over Q_a whose letters occur in the order given by $(V, E^\star)$.

5 From Σ-ACMs to Transition Systems

A *transition system* is a set S endowed with a binary relation $\to \subseteq S^2$. In this section, we will derive from a Σ-ACM $\mathcal{A}$ a transition system $(S, \to)$ that reflects the computations of the Σ-ACM.

So let $\mathcal{A} = ((Q_a, \sqsubseteq_a)_{a\in\Sigma}, (\delta_{a,J})_{a\in\Sigma, J\subseteq\Sigma}, F)$ be some Σ-ACM. The runs of $\mathcal{A}$ correspond to states of the transition system $(S, \to)$. Before defining the set S formally, we first formalize this correspondence: Let $t = (V, E, \lambda)$ be a Σ-dag and let $r : V \to Q$ be a run of $\mathcal{A}$ on t. For $a \in \Sigma$, let $v_a := r \restriction \lambda^{-1}(a)$. As explained above, $\lambda^{-1}(a)$ is a subset of $\mathbb{N}^+$ where the order relation $E^\star$ coincides with the usual linear order $\leq$ on $\mathbb{N}$. Hence $v_a : \lambda^{-1}(a) \to Q_a$ is a word over Q_a. Now we define mappings $\mathrm{pos}^v_a : \Sigma \to V$ as follows: For $a, b \in \Sigma$, let $\mathrm{pos}^v_a(b)$ denote the last position in the word v_a that is read by some b-labeled vertex. Formally

$$\mathrm{pos}^v_a(b) := \sup\{x \in \lambda^{-1}(a) \mid \exists y \in \lambda^{-1}(b) : (x, y) \in E\}$$

where the supremum is taken in $\mathbb{N}$ such that, if the set is empty, we have $\mathrm{pos}^v_a(b) = 0$. Note that $\mathrm{pos}^v_a(b)$ is in general not the last position in v_a that is *dominated* by some b-labeled vertex in the *partial order* $(V, E^\star, \lambda)$. The tuple $v = (v_a, \mathrm{pos}^v_a)_{a\in\Sigma}$ is called the *state associated with the run* r, denoted $\mathrm{state}(r) := (v_a, \mathrm{pos}^v_a)_{a\in\Sigma}$.

Example 2 (continued) Let $t = (V, E, \lambda)$ be the Σ-dag and let r denote the run of $\mathcal{A}$ depicted in Figure 2. Then we have the following:

$$\begin{array}{ll} v_a = (1,0,0)(2,0,0)(3,0,0) & \mathrm{pos}^v_a = \{(a,2),(b,2),(c,0)\} \\ v_b = (1,1,0)(2,2,0) & \mathrm{pos}^v_b = \{(a,0),(b,1),(c,1)\} \\ v_c = (1,1,1)(1,1,2) & \mathrm{pos}^v_c = \{(a,0),(b,0),(c,1)\} \end{array}$$

This situation is visualized in Figure 3. There, the words v_a, v_b and v_c are drawn vertically. On the left of a node, the associated state of $\mathcal{A}$ can be found. The letter b at the right of the second a-node indicates that this node equals $\mathrm{pos}^v_a(b)$. Finally, $\mathrm{pos}^v_a(c) = 0$ is indicated by the fact that c does not appear at the right of the word v_a.

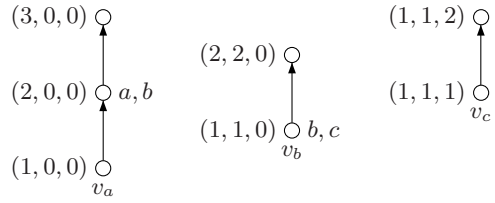


Fig. 3. The state state(r) of the run from Figure 2

As explained above, we want the set of states S to contain state(r). This is achieved by the following definition of the state set S:

$$S := \left\{ (v_a, \mathrm{pos}_a^v)_{a\in\Sigma} \in \prod_{a\in\Sigma} (Q_a^\star \times \mathbb{N}^\Sigma) \mid \mathrm{im}(\mathrm{pos}_a^v) \subseteq \mathrm{dom}\, v_a \cup \{0\} \text{ for } a \in \Sigma \right\}.$$

Next, we describe the transition relation $\to \subseteq S^2$ that is meant to reflect the computation steps of the Σ-ACM $\mathcal{A}$. Let $t' = (V', E', \lambda)$ be a Σ-dag and let $x \in \max(V', (E')^\star)$ be a maximal element of V'. Then $t = (V' \setminus \{x\}, E' \setminus (V' \times \{x\}), \lambda)$ is the restriction of t' to $V' \setminus \{x\}$ and therefore another Σ-dag. We consider t' as an extension of t by one node x. Now, let r' be a run of $\mathcal{A}$ on t' and let r be the restriction of r' to t. Then, intuitively, r' is the extension of the run r by one computational step of the Σ-ACM $\mathcal{A}$. Therefore, we want the state $(w_a, \mathrm{pos}_a^w)_{a\in\Sigma} := \mathrm{state}(r') \in S$ to be a successor of the state $(v_a, \mathrm{pos}_a^v)_{a\in\Sigma} := \mathrm{state}(r)$. One can show (cf. the example below) that there exist $a \in \Sigma$, $\emptyset \neq J \subseteq \Sigma$, $p_b \in Q_b$ for $b \in J$ and $q \in Q_a$ such that

(i) $q \in \delta_{a,J}((p_b)_{b\in J})$,
(ii) $w_c = \begin{cases} v_c q & \text{for } c = a \\ v_c & \text{otherwise,} \end{cases}$
(iii) $\mathrm{pos}_c^w(b) = \mathrm{pos}_c^v(b)$ for all $b, c \in \Sigma$ satisfying either $c \notin J$ or $a \neq b$, and
(iv) $\mathrm{pos}_c^v(a) < \mathrm{pos}_c^w(a) \in \mathrm{dom}\, v_c$ such that $v_c \circ \mathrm{pos}_c^w(a) = p_c$ for $c \in J$.

These four properties constitute the relation $\to$, i.e. for states $(v_a, \mathrm{pos}_a^v)_{a\in\Sigma}$ and $(w_a, \mathrm{pos}_a^w)_{a\in\Sigma}$ from S, we define $(v_a, \mathrm{pos}_a^v)_{a\in\Sigma} \to (w_a, \mathrm{pos}_a^w)_{a\in\Sigma}$ iff there exist $a \in \Sigma$, $\emptyset \neq J \subseteq \Sigma$, $p_b \in Q_b$ for $b \in J$ and $q \in Q_a$ such that (i)-(iv) hold. Let $\mathcal{S}(\mathcal{A}) = (S, \to)$ denote the *transition system associated to* $\mathcal{A}$.

Example 2 (continued) Let t' denote the extension of the Σ-dag t from Figure 2 by an a-labeled node as indicated in Figure 4 (left picture). Furthermore, this picture shows an extension r' of the run r, too. The second picture depicts the state state(r'). The reader might check that state(r') is a successor state of state(r).

It is not surprising that the set S contains many states which do not correspond to runs of $\mathcal{A}$, i.e. there are $(v_a, \mathrm{pos}_a^v)_{a\in\Sigma} \in S$ such that for no run r of $\mathcal{A}$,

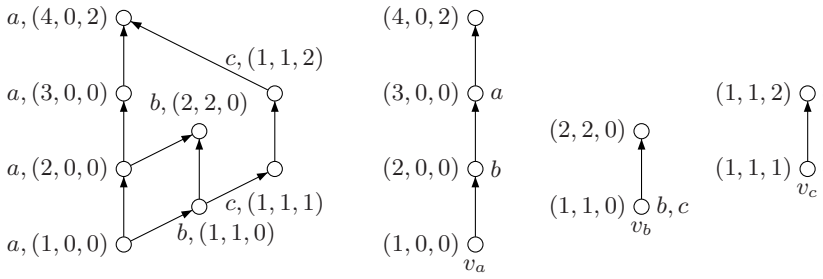


Fig. 4. A successor state of state(r) from Figure 3

we have $\text{state}(r) = (v_a, \text{pos}_a^v)_{a\in\Sigma}$. Our next aim is to identify the states of the form state(r) for some run r on some Σ-dag t inside the transition system $\mathcal{S}(\mathcal{A})$. To begin, one can easily show that a state from S corresponds to a run on an antichain iff it is a depth-1-state defined as follows: A state $(w_a, \text{pos}_a^w)_{a\in\Sigma} \in S$ is a *depth-1-state* if $|w_a| \leq 1$, $\text{pos}_a^w(b) = 0$, and $w_c(\min \text{dom}(w_c)) \in \delta_{c,\emptyset}$ for $a, b, c \in \Sigma$ with $w_c \neq \varepsilon$. The idea now is that a state is of the form state(r) if it can be reached from some depth-1-state in the transition system $\mathcal{S}(\mathcal{A})$. Unfortunately, this is not true in general: Let $t = (V, E, \lambda)$ be some Σ-dag and r a run on t. Furthermore, suppose $(v_a, \text{pos}_a^v)_{a\in\Sigma} = \text{state}(r) \to (w_a, \text{pos}_a^w)_{a\in\Sigma}$. Hence there exist $a \in \Sigma$, $\emptyset \neq J \subseteq \Sigma$, $p_b \in Q_b$ for $b \in J$ and $q \in Q_a$ such that (i)-(iv) hold. Now, one can try to add one maximal node to t and connect it to the nodes of t that correspond to the states p_b for $b \in J$. The problem now is that the new a-labeled node need not be comparable with the old a-labeled nodes from t. The solution of this problem is to consider not the transition system $\mathcal{S}(\mathcal{A})$, but finitely many transition systems $\mathcal{S}(\mathcal{A} \times \mathcal{A}_i)$:

Lemma 1. *There exists an algorithm that on input of an alphabet Σ, outputs finitely many Σ-ACAs $\mathcal{A}_i$ $(1 \leq i \leq |\Sigma^\Sigma| =: n)$, such that for any Σ-ACM $\mathcal{A}$ and any $1 \leq i \leq n$, we have*

1. $\bigcup_{1\leq i\leq n} L(\mathcal{A}_i) = \mathbb{D}(\Sigma)$, *and*
2. *A state $(w_a, \text{pos}_a^w)_{a\in\Sigma}$ from $\mathcal{S}(\mathcal{A}\times\mathcal{A}_i)$ is of the form* state(r) *for some run r of $\mathcal{A} \times \mathcal{A}_i$ iff it can be reached from some depth-1-state in the transition system $\mathcal{S}(\mathcal{A} \times \mathcal{A}_i)$.*

As usual, the direct product of Σ-ACMs accepts the intersection of the two accepted languages. Since any Σ-dag is accepted by at least one of the Σ-ACAs $\mathcal{A}_i$, the Σ-ACM $\mathcal{A}$ accepts some Σ-dag iff there exists $1 \leq i \leq n$ such that $L(\mathcal{A} \times \mathcal{A}_i) \neq \emptyset$. Thus, in order to decide the emptiness of $L(\mathcal{A})$, it suffices to decide whether $L(\mathcal{A} \times \mathcal{A}_i)$ is empty. In other words, from now on we can assume that the Σ-ACM $\mathcal{A}$ satisfies:

(∗) A state $(w_a, \mathrm{pos}_a^w)_{a\in\Sigma}$ from $\mathcal{S}(\mathcal{A})$ is of the form state(r) for some run r of $\mathcal{A}$ iff it can be reached from some depth-1-state in the transition system $\mathcal{S}(\mathcal{A})$.

Under this general assumption, it remains to deal with the following two questions: "Which states from $\mathcal{S}(\mathcal{A})$ correspond to successful runs of $\mathcal{A}$?" and "How can we decide whether such a state can be reached from some depth-1-state?" The answers to these two questions are based on the notion of a well-structured transition system that we consider in the following section.

6 Well-Structured Transition Systems

Let $(S, \to)$ be a transition system. For $t \in S$, we denote by $\mathrm{Pred}(t)$ the set of *predecessors* of t, i.e. the set of all $s \in S$ with $s \to t$. A *well-structured transition system* or *WSTS* is a triple $(S, \to, \preceq)$ where $(S, \to)$ is a transition system, $\preceq$ is a wqo on S and for any $s, s', t \in S$ with $s \to t$ and $s \preceq s'$ there exists $t' \in S$ with $s' \to t'$ and $t \preceq t'$. Thus, a WSTS is a well-quasi ordered transition system such that any transition $s \to t$ "lifts" to a larger state $s' \succeq s$ (cf. Figure 5). This

$$\begin{array}{ccc} s' & \longrightarrow & t' \\ \curlyvee| & & \curlyvee| \\ s & \longrightarrow & t \end{array}$$

Fig. 5. Lifting of a transition in a WSTS

definition differs slightly from the original one by Finkel & Schnoebelen [7] in two aspects: First, they require only $s' \to^\star t'$ and they call WSTS' satisfying our axiom "WSTS with strong compatibility". Secondly, and more seriously, their transition systems are finitely branching. But it is easily checked that the results from [7, Section 2 and 3] hold for infinitely branching transition systems, too. Since we use only these results (namely [7, Theorem 3.6]) here, it is not necessary to restrict well-structured transition systems to finitely branching ones. In [7], several decidability results are shown for WSTS. In particular, they showed that the "coverability" is decidable for any WSTS $(S, \to, \preceq)$ if $\preceq$ is decidable and a finite basis of $\mathrm{Pred}(\uparrow s) = \bigcup_{t\succeq s} \mathrm{Pred}(t)$ can be computed effectively. Since in their proof the decision algorithm is uniformly constructed from the decision algorithm for $\preceq$ and the algorithm that computes a finite predecessor basis, one gets even more:

Theorem 1 (cf. [7, Theorem 3.6]). *There exists an algorithm that solves the following decision problem:*

input: *1. an algorithm that decides* $\preceq$,
2. an algorithm computing a finite basis for $\mathrm{Pred}(\uparrow s)$ *for* $s \in S$, *and*
3. two states s *and* t *from* S
for some well-structured transition system $(S, \to, \preceq)$.
output: *Does there exist a state* $t' \in S$ *such that* $s \to^\star t' \succeq t$?

To use this theorem, we have to define a wqo on the set of states from $\mathcal{S}(\mathcal{A})$. This wqo is based on a generalization of the subword relation defined as follows: Recall that we consider words as mappings from a finite linear order into the well-quasi ordered set Q_a. Therefore, an *embedding* $\eta : v \hookrightarrow w$ is defined to be an order embedding of $\mathrm{dom}\, v \cup \{0\}$ into $\mathrm{dom}\, w \cup \{0\}$ such that

$$\eta(0) = 0,\ \eta(\sup \mathrm{dom}\, v) = \sup \mathrm{dom}\, w,\ \text{and } v(i) \sqsubseteq_a w \circ \eta(i) \text{ for } i \in \mathrm{dom}\, v.$$

Thus, there is an embedding $\eta : v \hookrightarrow w$ iff one obtains v from w by first deleting some letters (but not the last) and then decreasing the remaining ones with respect to $\sqsubseteq_a$. If $\sqsubseteq_a$ is trivial (i.e. the identity relation Δ_{Q_a}), there exists such an embedding iff v is a subword of w and the last letters of v and w coincide. Now a quasi-order $\preceq$ on the states of the transition system $(S, \to)$ is defined by $(v_a, \mathrm{pos}_a^v)_{a\in\Sigma} \preceq (w_a, \mathrm{pos}_a^w)_{a\in\Sigma}$ iff

there exist embeddings $\eta_a : v_a \hookrightarrow w_a$ such that $\eta_a \circ \mathrm{pos}_a^v = \mathrm{pos}_a^w$ for any $a \in \Sigma$.

As explained above, the existence of the embeddings η_a ensures that v_a is dominated by some subword (including the last letter) of w_a letter by letter. The requirement $\eta_a \circ \mathrm{pos}_a^v = \mathrm{pos}_a^w$ ensures that the pointer $\mathrm{pos}_a^w(b)$ (if not 0) points to some position in this *subword* and that this position corresponds (via η_a) to the position in v_a to which $\mathrm{pos}_a^v(b)$ points. It is obvious that $\preceq$ is reflexive and transitive, i.e. $\preceq$ is a quasiorder.

Using Higman's Theorem [10], one can show that indeed $\preceq$ is a wqo on S. To obtain that $(S, \to, \preceq)$ is a WSTS, we have in addition to assume that $\mathcal{A}$ is monotone. Then one can show:

Theorem 2. *Let* $\mathcal{A}$ *be a monotone* Σ*-ACM. Then* $\mathcal{S}(\mathcal{A}) = (S, \to, \preceq)$ *is a well-structured transition system.*

To apply Theorem 1 to the WSTS $(S, \to, \preceq)$, our next aim is to show that in $(S, \to, \preceq)$ a finite predecessor basis, i.e. a finite basis of $\mathrm{Pred}(\uparrow(x_c, \mathrm{pos}_c^x)_{c\in\Sigma})$, can be computed for any state $(x_c, \mathrm{pos}_c^x)_{c\in\Sigma}$. Since, in general, this is not the case, we have to introduce another notion: We call a Σ-ACM *effective* if there is an algorithm that given $a \in \Sigma$, $J \subseteq \Sigma$, $p_b \in Q_b$ for $b \in J$ and $q \in Q_a$ computes a finite basis of the set of all tuples $(q', (p'_b)_{b\in J}) \in \prod_{b\in J} Q_b \times Q_a$ satisfying

$$q' \in \delta_{a,J}((p'_b)_{b\in J}),\ q \sqsubseteq_a q' \text{ and } p_b \sqsubseteq_b p'_b \text{ for } b \in J$$

with respect to the direct product $\sqsubseteq_a \times \prod_{b\in J} \sqsubseteq_b$. We call such an algorithm a *basis algorithm of* $\mathcal{A}$. Intuitively, an ACM is effective if a finite basis of all

transitions above a given tuple of states can be computed. Note that this tuple is not necessarily a transition. On the other hand, we do not require that the set of all transitions, i.e. the set $\{(q,(p_b)_{b\in J}) \mid q \in \delta_{a,J}((p_b)_{b\in J})\}$ is a recursive subset of $Q_a \times \prod_{b\in J} Q_b$, and this might not be the case as the following example shows. Furthermore note that any asynchronous cellular automaton is effective since (as a finite object) it can be given explicitly.

Example 5. Let $\Sigma = \{a\}$ and $Q_a = \mathbb{N}$. On this set, we consider the complete relation $\mathbb{N} \times \mathbb{N}$ as wqo $\sqsubseteq_a$. Furthermore, let M be some non recursive subset of $\mathbb{N}$ and define, for $n \in \mathbb{N}$:

$$\delta_{a,\{a\}}(n) = \begin{cases} \{n, n+1\} & \text{if } n \in M \\ \{n\} & \text{if } n \notin M. \end{cases}$$

Furthermore, let $\delta_{a,\emptyset} = \{1\}$. Now let $t = (V, E, \lambda)$ be a Σ-dag (i.e. t is the Hasse-diagram of a finite linear order) and let $r : V \to \mathbb{N}$ be some mapping. Then r is a run of the Σ-ACA $\mathcal{A} = (Q_a, (\delta_{a,J})_{J\subseteq\{a\}}, F)$ iff $r(x) \le r(x+1) \le r(x)+1$ for any $x \in V$ and $\{x \in V \mid r(x) \ne r(x+1)\} \subseteq M$. Since this latter inclusion is not decidable, one cannot decide whether r is a run. On the other hand, $\mathcal{A}$ is effective since $\{(1,1)\}$ is a finite basis of any nonempty subset of $Q_a \times Q_a$.

The preceding example suggests that $L(\mathcal{A})$ is recursive for any monotone and effective Σ-ACM $\mathcal{A}$. Later (Corollary 2), we will show that this is indeed the case.

So let $(x_c, \mathrm{pos}_c^x)_{c\in\Sigma}$ be a state from S. Then one can show that there exists a basis of $\mathrm{Pred}(\uparrow(x_c, \mathrm{pos}_c^x)_{c\in\Sigma})$ such that any state $(v_c, \mathrm{pos}_c^v)_{c\in\Sigma}$ in this basis satisfies $|v_c| - |x_c| \le 2(|\Sigma|+1)$, i.e. the length of the words v_c is bounded. Although there are infinitely many such words, the proof of the following lemma relies on this observation. This is possible since this set of words has a finite basis.

Lemma 2. *There exists an algorithm that computes the following function:*
input: *1. an alphabet Σ,*
2. a basis algorithm of an effective and monotone Σ-ACM $\mathcal{A}$,
3. a finite basis B_c of $(Q_c, \sqsubseteq_c)$ and an algorithm to decide $\sqsubseteq_c$ for $c \in \Sigma$, and
4. a state $(x_c, \mathrm{pos}_c^x)_{c\in\Sigma} \in S$
output: *a finite basis of the set* $\mathrm{Pred}(\uparrow(x_c, \mathrm{pos}_c^x)_{c\in\Sigma})$.

By Theorem 2 and Lemma 2, we can apply Theorem 1 to the transition system $\mathcal{S}(\mathcal{A}) = (S, \to, \preceq)$: There is an algorithm that, given a monotone and effective Σ-ACM $\mathcal{A}$ and a state $(w_c, \mathrm{pos}_c^w)_{c\in\Sigma}$ from S, decides whether $(w_c, \mathrm{pos}_c^w)_{c\in\Sigma}$ is dominated by some reachable state in the WSTS $\mathcal{S}(\mathcal{A})$. It remains to transfer this decidability to the question whether the language accepted by $\mathcal{A}$ is empty:

Let B_c be a finite basis of the set of local states Q_c of the Σ-ACM $\mathcal{A}$ for $c \in \Sigma$. Now let $J \subseteq \Sigma$ and let q_c be some local state of $\mathcal{A}$ for $c \in J$. We define $\mathrm{States}((q_c)_{c\in J})$ to consist of all states $(w_c, \mathrm{pos}_c^w)_{c\in\Sigma}$ from $\mathcal{S}(\mathcal{A})$ such that for all

$c \in \Sigma$:

$$|w_c| \leq |\Sigma|,\ (w_c = \varepsilon \iff c \notin J),\ \text{and}\ w_c \in B_c^\star q_c \ \text{for}\ c \in J.$$

Note that the set $\mathrm{States}((q_c)_{c\in J})$ is finite due to the restrictions $|w_c| \leq |\Sigma|$ and $w_c \in B_c^\star q_c \cup \{\varepsilon\}$. Since, in addition, the set F of accepting states of $\mathcal{A}$ is finite, we even have that $\bigcup_{\bar{q}\in F} \mathrm{States}(\bar{q})$ is finite. The following lemma states that $L(\mathcal{A})$ is not empty iff some state of this finite set $\bigcup_{\bar{q}\in F} \mathrm{States}(\bar{q})$ is dominated by a state in $\mathcal{S}(\mathcal{A})$ that is reachable from a depth-1-state.

Lemma 3. *Let $\mathcal{A}$ be a Σ-ACM satisfying $(*)$ and let $\mathcal{S}(\mathcal{A}) = (S, \to, \preceq)$. Then the following are equivalent:*

1. *$\mathcal{A}$ accepts some Σ-dag, i.e. $L(\mathcal{A}) \neq \emptyset$.*
2. *There exist an accepting state $(q_a)_{a\in J}$ of $\mathcal{A}$, a depth-1-state $(v'_a, \mathrm{pos}_a^{v'})_{a\in\Sigma}$ from S, a state $(w_a, \mathrm{pos}_a^{w})_{a\in\Sigma} \in \mathrm{States}((q_a)_{a\in J})$ and a state $(w'_a, \mathrm{pos}_a^{w'})_{a\in\Sigma}$ in S such that $(v'_a, \mathrm{pos}_a^{v'})_{a\in\Sigma} \to^\star (w'_a, \mathrm{pos}_a^{w'})_{a\in\Sigma} \succeq (w_a, \mathrm{pos}_a^{w})_{a\in\Sigma}$.*

Summarizing the results of this section, finally we show that the emptiness of effective and monotone Σ-ACMs is uniformly decidable:

Theorem 3. *There exists an algorithm that solves the following decision problem:*
input: *1. an alphabet Σ,*
2. a basis algorithm of an effective and monotone Σ-ACM $\mathcal{A}$,
3. the set of final states F of $\mathcal{A}$,
4. a finite basis of $(Q_c, \sqsubseteq_c)$, and an algorithm to decide $\sqsubseteq_c$ for $c \in \Sigma$.
output: *Is $L(\mathcal{A})$ empty?*

Proof. As explained at the end of Section 5, is suffices to deal with Σ-ACMs $\mathcal{A}$ that satisfy the condition $(*)$. In addition, we can assume that there is $a \in \Sigma$ such that $\delta_{a,\emptyset} = \{\bot\}$ and $\delta_{c,\emptyset} = \emptyset$ for $c \neq a$. Then there is only one depth-1-state $(v_c, \mathrm{pos}_c^{v})_{c\in\Sigma}$.

By Theorem 2, $\mathcal{S}(\mathcal{A}) = (S, \to, \preceq)$ is a WSTS. Using the algorithms that decide the wqos of local states in $\mathcal{A}$, one gets a decision procedure for $\preceq$. By Lemma 2, a finite basis of the set $\mathrm{Pred}(\uparrow(w_c, \mathrm{pos}_c^{w})_{c\in\Sigma})$ can be computed effectively for any state $(w_c, \mathrm{pos}_c^{w})_{c\in\Sigma} \in S$. Hence, by Theorem 1 the set of states that are dominated by a state reachable from $(v_c, \mathrm{pos}_c^{v})_{c\in\Sigma}$ is recursive. Since $\bigcup_{\bar{q}\in F} \mathrm{States}(\bar{q})$ is finite, the result follows from Lemma 3 and Theorem 1. □

If we restrict attention to asynchronous cellular automata (where the set of accepting states is finite), this theorem translates into the following:

Corollary 1. *There exists an algorithm that solves the following decision problem:*
input: *An alphabet Σ and a Σ-ACA $\mathcal{A}$.*
output: *Is $L(\mathcal{A})$ empty?*

One might consider this corollary as the main contribution of the present paper. This statement on *finite* devices is proved using *infinite* devices. Of course,

one can restrict our proof of Theorem 3 to asynchronous cellular automata. But this does not eliminate the *infinite* well-structured transition systems. I am doubtful that infinite structures can be eliminated from the proof completely. The reason is that at any stage of a computation of an ACA, it still may read the very first local state of one of its components.

Another consequence of Theorem 3 is that for any monotone and effective Σ-ACM $\mathcal{A}$ membership in $L(\mathcal{A})$ is decidable:

Corollary 2. *Let $\mathcal{A}$ be a monotone and effective Σ-ACM. Then the set $L(\mathcal{A})$ is recursive.*

7 Open Problems

Since there are uncountably many Σ-ACMs, one cannot expect that the emptiness is decidable in general. This explains why we had to restrict to effectice ACMs. On the other hand, the monotonicity originates only in our proof using well structured transition systems. It is not clear whether this is really needed for the result on asynchronous cellular machines. Furthermore, it is an open question whether the emptiness is decidable when restricted to Hasse-diagrams. The results on ACAs from [3,11,4] easily extend to infinite posets using trace-ACAs that accept infinite Mazurkiewicz traces. It is not clear, whether the result presented here can be extended in the same direction.

References

1. J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960. 536
2. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000. 537
3. M. Droste and P. Gastin. Asynchronous cellular automata for pomsets without autoconcurrency. In *CONCUR'96*, Lecture Notes in Comp. Science vol. 1119, pages 627–638. Springer, 1996. 536, 537, 550
4. M. Droste, P. Gastin, and D. Kuske. Asynchronous cellular automata for pomsets. *Theoretical Comp. Science*, 2000. To appear. 537, 550
5. V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publ. Co., 1995. 536
6. W. Ebinger and A. Muscholl. Logical definability on infinite traces. *Theoretical Comp. Science*, 154:67–84, 1996. 536
7. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! Technical Report LSV-98-4, ENS Cachan, 1998. To appear in Theoretical Computer Science. 536, 537, 546
8. P. Gastin and A. Petit. Asynchronous automata for infinite traces. In *19th ICALP*, Lecture Notes in Comp. Science vol. 623, pages 583-594, Springer, 1992.
9. J.L Gischer. The equational theory of pomsets. *Theoretical Comp. Science*, 61:199–224, 1988. 536
10. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, 2:326–336, 1952. 547

11. D. Kuske. Asynchronous cellular automata and asynchronous automata for pomsets. In *CONCUR'98*, Lecture Notes in Comp. Science vol. 1466, pages 517–532. Springer, 1998. 536, 537, 550
12. D. Kuske. Contributions to a trace theory beyond Mazurkiewicz traces. Submitted manuscript, TU Dresden, 1999. 537, 541
13. V. Pratt. Modelling concurrency with partial orders. *Int. J. of Parallel Programming*, 15:33–71, 1986. 536
14. P. H. Starke. Processes in Petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 17:389–416, 1981. 536
15. W. Zielonka. Notes on finite asynchronous automata. *R. A. I. R. O. - Informatique Théorique et Applications*, 21:99–135, 1987. 536

Revisiting Safety and Liveness in the Context of Failures*

Bernadette Charron-Bost[1], Sam Toueg[2], and Anindya Basu[3]

[1] LIX, École Polytechnique, 91128 Palaiseau Cedex, France
charron@lix.polytechnique.fr
[2] DIX, École Polytechnique, 91128 Palaiseau Cedex, France
[3] Bell Labs, Lucent Technologies,
600 Mountain Avenue Murray Hill, NJ 07974-0636 USA

Abstract. Safety and liveness are two fundamental concepts for proving the correctness of concurrent programs. In the context of failures, however, we observe that some properties that are commonly believed to be safety properties are actually liveness properties. In this paper, we propose refinements of the concepts of safety and liveness that avoid this counterintuitive classification.

1 Introduction

Safety and liveness are two fundamental classes of properties [Pnu77, Lam83]. Informally, a safety property asserts that something bad does not happen, and a liveness property asserts that something good eventually happens. The starting point of our work is the observation that the presence of failures can introduce some counterintuitive phenomena in this classification.

To illustrate this point, consider the well-known problem of *consensus*, and systems with process crashes. In most papers in the literature, a *correct process* is defined to be a process that takes an infinite number of steps, and the *agreement* property of consensus is defined as "no two correct processes decide differently" [FLP85, DDS87, DLS88, CHT96, AW98]. It is commonly believed that this agreement property is a safety property (e.g., [DLS88]). Upon closer look, however, it becomes clear that it is actually a liveness property. Indeed, any partial run can be extended into a run[1] that satisfies the agreement property: even if two processes have decided differently by some time t, agreement can still be satisfied if one of the two processes crashes, i.e., takes no step after time t (such a process is not correct, and so it is exempt from the agreement property). So there is a discrepancy between the intuition that agreement should be a safety property and the fact that it is actually a liveness property. Basically, this discrepancy is due to the fact that at any point, the agreement property can always be achieved thanks to future crashes.

* Research partially supported by NSF grants CCR-9711403.

[1] In this paper, runs are infinite and partial runs are finite.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 552–565, 2000.

In this work, we propose concepts of safety and liveness that are appropriate to systems with failures in that they avoid the counterintuitive classification illustrated above. These concepts that we call *pure safety* and *pure liveness* are explained below.

First consider the concept of liveness. Roughly speaking, P is a liveness property if P says that at any point in a run, no matter what has happened up to that point, it is still possible for something good to eventually happen. In contrast, we say that P is a *pure liveness* property if P says that at any point in a run it is still possible for something good to eventually happen *and this can happen without the help of subsequent failures.* Intuitively, the motivation here is that liveness should not depend on future failures, which may or may never occur.

To illustrate the difference between liveness and pure liveness, consider the agreement property of consensus and a point in a run in which two processes have decided differently. From this point, something good may still happen to satisfy agreement — specifically, one (or both) processes may crash. Thus, as we pointed out before, agreement is a liveness property. In contrast, agreement is *not* a pure liveness property, because for something good to happen, i.e., for agreement to be satisfied, one of the two processes *must* crash at some point in the future.

To explain the concept of pure safety, we first recall the definition of safety. A partial run is *bad w.r.t some property* P, if it cannot be extended into a run that satisfies P. Property P is a *safety* property if every run that does not satisfy P has a prefix that is a bad partial run. Bad partial runs, however, are not the only undesirable ones with respect to P. Consider a partial run that cannot be extended to satisfy P without the occurrence of new failures. Intuitively, such a partial run is undesirable because failures are unpredictable, and we should not count on the occurrence of future failures to satisfy P. Formally, we say that a partial run is *undesirable w.r.t* P if every extension that satisfies P contains *additional* failures. This leads to our definition of pure safety: Property P is a *pure safety* property if every run that does not satisfy P has a prefix that is an undesirable partial run. Note that a bad partial run is also undesirable, and so pure safety is stronger than safety.

As exemplified by the agreement property of consensus, there exist liveness properties that are not pure liveness properties. Similarly, we can exhibit safety properties that are not pure safety properties. Pure safety and pure liveness are thus strict refinements of safety and liveness, respectively.

In the full paper, we first formalize the concepts of pure safety and pure liveness. To do so, we model a property as a set of runs and introduce a property transformer denoted $Pure$: roughly speaking, a property P is transformed into $Pure(P)$ by removing all the runs from P that contain an undesirable partial run. For example, for the agreement property of consensus, $Pure(agreement)$ specifies that if a process decides then its decision value is not different from the decision value of all the processes that have previously decided *and that are still alive*. In other words, $Pure(agreement)$ requires that at any time no two *alive*

processes have decided differently. Clearly, $Pure(agreement)$ is stronger than *agreement*, but weaker than *uniform agreement* — a property that requires that no two processes (whether correct or faulty) ever decide differently. The property transformer $Pure$ may thus lead to some new problem specifications that are better suited to systems with failures. A property P is *pure* if it is a fixed point of the property transformer $Pure$, i.e., $Pure(P) = P$. If, in addition, P is a safety (liveness) property, then we say that P is a *pure safety (pure liveness)* property.

We show that pure safety and pure liveness are exempt from the counter-intuitive classification of properties illustrated above. We prove a counterpart of the well-known result of Alpern and Schneider [AS85], specifically, we show that every pure property is the conjunction of a pure safety property and a pure liveness property. We also give alternative definitions of pure safety and pure liveness properties in terms of an appropriate closure operator, which is the counterpart of the topological closure operator used to define safety and liveness in [AS85].

2 Background

In this section, we describe the model and the notation that we use. We consider a collection of processes (automata) Π and a fixed set of actions Act. Each action $a \in Act$ is signed with the identity of the process which executes a. If p executes action a, then we denote $id(a) = p$. The crash failure of $p \in \Pi$ is modeled by including in the action set of p a $crash_p$ action, the effect of which is to permanently disable all subsequent actions of p. We consider the set Act^ω of all infinite sequences on Act. The i-th action in a sequence $\sigma \in Act^\omega$ is denoted by $\sigma[i]$. The set of crash actions that occur in a sequence σ is denoted by $Crash(\sigma)$. Executions of Π correspond to a sequence in the set

$$X = \{\sigma \in Act^\omega \mid \exists i,\ \sigma[i] = crash_p \Rightarrow \forall j > i,\ id(\sigma[j]) \neq p\}.$$

Let X^* denote the set of (finite) prefixes of sequences in X. If σ and β are in X and X^*, respectively, then $\beta \prec \sigma$ means that β is a prefix of σ and $\sigma \setminus \beta$ denotes the subsequence of σ consisting of all actions that are in σ but not in β.

A process p is *correct in* $\sigma \in X$ if $crash_p$ does not occur in σ.

A *property* specifies the set of executions which it allows. This leads one to define a property formally as a predicate on X, or equivalently as a subset of X (precisely the subset of sequences satisfying the property). A sequence σ that satisfies a property P (i.e., $\sigma \in P$) is called a *trace of* P. The set of properties on X is denoted by $\mathcal{P}$. Implication, conjunction, and disjunction of properties are trivially expressed by means of set inclusion, intersection, and union, respectively.

Most of the reasoning about distributed systems has been aimed at proving two types of properties: *safety* and *liveness*. Intuitively a safety property asserts that some "bad" thing never happens. We presume that if something bad happens in a sequence, then it is as a result of some particular actions in the

sequence. This yields the following definition: a finite sequence β in X^* is *bad with respect to a property* P if β has no extension in P, i.e.,

$$\forall \sigma \succ \beta \ : \ \sigma \notin P.$$

A property P is a *safety property* if P stipulates there is no bad (finite) sequence with respect to P, namely:

$$\forall \sigma \in X \ : \ \sigma \notin P \Rightarrow \exists \beta \prec \sigma, \ \forall \sigma' \succ \beta \ : \ \sigma' \notin P.$$

A property P is a *liveness property* if every finite sequence in X^* has some extension in P:

$$\forall \beta \in X^*, \exists \sigma \succ \beta \ : \ \sigma \in P.$$

A liveness property is often understood as saying that some "good" thing eventually happens: no matter what happens up to some point, it is still possible for the good thing to occur at some time in the future.

There is a natural (metrical) topology on X: the basic open sets are the sets of all sequences in X which share a common prefix. With regard to this topology, the *closure of* P is the set of sequences in X such that every prefix has an extension in P:

$$\overline{P} = \{\sigma \in X \mid \forall \beta \prec \sigma, \ \exists \sigma' \succ \beta \ : \ \sigma' \in P\}.$$

Interestingly, safety properties are exactly the closed sets (i.e., $\overline{P} = P$), and liveness properties are exactly the dense sets (i.e., $\overline{P} = X$).

In this paper, we will illustrate our definitions and concepts with the *agreement* property of consensus. This property stipulates that no two correct processes decide differently. More precisely, let V be a fixed set of values, and assume that the set of actions of each process p contains the actions $init_p(v)$ and $decide_p(v)$, for every $v \in V$. The *agreement* property is the set of sequences σ such that

$$\left.\begin{array}{l} \forall i, \ \sigma[i] \notin \{crash_p, crash_q\} \\ \exists k, \ \sigma[k] = decide_p(v) \\ \exists l, \ \sigma[l] = decide_q(v') \end{array}\right\} \Rightarrow v = v'$$

As explained in the introduction, *agreement* is a liveness property.

3 Bizarre Sequences and Pure Properties

As we pointed out earlier, the *agreement* property of consensus is a liveness property. This is due to the finite sequences in X^* in which two non-crashed processes disagree: an extension of such a sequence satisfies *agreement* only if at least one of the two processes crashes. Motivated by this observation, for any property P, we define the notions of *undesirable finite sequences* and *bizarre infinite sequences with respect to* P. A finite sequence β in X^* is said to be

undesirable with respect to P if all extensions of β satisfying P contain a crash action after β:

$$\forall \sigma \in X \ : \ \beta \prec \sigma \text{ and } \sigma \in P \Rightarrow Crash(\sigma \setminus \beta) \neq \emptyset.$$

We say that an infinite sequence σ in X is *bizarre with respect to* P if it has a prefix that is undesirable with respect to P:

$$\exists \beta \prec \sigma, \ \forall \sigma' \in X \ : \ \beta \prec \sigma' \text{ and } \sigma' \in P \Rightarrow Crash(\sigma' \setminus \beta) \neq \emptyset.$$

The set of sequences in X that are bizarre with respect to P is denoted by $\mathcal{B}(P)$. We then define the property transformer $Pure$ which removes the bizarre sequences from the traces of P. More formally,

$$Pure(P) = P \setminus \mathcal{B}(P).$$

Clearly, we have $Pure(P) \subseteq P$.

If P is a fixed point of $Pure$, i.e., $Pure(P) = P$, then P is said to be *pure*.

To illustrate this definition, we now give the pure versions of some properties. From $Pure(P) \subseteq P$, it follows that $Pure(\emptyset) = \emptyset$. Because every $\beta \in X^*$ has an extension in X with no crash action after β, X is also pure, i.e., $Pure(X) = X$. More interesting is the example of the *agreement* property of consensus: it is easy to show that $Pure(agreement)$ consists of all the sequences in X for which at any point no two alive[2] processes have decided differently. Formally, $Pure(agreement)$ is the set of sequences σ such that for any processes $p, q \in \Pi$,

$$\left.\begin{array}{l} \exists i \ : \ \sigma[i] = decide_p(v) \\ \exists j \ : \ \sigma[j] = decide_q(v') \\ \quad i < j \text{ and } v \neq v' \end{array}\right\} \Rightarrow \exists k \ : \ i < k < j \text{ and } \sigma[k] = crash_p.$$

In order to prevent any disagreement, some works [NT90, Lyn96, CBS00] introduced a strengthening of the agreement property, called the *uniform agreement* property. More precisely, the *uniform agreement* property specifies that no two processes (whether correct or not) decide differently. The pure version of agreement only precludes disagreement among processes that are alive. The $Pure(agreement)$ property is thus weaker than *uniform agreement* but stronger than *agreement*. To the best of our knowledge, this is the first time that this refinement of the *agreement* property of consensus is proposed in the literature.

In an extended version of the paper, we study the pure versions of other properties (as for example, the validity property of atomic commitment). Interestingly, the property transformer $Pure$ leads to new problem specifications that are worthy studying from an algorithmic point of view.

4 Properties of the *Pure* Transformer

In the following three propositions, we state some basic properties of the property transformer $Pure$. We start with a straightforward observation:

[2] A process p is *alive at time* k *in* $\sigma \in X$ if for all $k' \leq k$, $\sigma[k'] \neq crash_p$.

Proposition 1. *Let P and Q be any two properties in $\mathcal{P}$. If $P \subseteq Q$ then $\mathcal{B}(Q) \subseteq \mathcal{B}(P)$ and $Pure(P) \subseteq Pure(Q)$.*

As an easy consequence of Proposition 1, we have the following properties of the *Pure* transformer:

Proposition 2. *Let P and Q be any two properties in $\mathcal{P}$.*

1. $Pure(P \cap Q) \subseteq Pure(P) \cap Pure(Q)$.
2. $Pure(P) \cup Pure(Q) \subseteq Pure(P \cup Q)$.

Note that part 2 of Proposition 2 implies the union (or equivalently, the disjunction) of two pure properties is pure.

Any property can be purified at most once. This is formally expressed by the following proposition:

Proposition 3. *The property transformer Pure is idempotent, i.e.,*

$$\forall P \in \mathcal{P} \ : \ Pure(Pure(P)) = Pure(P).$$

In other words, the pure properties (i.e., the fixed points of *Pure*) are exactly the properties in the image $Pure(\mathcal{P})$.

Proof. We show that $\mathcal{B}(Pure(P)) = \mathcal{B}(P)$. From this intermediate result, it will follow that

$$\begin{aligned} Pure(Pure(P)) &= Pure(P) \setminus \mathcal{B}(Pure(P)) \\ &= P \setminus \mathcal{B}(P) \setminus \mathcal{B}(Pure(P)) \\ &= P \setminus \mathcal{B}(P) \\ &= Pure(P). \end{aligned}$$

First, since $Pure(P) \subseteq P$, Proposition 1 implies that $\mathcal{B}(P) \subseteq \mathcal{B}(Pure(P))$. Second, we show that $\mathcal{B}(Pure(P)) \subseteq \mathcal{B}(P)$. Let σ be in $\mathcal{B}(Pure(P))$ and let β be the *shortest* prefix of σ such that

$$\forall \sigma' \succ \beta \ : \ \sigma' \in Pure(P) \Rightarrow crash(\sigma' \setminus \beta) \neq \emptyset.$$

Let $\hat{\sigma}$ be any sequence in P such that $\beta \prec \hat{\sigma}$. To complete the proof, we need to show that a crash action occurs in $\hat{\sigma} \setminus \beta$. For that, we consider two cases:

1. $\hat{\sigma} \in Pure(P)$. Since $Pure(P) \subseteq P$, it follows that $Crash(\hat{\sigma} \setminus \beta) \neq \emptyset$ as needed.
2. $\hat{\sigma} \notin Pure(P)$. Since $\hat{\sigma} \in P$, we have $\hat{\sigma} \in \mathcal{B}(P)$. Hence, there is a prefix γ of $\hat{\sigma}$ such that:
$$\forall \sigma'' \succ \gamma \ : \ \sigma'' \in P \Rightarrow Crash(\sigma'' \setminus \gamma) \neq \emptyset.$$
We consider two sub-cases:
   (a) $\beta \preceq \gamma$. Since $\hat{\sigma} \in P$, a crash action occurs in $(\hat{\sigma} \setminus \gamma)$, and *a fortiori* in $(\hat{\sigma} \setminus \beta)$.

(b) $\gamma \prec \beta$. Let σ'' be any sequence in $Pure(P)$ such that $\gamma \prec \sigma''$. Since $Pure(P) \subseteq P$, $\sigma'' \in P$ and a crash action thus occurs in $\sigma'' \setminus \gamma$ – a contradiction with the definition of β. Therefore, this sub-case cannot occur.

As a consequence of Proposition 1 and Proposition 3, we get that every property in $\mathcal{P}$ can be given as the disjoint union of a pure property and a *completely non pure* property, i.e., a property such that its image under $Pure$ is the empty set.

Proposition 4. *For every property P in $\mathcal{P}$, there exist two properties Q and R such that:*

1. $P = Q \cup R$
2. $Pure(Q) = Q$ *and* $Pure(R) = \emptyset$.

Proof. By definition of $Pure(P)$, we have:

$$P = Pure(P) \cup (P \cap \mathcal{B}(P)).$$

Note that $Pure(P)$ and $P \cap \mathcal{B}(P)$ are disjoint.

Proposition 3 implies that $Pure(P)$ is pure. We claim that $Pure(P \cap \mathcal{B}(P)) = \emptyset$. By definition of the $Pure$ transformer, we have that $Pure(P \cap \mathcal{B}(P)) = (P \cap \mathcal{B}(P)) \setminus \mathcal{B}(P \cap \mathcal{B}(P))$. From Proposition 1, it follows that $\mathcal{B}(P) \subseteq \mathcal{B}(P \cap \mathcal{B}(P))$. We thus have $Pure(P \cap \mathcal{B}(P)) \subseteq (P \cap \mathcal{B}(P)) \setminus \mathcal{B}(P)) = \emptyset$, i.e., $Pure(P \cap \mathcal{B}(P)) = \emptyset$ as needed.

5 The *Pure* Transformer and the Closure Operator

For some of the developments below, it is useful to study the joint actions of the $Pure$ transformer and the topological closure operator. First, we show that any sequence that is not bizarre with respect to some property P, is necessarily a trace of $\overline{P}$.

Proposition 5. *For any property $P \in \mathcal{P}$, $X \setminus \mathcal{B}(P) \subseteq \overline{P}$.*

Proof. Let σ be any sequence in $X \setminus \mathcal{B}(P)$, i.e., for any prefix β of σ, there exists $\sigma' \in P$ that extends β and such that $Crash(\sigma' \setminus \beta) = \emptyset$. In particular, any prefix β of σ can be extended into a trace of P. This proves that σ is a trace of $\overline{P}$.

The following proposition describes the set of traces of $\overline{Pure(P)}$ in terms of the set of sequences that are bizarre with respect to P. This is a key technical result when studying the combinations of the $Pure$ transformer and the topological closure operator.

Proposition 6. *For any property $P \in \mathcal{P}$, $\overline{Pure(P)} = X \setminus \mathcal{B}(P)$.*

Proof. First, we show that if σ is a trace of $\overline{Pure(P)}$ then σ is not bizarre with respect to P. Let β be any prefix of σ; there exists a trace of $Pure(P)$, σ', such that $\beta \prec \sigma'$. In turn, this implies β has some extension $\hat{\sigma}$ in P such that no crash action occurs in $\hat{\sigma} \setminus \beta$. This shows that σ is not bizarre with respect to P, as needed.

We then show the converse, namely: if σ is not bizarre with respect to P then σ is a trace of $\overline{Pure(P)}$. Let β be any prefix of σ. Since $\sigma \notin \mathcal{B}(P)$, there exists an extension σ' of β in P such that no crash action occurs in $\sigma' \setminus \beta$. We now prove that σ' is a trace of $Pure(P)$. We proceed by contradiction and we assume that there exists a prefix γ of σ' such that for any $\sigma'' \in P$ extending γ, a crash action occurs in $\sigma'' \setminus \gamma$. We consider two cases:

1. $\beta \preceq \gamma$. Then $\sigma' \setminus \gamma$ is a subsequence of $\sigma' \setminus \beta$ – a contradiction to the fact that $Crash(\sigma' \setminus \beta) = \emptyset$ and $\sigma' \setminus \gamma \neq \emptyset$.
2. $\gamma \prec \beta$. Then γ is a prefix of σ. Since σ is not bizarre with respect to P, γ has an extension $\hat{\sigma}$ in P such that no crash action occurs in $\hat{\sigma} \setminus \gamma$. This contradicts the definition of γ.

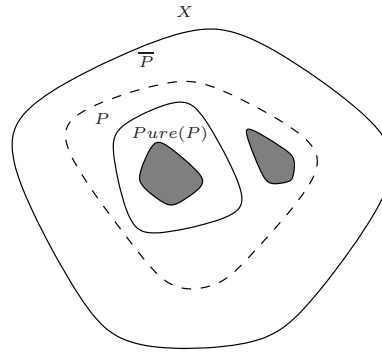


Fig. 1. Grey parts represent $\overline{Pure(P)} = X \setminus \mathcal{B}(P)$.

From the two previous propositions, we get the following useful relation between $\overline{Pure(P)}$ and $Pure(\overline{P})$.

Corollary 7. *For any property* $P \in \mathcal{P}$, $\overline{Pure(P)} \subseteq Pure(\overline{P})$.

Proof. By Proposition 6, $\overline{Pure(P)} = X \setminus \mathcal{B}(P)$. Since $P \subseteq \overline{P}$, Proposition 1 implies that $X \setminus \mathcal{B}(P) \subseteq X \setminus \mathcal{B}(\overline{P})$. Moreover, by Proposition 5, $X \setminus \mathcal{B}(P) \subseteq \overline{P}$. It follows that

$$X \setminus \mathcal{B}(P) \subseteq (X \setminus \mathcal{B}(\overline{P})) \cap \overline{P} = Pure(\overline{P}).$$

Therefore, we have $\overline{Pure(P)} \subseteq Pure(\overline{P})$, as needed.

Inclusion in Corollary 7 is strict in general. To see that, consider the *agreement* property of consensus. As mentioned already, *agreement* is a liveness property, i.e., $\overline{agreement} = X$ and X is pure. Thus, we get $Pure(\overline{agreement}) = X$. On the other hand, by the description of $Pure(agreement)$ given above, any sequence that is not a trace of $Pure(agreement)$ has a prefix that cannot be extended in a trace of $Pure(agreement)$. In other words, $Pure(agreement)$ is a safety property, i.e., $\overline{Pure(agreement)} = Pure(agreement)$. This proves that $\overline{Pure(agreement)} \neq Pure(\overline{agreement})$. However, it is easy to prove that if P is a pure or a safety property, then we have $\overline{Pure(P)} = Pure(\overline{P})$. More precisely, we have the following two corollaries:

Corollary 8. *For any property $P \in \mathcal{P}$, if P is pure then $\overline{P}$ is also pure.*

Proof. Let P be any pure property in $\mathcal{P}$. From Corollary 7, it follows that $\overline{P} \subseteq Pure(\overline{P})$ since P is pure. By definition of the $Pure$ transformer, $Pure(\overline{P}) \subseteq \overline{P}$, and so $Pure(\overline{P}) = \overline{P}$ as needed.

Corollary 9. *For any property $P \in \mathcal{P}$, we have:*

1. *If P is a safety property, then $Pure(P)$ is a safety property.*
2. *If $Pure(P)$ is a liveness property, then P is pure and P is a liveness property.*

Proof. For Part 1, let P be a safety property in $\mathcal{P}$, i.e., $\overline{P} = P$. From Corollary 7, it follows that $\overline{Pure(P)} \subseteq Pure(P)$. By definition of property closure, $Pure(P) \subseteq \overline{Pure(P)}$. Therefore, $\overline{Pure(P)} = Pure(P)$ as needed.

For Part 2, let $P \in \mathcal{P}$ such that $\overline{Pure(P)} = X$. Proposition 5 implies that $\mathcal{B}(P) = \emptyset$. It follows that $Pure(P) = P$.

As explained above, the *agreement* property of consensus is a liveness property whereas $Pure(agreement)$ is a safety property. This example shows that the converse of part 1 and part 2 in Corollary 9 do not hold. In other words, it may be the case that some property P is such that $Pure(P)$ is a safety property but not P. Similarly, there exist some liveness properties whose pure version is not a liveness property.

6 Most Safety Properties Are Pure

In this section, we study whether there are safety properties that are not pure. In general, the answer is yes. Indeed, let σ be a trace of X in which (at least) one crash occurs, and consider the property $P = \{\sigma\}$. Clearly, P is a safety property, i.e., $\overline{P} = P$. Let β be any prefix of σ that contains no crash action. There is exactly one extension of β that satisfies P, namely σ. By definition of β, $\sigma \setminus \beta$ contains a crash action. Therefore, $Pure(P)$ is empty, and so $Pure(P) \neq P$. In fact, as we will see below, P is a typical example of the safety properties that are not pure: a safety property that specifies the time when some process must crash is not pure.

We first formalize the notion of properties that are independent of the times at which crash failures occur. To do so, we consider the natural binary equivalence relation on X defined as follows:

$$\sigma \sim \sigma' \Leftrightarrow Crash(\sigma) = Crash(\sigma') \text{ and } \sigma \setminus Crash(\sigma) = \sigma' \setminus Crash(\sigma')$$

Informally, this means that σ and σ' have the same subsequences of non-crash actions and the same sets of crash actions. For any property $P \in \mathcal{P}$, we define property $K(P)$ by

$$K(P) = \{\sigma \in P \mid \forall \sigma' \in X,\ \sigma' \sim \sigma \Rightarrow \sigma' \in P\}.$$

If $K(P) = P$, then, intuitively, P is independent of the times when crash failures occur.

Proposition 10. *If P is a safety property such that $K(P) = P$, then P is pure.*

Proof. Suppose, by contradiction, that there exists $\sigma \in P \setminus Pure(P)$. Therefore σ has a prefix β such that

$$\forall \sigma' \succ \beta \ :\ \sigma' \in P \Rightarrow Crash(\sigma' \setminus \beta) \neq \emptyset.$$

In particular some crash actions occur in $\sigma \setminus \beta$. Let $(\sigma^k)_{k \in \mathbf{N}}$ be the sequence of traces such that for any index k

1. $Crash(\sigma^k) = Crash(\sigma)$;
2. $\sigma^k \setminus Crash(\sigma^k) = \sigma \setminus Crash(\sigma)$;
3. $\forall a \in Crash(\sigma)$, a does k moves to the right.

Clearly, the $(\sigma^k)_{k \in \mathbf{N}}$ sequence converges, and its limit $\tilde{\sigma}$ is an extension of β. Moreover, no crash action occurs in $\tilde{\sigma} \setminus \beta$.

From $K(P) = P$, it follows that every σ^k is in P. Since P is a safety property, $\tilde{\sigma}$ is also in P. This yields a contradiction to the fact that in each trace of P which extends β, a crash action occurs after β.

7 Pure Safety and Pure Liveness

In this section, we define two new classes of properties that are more restrictive than the classes of safety and liveness properties. In order to do so, we substitute a *pure closure* operator for the classical closure operator: a *pure safety* property will coincide with its pure closure; a *pure liveness* property will be a property whose pure closure is equal to X. We give several characterizations of pure safety properties and of pure liveness properties. We then prove a counterpart of the decomposition theorem of Alpern and Schneider [AS85], stating that every pure property is the conjunction of a pure safety property and a pure liveness property.

For any property $P \in \mathcal{P}$, we consider the property $\overline{Pure(P)}$ that we call the *pure closure* of P and denote $\tilde{P}$ (since, as shown below, the transformer $P \to \tilde{P}$

is non decreasing and idempotent, we are indeed allowed to call it a "closure"). By Proposition 6, the traces of $\tilde{P}$ are the sequences that are not bizarre with respect to P, i.e., $\tilde{P} = X \setminus \mathcal{B}(P)$. First, we prove a basic result on the pure closure operator.

Proposition 11. *For any property $P \in \mathcal{P}$, $\tilde{P}$ is both a safety property and a pure property.*

Proof. By definition, $\tilde{P}$ is a safety property. Then we claim that $\tilde{P}$ is pure. By applying Corollary 7 to $Pure(P)$, we obtain

$$\overline{Pure(Pure(P))} \subseteq Pure(\overline{Pure(P)}).$$

By Proposition 3, this becomes

$$\overline{Pure(P)} \subseteq Pure(\overline{Pure(P)}),$$

i.e.,

$$\tilde{P} \subseteq Pure(\tilde{P}).$$

Since $Pure(\tilde{P}) \subseteq \tilde{P}$, we deduce that $Pure(\tilde{P}) = \tilde{P}$, i.e., $\tilde{P}$ is pure.

As a consequence, we can easily prove that the pure closure operator is idempotent.

Proposition 12. *The pure closure operator is idempotent, i.e.,*

$$\forall P \in \mathcal{P} \ : \ \widetilde{\widetilde{P}} = \tilde{P}.$$

Proof. By definition of the pure closure operator, we have $\widetilde{\widetilde{P}} = \overline{Pure(\tilde{P})}$. Since $\tilde{P}$ is a pure and a safety property, we get that $\widetilde{\widetilde{P}} = \overline{\tilde{P}}$, and then $\widetilde{\widetilde{P}} = \tilde{P}$.

In addition, the pure closure operator is non decreasing.

Proposition 13. *Let P and Q be any two properties in $\mathcal{P}$.*

1. *If $P \subseteq Q$, then $\tilde{P} \subseteq \tilde{Q}$.*
2. *$\widetilde{P \cap Q} \subseteq \tilde{P} \cap \tilde{Q}$ and $\tilde{P} \cup \tilde{Q} \subseteq \widetilde{P \cup Q}$.*

Proof. Part 1 easily follows from Proposition 1 and the fact that the topological closure operator is non decreasing.

Part 2 is a straightforward consequence of Part 1.

The following proposition characterizes properties that are invariant under the pure closure operator.

Proposition 14. *For any property $P \in \mathcal{P}$, the following three assertions are equivalent:*

1. $\tilde{P} = P$.
2. *P is a safety property and is pure.*
3. *σ is a trace of P iff σ has no undesirable prefix with respect to P, i.e.,*

$$\sigma \in P \Leftrightarrow \forall \beta \prec \sigma,\ \exists \sigma' \succ \beta\ :\ \sigma' \in P \text{ and } Crash(\sigma' \setminus \beta) = \emptyset.$$

is called the *pure closure* of P and denoted $\tilde{P}$.

If P satisfies one of these equivalent assertions, P is said to be a *pure safety* property. Because of the last characterization, a pure safety property can be interpreted as saying that some particular "undesirable" thing never happens.

Proof. The implication (1) $\Rightarrow$ (2) is a straightforward consequence of Proposition 11. Conversely, if P is both a safety property and a pure property, then $Pure(P) = P$ and $\overline{P} = P$, and so $\tilde{P} = P$. This proves (2) $\Rightarrow$ (1).

We now show that (1) $\Leftrightarrow$ (3). By Proposition 6, (1) is equivalent to $P = X \setminus \mathcal{B}(P)$, and hence to the fact that a sequence σ is a trace of P iff σ is not bizarre with respect to P. Since a sequence is bizarre iff it has an undesirable prefix, this yields the required equivalence.

For example, the property whose set of traces is empty is trivially a pure safety property. This is also the case of property X.

At this point, it is interesting to note that Proposition 10 says that a large class of safety properties (namely the safety properties which are fixed points of the property transformer K introduced in Section 6) are indeed pure safety properties.

We now give four equivalent characterizations of the properties whose pure closure is equal to X.

Proposition 15. *For any property $P \in \mathcal{P}$, the following four assertions are equivalent:*

1. $\tilde{P} = X$.
2. *P is a liveness property and is pure.*
3. *$Pure(P)$ is a liveness property.*
4. $\forall \beta \in X^*,\ \exists \sigma \succ \beta\ :\ \sigma \in P$ *and* $Crash(\sigma \setminus \beta) = \emptyset$.

If P satisfies one of these equivalent assertions, P is said to be a *pure liveness* property. According to the fourth characterization, a pure liveness property can be understood as saying that there is no undesirable sequence, i.e., some particular "good" thing eventually happens without the help of crashes.

Proof. The implication (1) $\Rightarrow$ (2) follows immediately from the second point of Corollary 9. Moreover, implications (2) $\Rightarrow$ (3) and (3) $\Rightarrow$ (1) are obvious. This yields the equivalence of (1), (2) and (3).

We now show that (1) $\Rightarrow$ (4). Suppose that $\tilde{P} = X$, and let β be an arbitrary (finite) sequence of X^*. Since $\overline{Pure(P)} = X$, there exits an extension σ of β that belongs to $Pure(P)$. Because $Pure(P) \subseteq P$, σ is a trace of P, and so (4) holds.

Conversely, we show that (4) $\Rightarrow$ (1). Assume that (4) holds. Let β be an arbitrary sequence of X^*. We must prove that β can be extended in a trace of $Pure(P)$. By assertion (4), β has an extension σ in P. We claim that σ is indeed in $Pure(P)$. To see this, consider any prefix γ of σ. By assertion (4), there exists an extension of γ in P in which no crash action occurs after γ. We then have $\sigma \in Pure(P)$ as needed.

We now give the counterpart of the decomposition theorem of Alpern and Schneider:

Proposition 16. *If P is a pure property, then there exist a pure safety property S and a pure liveness property L such that*

$$P = S \cap L.$$

Proof. For any property P, we trivially have:

$$P = \overline{P} \cap (X \setminus (\overline{P} \setminus P)) = \overline{P} \cap (P \cup (X \setminus \overline{P})).$$

According to [AS85] $\overline{P}$ and $P \cup (X \setminus \overline{P})$ are safety and liveness properties, respectively.

Corollary 8 states that if P is a pure property, then $\overline{P}$ is also a pure property. We now show that if P is a pure property then $P \cup (X \setminus \overline{P})$ is also a pure property. We claim that $\mathcal{B}(P \cup (X \setminus \overline{P})) = \emptyset$. Suppose for the sake of contradiction that σ is a bizarre sequence with respect to $P \cup (X \setminus \overline{P})$, i.e., $\sigma \in \mathcal{B}(P \cup (X \setminus \overline{P}))$. So σ has a finite prefix β such that

$$\forall \sigma' \succ \beta \ : \ \sigma' \in P \text{ or } \sigma' \notin \overline{P} \Rightarrow Crash(\sigma' \setminus \beta) \neq \emptyset.$$

But $Pure(X) = X$, and so there exists an extension $\tilde{\sigma}$ of β without any crash after β. By definition of β, $\tilde{\sigma} \in \overline{P} \setminus P$. In particular, β has an extension $\hat{\sigma} \in P$. On the other hand, since P is pure, some extension $\check{\sigma}$ of β in P is such that there is no crash action in $\check{\sigma}$ after β. This is a contradiction.

References

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. 554, 561, 564

[AW98] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Mc Graw Hill, 1998. 552

[CBS00] B. Charron-Bost and André Schiper. Uniform consensus is harder than consensus. Technical report, École Polytechnique Fédérale de Lausanne, April 2000. 556

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996. An extended abstract appeared in *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, August, 1992, 147–158. 552

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987. 552

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988. 552

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. 552

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: proceedings of the IFIP 9th World Congress*, pages 657–668. IFIP, North-Holland, September 1983. 552

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996. 556

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990. 556

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977. 552

Well-Abstracted Transition Systems (Extended Abstract)

Alain Finkel[1⋆], Purushothaman Iyer[2⋆⋆], and Gregoire Sutre[1]

[1] LSV, ENS Cachan & CNRS UMR 8643
61, Av. du Pdt. Wilson, Cachan, France
[2] Dept of Computer Science, NC State University
Raleigh, NC 27615, USA

Abstract. Formal methods based on symbolic representations have been found to be very effective. In the case of infinite state systems, there has been a great deal of interest in *accelerations* – a technique for characterizing the result of iterating an execution sequence an arbitrary number of times, in a sound, but not necessarily complete, way. We propose the use of abstractions as a general framework to design accelerations. We investigate SemiLinear Regular Expressions (SLREs) as symbolic representations for FIFO automata. In particular, we show that SLREs are *easy to manipulate* (inclusion between two SLREs is in NP ∩ coNP), they form the *core* of known FIFO symbolic representations ($SLREs = QDDs \cap CQDDs$), and they are usually *sufficient* since for FIFO automata with one channel, an arbitrary iteration of a loop is LRE representable.

1 Introduction

Formal methods are now routinely applied in design and implementation of finite state systems, such as those that occur in VLSI circuits. It has also been applied fairly regularly in the design and implementation of network protocols. Based on the success of formal methods in reasoning about finite state systems [BCM+92] there has been a great deal of interest in reasoning about infinite state systems. Given that programs, as well as network protocols, are infinite state in nature there is a need for automatic techniques to extend the reach of formal methods to a much larger class.

Infinite state systems could, in general, be Türing-powerful. Consequently, in reasoning about any non-trivial property of such systems we will have to contend with incompleteness. At least two approaches have been considered in the literature: (a) semi-computation of the set of reachable states [BGWW97, BH97, AAB99], and (b) computation of a superset of reachability set [CC77, PP91]. A requirement common to both approaches is that an infinite set of reachable states (from some given initial state) be finitely described. Clearly,

⋆ Supported in part by FORMA, a project funded by DGA, CNRS and MENSER.
⋆⋆ Supported in part by ARO under grant DAAG55-98-1-03093.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 566–581, 2000.

the finite description should be such that it admits questions of membership and emptiness to be answered effectively. However, given that the reachability set is explored in an iterative fashion, an even more important question is "how does one infer the existence of an infinite set of states in the reachability set? And how does one calculate it?" Techniques called *accelerations* or *meta*-transitions have been discussed in the literature [Céc98, BGWW97, BH97, AAB99, CJ98, FS00]. We focus in this paper on symbolic representations for the computation of the reachability set of FIFO automata — a finite control with multiple unbounded FIFO channels. To the best of our knowledge, Pachl uses for the first time regular expressions to represent infinite sets of channel contents [Pac87]. In [FM96], linear regular expressions have been defined and used. Boigelot et al. chosed a deterministic finite automata based representation [BGWW97] and afterwards Bouajjani et al. added Pressburger formulae [BH97]. Simple regular expressions have been introduced for lossy FIFO automata [AAB99].

We propose to address the issue "what are symbolic representations?" In this paper, we show how symbolic representations and accelerations can be couched in terms of abstract interpretation [CC77], a powerful semantics-based technique for explaining data flow analysis. We illustrate the usefulness of this approach by exploring Linear and SemiLinear Regular Expressions (LREs and SLREs) as symbolic representations for FIFO automata. In particular, we show the following about SLREs:

- SLREs are *easy to manipulate*: indeed, SLREs are exactly regular languages of *polynomial density*. This class enjoys good complexity properties. In particular, we prove that inclusion between two SLREs is in NP $\cap$ coNP.
- they form the *core* of known FIFO symbolic representations: more formally, a set of queue contents is SLRE representable iff it is both CQDD representable and QDD representable ($SLREs = QDDs \cap CQDDs$),
- and they are usually *sufficient* since for FIFO automata with one channel, an arbitrary iteration of a loop is LRE representable. This result was used, but not proved, in [FM96]. The proof is technical and it can be found in the full paper [FIS00]. Several examples in the literature have a SLRE representable reachability set: the Alternating Bit Protocol [Pac87], the bounded retransmission protocol of Philips [AAB99], the producer/consumer described in [BGWW97] and the connection/deconnection protocol [JJ93].

The road map for the paper is as follows: in Section 2 we introduce the notion of abstractions, in Section 3 we discuss FIFO automaton and symbolic representations based on SLREs. In Section 4 we introduce our notion of symbolic representations as abstractions. In Section 5 we introduce the notion of acceleration in our setting, and also discuss accelerations based on LREs. Finally, in Section 6 we compare QDDs, CQDDs and SLREs.

2 Labelled Transition Systems

We write 2^S (resp. $\mathcal{P}_f(S)$) to denote the set of subsets (resp. finite subsets) of a set S. Let S^I be the set of I-indexed (I finite) vectors of elements of S. Given

$\mathsf{w} \in S^I$, $i \in I$ and $s \in S$, define $\mathsf{w}[i := s]$ to be the vector w' that differs from w at the index i ($\mathsf{w}'[i] = s$), but is the same elsewhere. Any element $s \in S$ induces the vector $\mathsf{s} \in S^I$ defined by $\mathsf{s}[i] = s$ for all $i \in I$. An I-indexed vector $\mathsf{w} \in S^I$ may also be denoted anonymously by $< i \mapsto \mathsf{w}[i] >$.

A *quasi-ordered set* is a pair $(S, \preceq)$ where S is a set and $\preceq$ is a reflexive and transitive relation over S ; we denote by $\prec$ the relation over S defined by $a \prec b$ if $a \preceq b$ and $b \not\preceq a$. A quasi-ordered set $(S, \preceq)$ satisfies the *ascending chain condition* if there does not exist an infinite strictly increasing sequence $s_0 \prec s_1 \prec s_2 \cdots s_k \prec s_{k+1} \cdots$ in S.

Let Σ be an alphabet (a finite, non empty set). We write Σ^* (resp. Σ^ω) for the set of all *finite words* (resp. *infinite words*), and ε denotes the empty word. We denote by Σ^+ the set $\Sigma^* \setminus \{\varepsilon\}$. For two words $x, y \in \Sigma^*$, $x \cdot y$ (shortly written xy) is their *concatenation*. A word y is a *left factor* of a word x, written $y \leq x$, if $x = y \cdot z$ for some word z and moreover we write $z = y^{-1}x$. The pair $(\Sigma^*, \leq)$ is a quasi-ordered set.

Definition 2.1. *A* labelled transition system *LTS is a triple $LTS = (\mathsf{S}, \Sigma, \rightarrow)$ where S is a set of states, Σ is a finite set a labels and $\rightarrow \subseteq \mathsf{S} \times \Sigma \times \mathsf{S}$ is a labelled transition relation.*

For any state $s \in \mathsf{S}$, we write $s \xrightarrow{\varepsilon} s$. If $\sigma = l_1 l_2 \cdots l_k$ is a finite word in Σ^+ and $s, s' \in \mathsf{S}$, we write $s \xrightarrow{\sigma} s'$ when there exists a finite sequence $s_0, s_1, \ldots, s_k \in \mathsf{S}$ such that $s = s_0$, $s' = s_k$ and $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \cdots s_{k-1} \xrightarrow{l_k} s_k$.

Note that the labelled transition relation $\rightarrow$ captures the system behavior as it moves from one state to another. Since the central problem we wish to discuss is based on set of states reachable from a given state s by the $\rightarrow$ relation we associate with every labelled transition system LTS, two total functions $post : 2^{\mathsf{S}} \times \Sigma^* \rightarrow 2^{\mathsf{S}}$ and $post^* : 2^{\mathsf{S}} \rightarrow 2^{\mathsf{S}}$ defined by:

$$post(X, \sigma) = \{s' \in \mathsf{S} \mid \exists\ s \in X, s \xrightarrow{\sigma} s'\}, \text{ and, } post^*(X) = \bigcup_{\sigma \in \Sigma^*} post(X, \sigma)$$

The main aim of our paper is a fast computation of $post^*(s_0)$, for a given $s_0 \in \mathsf{S}$. However, in general, $post^*(s_0)$ could be infinite (and even recursively enumerable). To deal with this problem symbolic representations have been used [BGWW97, BH97, CJ98]. Typically, assumptions are made about what properties these symbolic representations should satisfy. In section 4 we will discuss a minimal set of assumption on these abstractions (*aka* symbolic representations) and how they relate to transition systems.

3 FIFO Symbolic Representations to Compute *post**

Let us present the very well-known Alternating Bit Protocol. It is generally modelled by a system of 2 extended automata which communicate through 2 one directional FIFO channels. It is clear that we may always compute the cartesian product of the 2 extended automata, and this yields a single extended automaton with 2 bi-directional FIFO channels, namely a *FIFO automaton*.

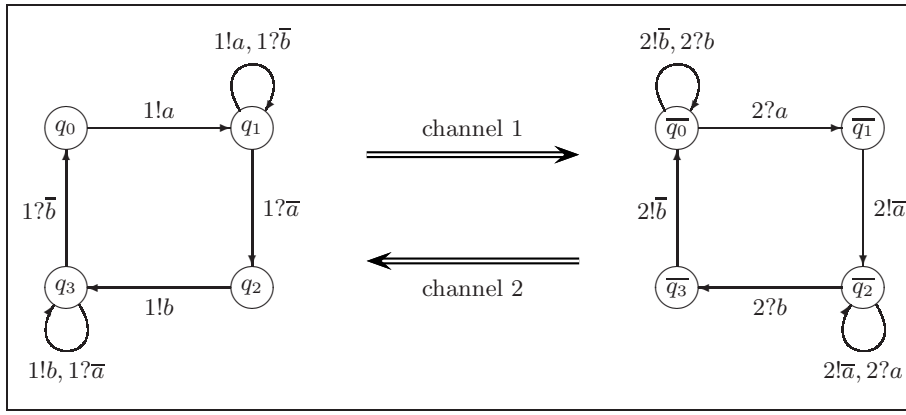


Fig. 1. A FIFO automaton ABP modeling the Alternating Bit Protocol

FIFO automata – a finite control with a set of FIFO channels – is a turing powerful [BZ83] mathematical model for the protocol specification languages Estelle and SDL. Even one channel FIFO automata can simulate Turing Machines and hence we cannot expect to find an algorithm computing exactly $post^*$. However, we will define a general semi-algorithm using symbolic representations, which computes exactly $post^*$ (or an over-approximation).

3.1 FIFO Automata

A FIFO automaton is a 4-tuple $F = (Q, C, M, \delta)$ where Q is a finite set of control states, C is a finite set of channel names, and, M is a finite set of message types, and, $\delta \subseteq Q \times (C \times \{?, !\} \times M) \times Q$ is a finite set of control transitions.

Transitions of the form $(q, c?m, q')$ denote receive actions and transitions of the form $(q, c!m, q')$ denote send actions. A *loop* in a FIFO automaton is any non-empty word $\sigma = (q_0, l_0, q_0')(q_1, l_1, q_1') \cdots (q_k, l_k, q_k') \in \delta^*$ such that $q_k' = q_0$ and for all $0 \leq i \leq k-1$, $q_i' = q_{i+1}$. Moreover, we say that σ is a loop *on* q_0.

Example 3.1. For ABP, we have e.g. that $\sigma_1 = (q_1, \overline{q_0}) \xrightarrow{1!a} (q_1, \overline{q_0})$ and $\sigma_2 = (q_0, \overline{q_0}) \xrightarrow{1!a} (q_1, \overline{q_0}) \xrightarrow{1?\overline{a}} (q_2, \overline{q_0}) \xrightarrow{2!\overline{b}} (q_2, \overline{q_0}) \xrightarrow{1!b} (q_3, \overline{q_0}) \xrightarrow{1?\overline{b}} (q_0, \overline{q_0})$ are loops.

We assume that the channels are perfect and that messages sent by a sender are received by a receiver in the same order they are sent. The operational semantics of a FIFO automaton is given by the following labelled transition system.

Definition 3.2. *A FIFO automaton $F = (Q, C, M, \delta)$ defines the labelled transition system $LTS(F) = (\mathsf{S}, \Sigma, \rightarrow)$ as follows:*

- $\mathsf{S} = Q \times (M^*)^C$*, the set of states containing a control state q and a C-indexed vector of words* w *denoting the channel contents.*

- $\Sigma = \delta$.
- $(q, \mathsf{w}) \xrightarrow{(p,c?m,p')} (q', \mathsf{w}')$ *provided* $q = p$, $q' = p'$ *and* $\mathsf{w} = \mathsf{w}'[c := m\mathsf{w}'[c]]$.
- $(q, \mathsf{w}) \xrightarrow{(p,c!m,p')} (q', \mathsf{w}')$ *provided* $q = p$, $q' = p'$ *and* $\mathsf{w}' = \mathsf{w}[c := \mathsf{w}[c]m]$.

For convenience, we write $(q, \mathsf{w}) \xrightarrow{c?m} (q', \mathsf{w}')$ (resp. $(q, \mathsf{w}) \xrightarrow{c!m} (q', \mathsf{w}')$) when we have $(q, \mathsf{w}) \xrightarrow{(q,c?m,q')} (q', \mathsf{w}')$ (resp. $(q, \mathsf{w}) \xrightarrow{(q,c!m,q')} (q', \mathsf{w}')$).

Example 3.3. For ABP, we have for instance: $(q_0, \overline{q_0}; \varepsilon, \varepsilon) \xrightarrow{1!a} (q_1, \overline{q_0}; a, \varepsilon) \xrightarrow{1!a} (q_1, \overline{q_0}; aa, \varepsilon) \xrightarrow{2?a} (q_1, \overline{q_1}; a, \varepsilon) \xrightarrow{2!\overline{a}} (q_1, \overline{q_2}; a, \overline{a}) \xrightarrow{2!\overline{a}} (q_1, \overline{q_2}; a, \overline{a}\,\overline{a}) \xrightarrow{1?\overline{a}} (q_2, \overline{q_2}; a, \overline{a})$.

3.2 Linear Regular Expressions

Since $post^*$ may be infinitely large for FIFO automata, we need to finitely represent infinite sets of states in order to compute $post^*$. Moreover, the unboundedness of $post^*$ arises from the unbounded channels (the set of control states is finite). Thus, a natural way to proceed is to finitely represent infinite sets of channel contents, i.e. vectors of words.

Example 3.4. For ABP, we have:

$$\begin{aligned} post^*(q_0, \overline{q_0}; \varepsilon, \varepsilon) = \quad & (q_0\ , \overline{q_0}) \times (b^*\ , \overline{b}^*) \quad \cup \quad (q_1\ , \overline{q_0}) \times (b^*a\,a^*\ , \overline{b}^*\ \ ) \\ \cup\ & (q_1\ , \overline{q_1}) \times (a^*\ , \overline{b}^*) \quad \cup \quad (q_1\ , \overline{q_2}) \times (a^*\ \ , \overline{b}^*\overline{a}\,\overline{a}^*\ ) \\ \cup\ & (q_2\ , \overline{q_2}) \times (a^*\ , \overline{a}^*) \quad \cup \quad (q_3\ , \overline{q_2}) \times (a^*b\,b^*\ , \overline{a}^*\ \ ) \\ \cup\ & (q_3\ , \overline{q_3}) \times (b^*\ , \overline{a}^*) \quad \cup \quad (q_3\ , \overline{q_0}) \times (b^*\ \ \ , \overline{a}^*\overline{b}\,\overline{b}^*\ ) \end{aligned}$$

In the rest of the paper, we write M for a finite set of message types and C for a finite set of channel names. Following Pachl, we will use regular expressions to represent infinite sets of FIFO channel contents. We write $REG(M)$ for the set of *regular expressions (REGs)* over M. We denote by $[\![\rho]\!]$ the language associated to a regular expression ρ, and we write $\emptyset$ for the regular expression denoting an empty set of words.

To compactly represent the result of receiving a message from the front of a channel we will use the notion of derivatives [Brz64]. We will thus write, for a regular expression ρ and a message $m \in M$, $m^{-1}(\rho)$ to denote a regular expression with meaning $[\![m^{-1}(\rho)]\!] = \{y \mid my \in [\![\rho]\!]\}$.

Let us define the two following subclasses of regular expressions:

- A *linear regular expression (written LRE)* is any regular expression of the form $x_0{y_0}^*x_1{y_1}^* \cdots x_{n-1}{y_{n-1}}^*x_n$ with $x_i \in M^*$ and $y_j \in M^+$.
- A *semilinear regular expression (written SLRE)* is any finite sum of LREs (possibly the empty sum $\emptyset$).

We write $LRE(M)$ (resp. $SLRE(M)$) for the set of linear (resp. semilinear) regular expressions over M.

Example 3.5. Note that $post^*$ for the Alternating Bit Protocol is composed of 16 LREs. There are other examples in the literature where $post^*$ can be described by SLREs, namely the bounded retransmission protocol of Philips [AAB99], the producer/consumer described in [BGWW97] and the connection/deconnection protocol [JJ93].

We define the *size* of LREs and SLREs as follows:

- the size $|\rho|$ of a LRE $\rho = x_0{y_0}^*x_1{y_1}^* \cdots x_{n-1}{y_{n-1}}^*x_n$ is given by $|\rho| = |x_n| + \Sigma_{i=0}^{n-1}(|x_i| + |y_i| + 1)$, and,
- the size $|\rho|$ of a SLRE $\rho = \rho_0 + \rho_1 \cdots + \rho_m$ is given by $|\rho| = \Sigma_{i=0}^{m}|\rho_i|$.

This definition will allow us to express a complexity result about SLREs in Section 6. The following proposition expresses a strong property of SLREs and it is used to prove Theorem 5.5 and Theorem 6.6. The proof is achieved by induction on the size of a SLRE.

Proposition 3.6. *The class of SLRE languages is closed under intersection with regular languages.*

3.3 FIFO Symbolic Representations

In order to handle different kinds of symbolic representations for FIFO automata (based on QDDs [BGWW97], CQDDs [BH97] or SLREs), we now define formally what we mean by FIFO symbolic representation. We essentially require a FIFO symbolic representation to be closed under *symbolic reception* (written ??) and under *symbolic emission* (written !!), in order to symbolically compute $post^*$.

Definition 3.7. *A* (C, M)-FIFO symbolic representation *(shortly a* FIFO symbolic representation*) is a* 6*-tuple* $\mathcal{F}_{C,M}$ *(shortly* $\mathcal{F}$*) such that* $\mathcal{F}_{C,M} = (\mathsf{F}, \bot, !!, ??, [\![\cdot]\!], +)$ *where* F *is a set of symbolic channel contents,* $\bot \in \mathsf{F}$*,* $?? : C \times M \times \mathsf{F} \to \mathsf{F}$*,* $!! : C \times M \times \mathsf{F} \to \mathsf{F}$*,* $[\![\cdot]\!] : \mathsf{F} \to 2^{(M^*)^C}$ *and* $+ : \mathsf{F} \times \mathsf{F} \to \mathsf{F}$ *are four total functions satisfying for every* $f, f' \in \mathsf{F}$*,* $c \in C$ *and* $m \in M$*:*

$$[\![\bot]\!] = \emptyset \tag{1}$$

$$[\![??(c, m, f)]\!] = \{\mathsf{w}[c := m^{-1}\mathsf{w}[c]] \mid \mathsf{w} \in [\![f]\!], m \leq \mathsf{w}[c]\} \tag{2}$$

$$[\![!!(c, m, f)]\!] = \{\mathsf{w}[c := \mathsf{w}[c]m] \mid \mathsf{w} \in [\![f]\!]\} \tag{3}$$

$$[\![f + f']\!] = [\![f]\!] \cup [\![f']\!] \tag{4}$$

Note that we suppose that FIFO symbolic representations are closed under union and contain the empty set, but these assumptions essentially allow us to simplify notations. We are now ready to show how a FIFO symbolic representation can be defined based on LREs.

Proposition 3.8. *Given* $m \in M$ *and a linear regular expression* ρ*, we have that* $m^{-1}(\rho)$ *is a semilinear regular expression over* M*.*

For any LRE ρ, we also denote by $m^{-1}(\rho)$ the set of LREs $\{\rho_1, \rho_2, \ldots, \rho_n\}$ such that $m^{-1}(\rho) = \rho_1 + \rho_2 + \cdots + \rho_n$.

Definition 3.9. *Let* $\mathcal{LRE} = (\mathcal{P}_f(LRE(M)^C), \emptyset, !!, ??, [\![\cdot]\!], \cup)$, *where* $??$, $!!$ *and* $[\![\cdot]\!]$ *are three total functions defined as follows:*

$$??(c, m, \chi) = \{\mathsf{r}[c := \rho] \mid \mathsf{r} \in \chi, \rho \in m^{-1}(\mathsf{r}[c])\}$$
$$!!(c, m, \chi) = \{\mathsf{r}[c := \mathsf{r}[c] \cdot m] \mid \mathsf{r} \in \chi\}$$
$$[\![\chi]\!] = \bigcup_{\mathsf{r} \in \chi} \prod_{c \in C} [\![\mathsf{r}[c]]\!]$$

Proposition 3.10. *$\mathcal{LRE}$ is a FIFO symbolic representation.*

Now, starting from a finite set of states, how to reach a "symbolic state" denoting an infinite set of states ? A natural way is to compute the effect of a loop, for instance iterating the loop σ_1 of ABP gives us: $(q_1, \overline{q_0}; \varepsilon, \varepsilon) \xrightarrow{(!a)^*} (q_1, \overline{q_0}; a^*, \varepsilon)$. We present in the next section our general framework of abstraction and acceleration formalizing these intuitive strategy. This general setting is given for labelled transition system, and is then applied to FIFO automata.

4 Abstraction and Acceleration of Labelled Transition Systems

We now introduce the notion of abstraction of labelled transition systems, a fairly general setting for our discussions. "What an abstraction is" is based on a minimal set of assumptions which allows us to present and compare several symbolic representations in an uniform setting. Formally,

Definition 4.1. *An* abstraction $\mathcal{A}_{LTS}$ of a labelled transition system $LTS = (\mathsf{S}, \Sigma, \rightarrow)$ *is a* 5*-tuple* $\mathcal{A}_{LTS} = (\mathsf{A}, post_{\mathcal{A}}, \gamma, \sqcup, \Sigma)$ *where* A *is a set of (abstract) states,* $post_{\mathcal{A}} : \mathsf{A} \times \Sigma \rightarrow \mathsf{A}$, $\gamma : \mathsf{A} \rightarrow 2^{\mathsf{S}}$ *and* $\sqcup : \mathsf{A} \times \mathsf{A} \rightarrow \mathsf{A}$ *are three total functions satisfying for every* $a, b \in \mathsf{A}$ *and* $l \in \Sigma$*:*

$$\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b), \text{ and,} \tag{5}$$
$$post(\gamma(a), l) \subseteq \gamma(post_{\mathcal{A}}(a, l)) \subseteq post^*(\gamma(a)) \tag{6}$$

An abstract state a denotes a potentially infinite set $\gamma(a)$ of (concrete) states. Condition 5 enforces the set of abstract states A to be closed under (abstract) union $\sqcup$. Condition 6 ensures that an abstract exact computation of $post^*$ can be performed. Note that a labelled transition system may have several abstractions. In the rest of the paper, we write $\mathcal{A}_{LTS}$ to denote that $\mathcal{A}_{LTS}$ is an abstraction of a labelled transition system LTS.

An abstraction carries with it a natural quasi-ordering $\sqsubseteq$ induced by the function γ and defined by $a \sqsubseteq b$ if $\gamma(a) \subseteq \gamma(b)$. Furthermore, this quasi-ordering induces a natural equivalence relation $\cong$ defined by $a \cong b$ if $a \sqsubseteq b$ and $b \sqsubseteq a$. Note that two equivalent abstract states denote the same set of states.

The importance of the $\cong$ relation is that the $\sqcup$ operation is commutative and associative with respect to the equivalence. Hence for every finite subset X of

A, $\sqcup X$ is well defined with respect to $\cong$. This allows us to compute $post^*$ in an iterative manner, where we take the unions (i.e., $\sqcup$) as we go along, and thus have to maintain only one element of the symbolic representation at any time in computing the reachability set. Note that on the contrary to the usual abstract interpretation framework [CC77], we are not assuming that the abstraction domain A is closed under arbitrary union (i.e., lubs) but only finite ones.

An abstraction also implicitly defines a deterministic LTS $(\mathsf{A}, \Sigma, post_{\mathcal{A}})$. A simple consequence of our definitions is that the notion of abstraction immediately gives us a simulation:

Proposition 4.2. *Let $\mathcal{A}_{LTS} = (\mathsf{A}, post_{\mathcal{A}}, \gamma, \sqcup, \Sigma)$ be an abstraction of a labelled transition system $LTS = (\mathsf{S}, \Sigma, \rightarrow)$. For every $a_0 \in \mathsf{A}$, the labelled transition system $(\mathsf{A}, \Sigma, post_{\mathcal{A}})$ with initial state a_0 simulates LTS with the set of initial states $\gamma(a_0)$.*

Abstraction of FIFO Automata A FIFO symbolic representation $\mathcal{F} = (\mathsf{F}, \bot, !!, ??, [\![\cdot]\!], +)$ allows us to associate a canonical abstraction to any FIFO automaton F, as follows.

Definition 4.3. *Let $\mathcal{A}^{\mathcal{F}}_{LTS(F)} = (\mathsf{A}, post_{\mathcal{A}}, \gamma, \sqcup, \Sigma)$, where $post_{\mathcal{A}} : \mathsf{A} \times \Sigma \rightarrow \mathsf{A}$, $\gamma : \mathsf{A} \rightarrow 2^{\mathsf{S}}$ and $\sqcup : \mathsf{A} \times \mathsf{A} \rightarrow \mathsf{A}$, be defined by:*

$$\begin{aligned}
\mathsf{A} &= \mathsf{F}^Q \\
post_{\mathcal{A}}(\mathsf{a}, (q, c?m, q')) &= \bot[q' := !!(c, m, \mathsf{a}[q])] \\
post_{\mathcal{A}}(\mathsf{a}, (q, c!m, q')) &= \bot[q' := ??(c, m, \mathsf{a}[q])] \\
\gamma(\mathsf{a}) &= \bigcup_{q \in Q} \{q\} \times [\![\mathsf{a}[q]]\!] \\
\sqcup(\mathsf{a}, \mathsf{b}) &= < q \mapsto \mathsf{a}[q] + \mathsf{b}[q] >
\end{aligned}$$

It is readily seen that for any FIFO symbolic representation $\mathcal{F}$ and for any FIFO automaton F, $\mathcal{A}^{\mathcal{F}}_{LTS(F)}$ is an abstraction of $LTS(F)$. Thus, we obtain that $\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}$ is an abstraction.

We will now discuss how several strategies alternately called acceleration or meta-transitions can be captured uniformly by our definitions. Without acceleration, the abstract state obtained at each step of the computation of $post^*(s_0)$ represents only a finite portion of $post^*(s_0)$. Consequently, the advantage of using a symbolic representation might never be realized.

The way out of this dilemma is based on the observation that if a sequence of transitions σ can be iterated infinitely often from a state s, and if $\bigcup_{i \in \mathbb{N}} post(s, \sigma^i)$ can be calculated in a symbolic manner, then a portion of the reachability set containing an *infinite* set of states can be captured in a *single* step. The motivation is exactly what is behind the notion of *widening* in abstract interpretation [CC77], where in iterative data flow analysis of programs the effect of executing a loop (in a program) an arbitrary number of times is captured as a

widening operator. In the following we adapt Cousot and Cousot's definition for our situation.

Definition 4.4. *A total function* $\nabla : \mathsf{A} \times \Sigma^+ \to \mathsf{A}$ *is an* acceleration *for an abstraction* $\mathcal{A}_{LTS} = (\mathsf{A}, post_{\mathcal{A}}, \gamma, \sqcup, \Sigma)$ *provided it satisfies the following condition for every* $a \in \mathsf{A}$ *and* $\sigma \in \Sigma^+$*:*

$$\nabla(a, \sigma) \sqsupseteq a \tag{7}$$

An acceleration ∇ *is* sound *if it satisfies the following condition for every* $a \in \mathsf{A}$ *and* $\sigma \in \Sigma^+$*:*

$$\gamma(\nabla(a, \sigma)) \subseteq post^*(\gamma(a)) \tag{8}$$

An acceleration ∇ *is* complete *if it satisfies the following condition for every* $a \in \mathsf{A}$ *and* $\sigma \in \Sigma^+$*:*

$$\gamma(\nabla(a, \sigma)) \supseteq \bigcup_{i \in \mathbb{N}} post(\gamma(a), \sigma^i) \tag{9}$$

The three conditions in the definition above deserve some explanation. Condition (7) — an inclusion condition — ensures that the abstract state obtained after an acceleration is bigger. The definition of a *sound* acceleration, Condition (8), enforces the constraint that all states computed by an acceleration are included in $post^*$. Similarly, the definition of a *complete* acceleration, Condition (9) enforces the constraint that all states resulting from an arbitrary number of executions of σ are included in result of the acceleration.

Given an acceleration we suggest developing a reachability tree in an uniform fashion according to the algorithm below. Most of the details of this algorithm

Algorithm 1 `SymbolicTree`$(\mathcal{A}_{LTS}, a_0, \nabla)$

Input: an abstraction $\mathcal{A}_{LTS} = (\mathsf{A}, post_{\mathcal{A}}, \gamma, \sqcup, \Sigma)$, an initial abstract state $a_0 \in \mathsf{A}$ and an acceleration ∇ for $\mathcal{A}_{LTS}$.

1: create root $\mathtt{r}$ labelled with a_0
2: **while** there are unmarked nodes **do**
3: pick an unmarked node $\mathtt{n}{:}a$
4: **if** $a \sqsubseteq \bigsqcup\{b \mid \mathtt{m}{:}b \text{ is a marked node}\}$ **then**
5: skip {the node $\mathtt{n}$ is covered}
6: **else**
7: $b \leftarrow a$
8: **for each** ancestor $\mathtt{m}$ of $\mathtt{n}$ **do**
9: $b \leftarrow \nabla(b, \sigma)$ where $\mathtt{m} \xrightarrow{\sigma} \mathtt{n}$
10: **for each** label $l \in \Sigma$ **do**
11: $a' \leftarrow post_{\mathcal{A}}(b, l)$
12: construct a son $\mathtt{n}'{:}a'$ of $\mathtt{n}$ and label the arc $\mathtt{n} \xrightarrow{l} \mathtt{n}'$
13: mark $\mathtt{n}$

should be clear. The acceleration gets applied, if at all possible, to obtain a potentially infinite set of reachable states. We may show that this symbolic tree, T, captures the reachable states, as required $post^*(\gamma(a_0)) \subseteq \bigcup_{\mathtt{n}:a\in T} \gamma(a)$. If moreover ∇ is a sound acceleration then we have $post^*(\gamma(a_0)) = \bigcup_{\mathtt{n}:a\in T} \gamma(a)$.

Remark 4.5. If $(\mathsf{A}, \sqsubseteq)$ satisfies the ascending chain condition then the `Symbolic Tree` algorithm terminates.

5 Acceleration of FIFO Automata Using Linear Regular Expressions

For any loop σ on some control state q, we define two total functions (also written) σ and σ^* from $2^{(M^*)^C}$ to $2^{(M^*)^C}$ as follows:

$$\sigma(X) = \{\mathsf{w} \in (M^*)^C \mid (q, \mathsf{w}) \in post(\{q\} \times X, \sigma)\}, \text{ and, } \sigma^*(X) = \bigcup_{i\in\mathbb{N}} \sigma^i(X)$$

We will use the notion of $\mathcal{F}$*-representable* loops to define a canonical acceleration to any abstraction.

Definition 5.1. *A loop σ is* $\mathcal{F}$-representable *if for every $f \in \mathsf{F}$, there exists $g \in \mathsf{F}$ such that $[\![g]\!] = \sigma^*([\![f]\!])$.*

When σ is $\mathcal{F}$-representable we write $g = \sigma^*(f)$ even if there is not a unique g such that $[\![g]\!] = \sigma^*([\![f]\!])$. Notice that $\sigma^*(f)$ is well defined with respect to $\cong$.

Definition 5.2. *Let $\nabla_{\mathcal{F}} : \mathsf{A} \times \Sigma^+ \to \mathsf{A}$ be defined by:*

$$\nabla_{\mathcal{F}}(\mathsf{a}, \sigma) = \begin{cases} \mathsf{a}[q := \sigma^*(\mathsf{a}[q])] & \text{if } \sigma \text{ is a } \mathcal{F}\text{-representable loop on } q \\ \mathsf{a} & \text{otherwise} \end{cases}$$

We now show that $\nabla_{\mathcal{F}}$ is indeed an acceleration.

Proposition 5.3. *The two following statements hold:*

i) the total function $\nabla_{\mathcal{F}}$ is a sound acceleration for $\mathcal{A}^{\mathcal{F}}_{LTS(F)}$, and,
ii) If every loop of F is $\mathcal{F}$-representable then $\nabla_{\mathcal{F}}$ is complete.

In the rest of this section, we focus on accelerations for $\mathcal{LRE}$ based abstractions. We first define a sound and complete abstraction for FIFO automata with one channel. We then define a sound acceleration and a complete acceleration for FIFO automata with multiple channels.

5.1 Acceleration for One Channel

The generic tree construction algorithm, from Section 4, suggests that we should accelerate a sequence σ if we can show that σ can be repeated infinitely often. To that end, consider that a FIFO automaton moves from (q, x) to (q, y) on a execution sequence σ. If it was possible for σ to be repeated infinitely often, then the only messages being removed from, or being added to the channel, come from σ. Clearly, the order in which messages are received, or sent, is preserved. The following theorem was used, but not proved, in [FM96]. The proof is technical and it can be found in the complete paper [FIS00].

Theorem 5.4. *Every loop of a FIFO automaton with one channel is $\mathcal{LRE}$-representable.*

Note that from [BH97], we know that every loop of a FIFO automaton is $\mathcal{CQDD}$-representable (see Theorem 6.5). However, their proof extensively use Presburger formulae (even for one channel) and hence our theorem cannot be deduced from [BH97]. A straightforward consequence of Theorem 5.4 is that for any FIFO automaton with one channel F, $\nabla_{\mathcal{LRE}}$ is a sound and complete acceleration for $\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}$.

5.2 Acceleration for Multiple Channels

To characterize $\mathcal{LRE}$-representable loops, we use a condition on loops defined in [BGWW97]. We say that a loop σ is *non counting* if one of the following conditions is satisfied:

i) $|M| > 1$ and there is a channel c such that every send transition in σ operates on c,
ii) $|M| = 1$ and there is at most one channel which is growing with σ.

Theorem 5.5. *A loop σ is $\mathcal{LRE}$-representable iff it is non counting.*

A straightforward consequence of the theorem is that for any FIFO automaton F, $\nabla_{\mathcal{LRE}}$ is a sound acceleration for $\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}$.

We now show how a complete acceleration can be defined for $\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}$. For every loop σ and for every channel $c \in C$, the *projection of σ on c*, denoted by $\sigma_{|c}$, is obtained from σ by replacing every operation on c' with $c' \neq c$ by an ε — the model of FIFO automata can be easily extended with ε transitions and the results we presented so far are easily extended to this more general model. We define $\overline{\sigma^*} : \mathcal{P}_f(LRE(M)^C) \to \mathcal{P}_f(LRE(M)^C)$ by:

$$\overline{\sigma^*}(\chi) = \bigcup_{\mathsf{r} \in \chi} \prod_{c \in C} (\sigma_{|c})^*(\mathsf{r}[c])$$

Lemma 5.6. *For every loop σ and for every $\mathcal{LRE}$ symbolic channel content $\chi \in \mathcal{P}_f(LRE(M)^C)$, we have $\sigma^*(\llbracket\chi\rrbracket) \subseteq \llbracket\overline{\sigma^*}(\chi)\rrbracket$.*

Notation. Given a control state q, an $\mathcal{LRE}$ symbolic channel content χ and a loop σ on q, we say that it is possible to execute σ^ω from (q, χ), written $(q, \chi) \xrightarrow{\sigma^\omega}$, if for every $n \in \mathbb{N}$ there exists $\mathsf{w} \in \llbracket\chi\rrbracket$ and $\mathsf{w}' \in (M^*)^C$ such that $(q, \mathsf{w}) \xrightarrow{\sigma^n} (q, \mathsf{w}')$.

We define the total function $\overline{\nabla}_{\mathcal{LRE}} : (\mathcal{P}_f(LRE(M)^C))^Q \times \Sigma^+ \to (\mathcal{P}_f(LRE(M)^C))^Q$ by:

$$\overline{\nabla}_{\mathcal{LRE}}(\mathsf{a}, \sigma) = \begin{cases} \mathsf{a}[q := \overline{\sigma}^*(\mathsf{a}[q])] & \text{if } \sigma \text{ is a loop on } q \text{ such that } (q, (\mathsf{a}[q])) \xrightarrow{\sigma^\omega} \\ \mathsf{a}[q := \bigsqcup_{i=0}^{k} \sigma^i(\mathsf{a}[q])] & \text{if } \sigma \text{ is a loop on } q \text{ such that } \sigma^k(\mathsf{a}[q]) = \emptyset \\ \mathsf{a} & \text{otherwise} \end{cases}$$

Proposition 5.7. *For any FIFO automaton $F = (Q, C, M, \delta)$ the total function $\overline{\nabla}_{\mathcal{LRE}}$ is a complete acceleration for $\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}$.*

Notice that when $|M| = 1$, a FIFO automaton can be viewed as a Petri Net (or equivalently a Vector Addition System with States) and in this case $\overline{\nabla}_{\mathcal{LRE}}$ is similar to Karp and Miller's strategy to compute a coverability set. Hence, we obtain that for every initial abstract state a_0, $\texttt{SymbolicTree}(\mathcal{A}^{\mathcal{LRE}}_{LTS(F)}, a_0, \overline{\nabla}_{\mathcal{LRE}})$ terminates and computes an abstract state which is a tight over-approximation of $post^*(\gamma(a_0))$.

6 Comparison of LREs, QDDs and CQDDs

We now show how our notion of abstraction and acceleration is applicable to current work on meta-transitions (QDDs and CQDDs).

6.1 QDDs as an Abstraction

In this subsection we present a slightly different abstraction, namely QDDs due to Boigelot and Godfroid [BGWW97]. A QDD D for a FIFO automaton $F = (Q, C, M, \delta)$, where $C = \{c_1, c_2, \cdots c_n\}$ is a deterministic finite automaton operating on the alphabet $C \times M$ and satisfying the following condition: for every accepted word $w \in L(D)$, the projection of w on C is in ${c_1}^* {c_2}^* \cdots {c_n}^*$. Intuitively, a channel content $\mathsf{w} \in (M^*)^C$ is in the set of channel contents represented by D iff there exists $u_1 u_2 \cdots u_n \in L(D)$ such that for every $1 \leq i \leq n$, u_i may be written as $u_i = (c_i, m_1)(c_i, m_2) \cdots (c_i, m_k)$ with $\mathsf{w}[c_i] = m_1 m_2 \cdots m_k$. In [BGWW97], is is proved that sets of channel contents representable by QDDs are exactly those that are recognizable sets. Hence, QDDs correspond to the FIFO symbolic representation based on regular expressions. More formally,

Definition 6.1. *Let $\mathcal{QDD} = (\mathcal{P}_f(REG(M)^C), \emptyset, !!, ??, [\![\cdot]\!], \cup)$, where $??$, $!!$ and $[\![\cdot]\!]$ are three total functions defined as follows:*

$$??(c, m, \chi) = \{\mathsf{r}[c := m^{-1}(\mathsf{r}[c])] \mid \mathsf{r} \in \chi\}$$
$$!!(c, m, \chi) = \{\mathsf{r}[c := \mathsf{r}[c] \cdot m] \mid \mathsf{r} \in \chi\}$$
$$[\![\chi]\!] = \bigcup_{\mathsf{r} \in \chi} \prod_{c \in C} [\![\mathsf{r}[c]]\!]$$

Note that $\mathcal{QDD}$ is clearly a FIFO symbolic representation and hence $\mathcal{A}^{\mathcal{QDD}}_{LTS(F)}$ is an abstraction of $LTS(F)$.

Theorem 6.2 ([BGWW97]). *Let $F = (Q, C, M, \delta)$ be a FIFO automaton. A loop σ is $\mathcal{QDD}$-representable iff it is non counting.*

Let us remark that a loop is $\mathcal{QDD}$-representable iff it is $\mathcal{LRE}$-representable. A straightforward consequence of the theorem is that for any FIFO automaton F, $\nabla_{\mathcal{QDD}}$ is a sound acceleration for $\mathcal{A}^{\mathcal{QDD}}_{LTS(F)}$.

6.2 CQDDs as an Abstraction

A generalization of QDDs called Constrained Queue Decision Diagrams (CQDDs) was considered in [BH97]. Let us retranslate in our formalism the original definition of CQDDs.

Definition 6.3 ([BH97]). *An* accepting component *is a pair* (r, φ) *such that:*

i) $\mathsf{r} \in LRE(M)^C$, *and,*
ii) for every $c \in C$, *the LRE* $\mathsf{r}[c]$ *contains none of the following factors:* $(au)^*a$, u^*v^* *with* $u, v \in M^*$ *and* $a \in M$, *and,*
iii) φ *is a Presburger arithmetic formula with a set of free variables* $X = \{x_c^i \mid c \in C, 1 \leq i \leq n_c\}$ *where* n_c *is the number of occurences of '*$*$*' in* $\mathsf{r}[c]$.

A Constrained Queue Decision Diagram (CQDD) *is a finite set of accepting components and we write* $CQDD(C, M)$ *for the set of CQDDs over* C *and* M.

This definition is more simple than the original definition (stated in [BH97]) given in terms of deterministic restricted simple automata.

We order the occurences of '$*$' in a LRE ρ by reading ρ from left to right. To a LRE ρ with n occurences of '$*$', we associate a function $f_\rho : \mathbb{N}^n \to \Sigma^*$ where $f_\rho(k_1, k_2, \ldots, k_n)$ is the word obtained from ρ by replacing the i^{th} occurence of '$*$' by k_i for every i, $1 \leq i \leq n$. The set of channel contents $[\![\chi]\!]$ associated with a CQDD χ is given by $[\![\chi]\!] = \cup_{(\mathsf{r},\varphi)\in\chi}[\![(\mathsf{r}, \varphi)]\!]$ where $[\![(\mathsf{r}, \varphi)]\!]$ is the set of channel contents $\mathsf{w} \in (M^*)^C$ such that for some valuation $v : X \to \mathbb{N}$ *satisfying* φ, we have $\mathsf{w}[c] = f_{\mathsf{r}[c]}(v(x_c^1), v(x_c^2), \ldots, v(x_c^{n_c}))$, for every $c \in C$. Notice that if φ is *valid* (i.e. satisfied by all valuations) then $[\![(\mathsf{r}, \varphi)]\!] = [\![r]\!]$ is a $\mathcal{LRE}$ set of channel contents.

CQDDs inspite of their power are closed under the operations we need:

Proposition 6.4 ([BH97]). *Given a CQDD* χ, *a channel* c *and a message* m, *there exists two CQDDs* χ' *and* χ'' *such that:*

$$[\![\chi']\!] = \{\mathsf{w}[c := m^{-1}\mathsf{w}[c]] \mid \mathsf{w} \in [\![\chi]\!], m \leq \mathsf{w}[c]\}$$
$$[\![\chi'']\!] = \{\mathsf{w}[c := \mathsf{w}[c]m] \mid \mathsf{w} \in [\![\chi]\!]\}$$

and we write $\chi' = ??(c, m, \chi)$ *and* $\chi'' = !!(c, m, \chi)$.

We are now ready to define the FIFO symbolic representation based on CQDDs. Defining $\mathcal{CQDD} = (CQDD(C, M), \emptyset, !!, ??, [\![\cdot]\!], \cup)$, we get that $\mathcal{CQDD}$ is a FIFO symbolic representation and hence $\mathcal{A}_{LTS(F)}^{\mathcal{CQDD}}$ is an abstraction of $LTS(F)$.

Theorem 6.5 ([BH97]). *Every loop of a FIFO automaton is* $\mathcal{CQDD}$*-representable.*

A straightforward consequence of the theorem is that for any FIFO automaton F, $\nabla_{\mathcal{CQDD}}$ is a sound and complete acceleration for $\mathcal{A}_{LTS(F)}^{\mathcal{CQDD}}$.

6.3 Comparison

The following theorem shows that $\mathcal{LRE}$ expressible sets of channel contents are exactly the intersection between $\mathcal{QDD}$ expressible sets of channel contents and $\mathcal{CQDD}$ expressible sets of channel contents.

Theorem 6.6. *Let L be a subset of $(M^*)^C$. The two following assertions are equivalent:*

i) there exists an $\mathcal{LRE}$ symbolic channel content χ such that $[\![\chi]\!] = L$,
ii) there exists a $\mathcal{QDD}$ symbolic channel content χ and a $\mathcal{CQDD}$ symbolic channel content χ' such that $[\![\chi]\!] = [\![\chi']\!] = L$

	Acceleration	
	one channel	multiple channels
$\mathcal{LRE}$	$\nabla_{\mathcal{LRE}}$: sound, complete	$\nabla_{\mathcal{LRE}}$: sound, non-complete $\overline{\nabla}_{\mathcal{LRE}}$: non-sound, complete
$\mathcal{QDD}$	$\nabla_{\mathcal{QDD}}$: sound, complete	$\nabla_{\mathcal{QDD}}$: sound, non-complete
$\mathcal{CQDD}$	$\nabla_{\mathcal{CQDD}}$: sound, complete	$\nabla_{\mathcal{CQDD}}$: sound, complete

Fig. 2. Comparison of the accelerations

It is readily seen that the $post_{\mathcal{A}^{\mathcal{LRE}}}$ and $\sqcup$ operations can be computed in *linear* time for the $\mathcal{LRE}$ based abstraction. Moreover, we have

Theorem 6.7. *The inclusion and the equivalence problems between SLREs are in* NP $\cap$ coNP.

No precise complexity results are known for $\mathcal{QDD}$ and $\mathcal{CQDD}$, but inclusion for these FIFO symbolic representations requires at least an exponential time.

Compared to QDDs and CQDDs, SLREs seem to have more practical relevance, since they are usually enough to represent sets of FIFO contents and to iterate loops (see Figure 2), and they have a better complexity. Starting from a single initial state of FIFO automaton with one channel, the generic `SymbolicTree` algorithm applied to these three abstractions/accelerations will give the same result. For FIFO automata with multiple channels, $\nabla_{\mathcal{LRE}}$ and $\nabla_{\mathcal{QDD}}$ applied to an $\mathcal{LRE}$ abstract state produce the same regular languages.

Acknowledgements

We thank the referees for their help to improve the paper.

References

[AAB99] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. *Lecture Notes in Computer Science*, 1579:208–222, 1999. 566, 567, 571

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992. 566

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. 4th Int. Symp. Static Analysis, Paris, France, September 1997*, volume 1302, pages 172–186. Springer-Verlag, 1997. 566, 567, 568, 571, 576, 577

[BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO channel systems with nonregular sets of configurations. In *Proc. 24th Int. Coll. Automata, Languages, and Programming (ICALP'97), Bologna, Italy, July 1997*, volume 1256 of *Lecture Notes in Computer Science*, pages 560–570. Springer-Verlag, 1997. 566, 567, 568, 571, 576, 578

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964. 570

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983. 569

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM SIGACT and SIGPLAN, ACM Press, 1977. 566, 567, 573

[Céc98] G. Cécé. Vérification, analyse et approximations symboliques des automates communicants. Thèse ENS de Cachan, January 1998. 567

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998. 567, 568

[FIS00] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems: application to fifo automata. Research Report LSV-2000-6, Lab. Specification and Verification, ENS de Cachan, Cachan, France, June 2000. 567, 575

[FM96] A. Finkel and O. Marcé. A minimal symbolic coverability graph for infinite-state comunicating automata. Technical report, LIFAC, 1996. 567, 575

[FS00] A. Finkel and G. Sutre. Decidability of reachability problems for classes of two counters automata. In *Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, France, Feb. 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 346–357. Springer, 2000. 567

[JJ93] Thierry Jéron and Claude Jard. Testing for unboundedness of fifo channels. *Theoretical Computer Science*, 113(1):93–117, 1993. 567, 571

[Pac87] J. K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Proc. of Protocol Specification, Testing and Verification, VII*, 1987. 567

[PP91] Wuxu Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Transactions on Programming Languages and Systems*, 13(3):399–442, July 1991. 566

A Unifying Approach to Data-Independence*

Ranko Lazić** and David Nowak

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{Ranko.Lazic,David.Nowak}@comlab.ox.ac.uk

Abstract. A concurrent system is data-independent with respect to a data type when the only operation it can perform on values of that type is equality testing. The system can also assign, input, nondeterministically choose, and output such values. Based on this intuitive definition, syntactic restrictions which ensure data-independence have been formulated for a variety of different formalisms. However, it is difficult to see how these are related.

We present the first semantic definition of data-independence which allows equality testing, and its extension which allows constant symbols and predicate symbols. Both are special cases of a definition of when a family of labelled transition systems is parametric. This provides a unified approach to data-independence and its extensions.

The paper also contains two theorems which, given a system and a specification which are data-independent, enable the verification for all instantiations of the data types (and of the constant symbols and the predicate symbols, in the case of the extension) to be reduced to the verification for a finite number of finite instantiations.

We illustrate the applicability of the approach to particular formalisms by a programming language similar to UNITY.

1 Introduction

Model Checking. During the past 20 years, model checking [3] has developed into a key approach to verifying finite-state concurrent systems. It has a number of advantages over traditional approaches that are based on simulation, testing, and deduction. In particular, model checking is automatic, and provides exact counter-examples in case of error. Model checking has been applied successfully to industrial systems.

The three main challenges for model checking are: the state-explosion problem, caused by the fact that the state space of a system may grow exponentially

* The research in this paper was supported in part by the EPSRC Standard Research Grant GR/M32900.

** Also affiliated to the Mathematical Institute, Belgrade. Supported in part by a Junior Research Fellowship at Christ Church, Oxford. Previously supported by a Domus and Harmsworth Senior Scholarship at Merton College, Oxford, and by a scholarship from Hajrija & Boris Vukobrat and Copechim France SA.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 581–596, 2000.

with the number of its parallel components; model checking of parameterised systems, where one or more parameters of a system have large or infinite ranges; and model checking of infinite-state systems. Over the past 10 to 15 years, considerable progress has been made on each of these challenges.

Data-Independence. A system is data-independent with respect to a data type when the only operation it can perform on values of that type is equality testing. The system can also assign, input, nondeterministically choose, and output such values. Many practically important systems are data-independent. For example: communication protocols are usually data-independent with respect to the types of data which they communicate; memory or database systems may be data-independent with respect to the type of values which they store, as well as the type of addresses.

A data type with respect to which a system is data-independent can be treated as a parameter of the system, i.e. as a type variable that can be instantiated by any concrete type. If correctness of the system is important for all instantiations, we get a special case of the problem of model checking parameterised systems. A simpler case is when correctness is important only for an infinite instantiation (for example, the type of natural numbers), which gives us a special case of the problem of model checking infinite-state systems.

Techniques for model checking data-independent systems have been developed for a number of different frameworks [23,10,9,7,8,12]. They enable verification for all instantiations of the data type to be reduced, either statically or dynamically, to the verification for a finite number of finite instantiations (or only one). Consequently, in the special case of data-independent systems, the challenges of model checking parameterised and infinite-state systems have been overcome.

Recently, it has been shown that data-independence techniques can be combined with induction to yield powerful methods for model checking of systems which are parameterised by their network topology [5], and that they can be applied to enable the verification of security protocols (which involves infinite-state systems) by model checking [19].

This Paper. Based on the intuitive definition of data-independence stated above, it is straightforward to formulate syntactic restrictions which ensure that a system or a specification is data-independent with respect to a data type. In [10,9,7,8,12], such syntactic restrictions are given for each of the formalisms used. However, it is hard to see how the restrictions and the results for different formalisms are related.

Data-independence therefore needs to be defined semantically. This has turned out to be difficult, since a satisfactory definition must not be stronger than the syntactic restrictions, but it has to be strong enough to prove results which enable reductions to finite instantiations. The only semantic definition in the literature is in [23], and it does not allow equality testing.

In this paper, we present a semantic definition of data-independence, in which we can choose whether to allow equality testing or not. The definition is a special case of a definition of what it means for a family of labelled transition systems to be parametric. Intuitively, systems that use type variables and operations on them (which can be thought of as abstract data types) in a polymorphic manner give rise to parametric families of LTSs, where each LTS is the semantics of the system with respect to an instantiation of the type variables and the operations. In the special case where the operations are only equality tests, the family of LTSs is data-independent.

The fact that the definition of data-independence is a special case of the definition of parametricity makes it easy to extend. We demonstrate this by extending it to allow constant symbols whose types are type variables, and predicate symbols which are functions from type variables to fixed non-empty finite types.

The definitions also apply to Rabin automata [20], by regarding them as a subclass of labelled transition systems.

Using the definition of data-independence and its extension, we prove two theorems which enable reductions to finite numbers of finite instantiations of the type variables (and the constants and predicates, in the case of the extension).

We may consider this to be a unifying approach to data-independence (and its extensions). Firstly, the definitions and theorems are not in terms of any particular formalism, but are semantic. Secondly, a wide range of specification frameworks, such as linear-time temporal logic [22] and CSP refinement [18], can be translated into labelled transition systems and Rabin automata. In particular, this is possible for each of the frameworks in [23,9,7,8,12].

The approach is therefore a solid basis for future work. Moreover, in comparison with the formalism-specific approaches, it allows one to draw and to exploit connections between data-independence and parametricity [17], abstract interpretation [4], and relational data refinement [11] more clearly and more deeply.

To illustrate applying the approach to particular formalisms, we consider a programming language similar to UNITY [1]. A way of applying it to linear-time temporal logic is presented in the full version of the paper [13].

Comparisons with Related Work. As we stated, the only semantic definition of data-independence in the literature [23] does not allow equality testing. Moreover, if it is weakened by restricting the functions to be injective, so that it does allow equality testing, it becomes too weak to imply results which enable reductions to finite instantiations.

In [16], data-insensitive programs are defined semantically as those that have finite simulation quotients which preserve the values of all control variables. However, most such programs are not data-independent, and the definition applies only to programs in which the relevant data types are infinite.

Theorem 2 in this paper is at least as powerful as the data-independence results in [9], the main results in [7], the general result in [8], and the main results in [12,15]. Moreover, nondeterministic sequential Rabin automata that

may be infinite-state and that may be defined using state variables, which provide the specification framework in this paper, are more expressive than each of the specification frameworks in [9,7,8,12,15].

Organisation. Section 2 is devoted to preliminaries. In Section 3, we define what it means for a family of labelled transition systems to be parametric (which also applies to families of Rabin automata), and then define data-independence and its extension by constants and predicates as special cases. The two reduction theorems are presented in Section 4. Section 5 considers a UNITY-like programming language. Section 6 contains some directions for future work.

Due to space limitations, proofs are either outlined briefly or omitted. They can be found in the appendix of the full version of the paper [13]. The full version also contains additional examples and explanations.

2 Preliminaries

Tuples and Maps. For any tuple $\underline{a} = (a_1, \ldots, a_n)$ and $i \in \{1, \ldots, n\}$, let $\pi_i(\underline{a}) = a_i$.

For any map f, any mutually distinct $a_1, \ldots, a_n$, and any $b_1, \ldots, b_n$, let $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ be the map which agrees with f except that it maps each a_i to b_i.

We also use the notation $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ to mean the map which maps each a_i to b_i.

Syntax of Types. In addition to type variables, we consider sum types, product types, inductive types (which can express types of finite lists or finite trees), and function types. These will act as types of operations (for example, $(X \times X) \to \mathit{Bool}$ for equality testing on a type variable X) in signatures of families of labelled transition systems or Rabin automata, and as types of states and transition labels of such families.

We assume $\mathit{TypeVars}$ is an infinite set of names for type variables.

The syntax of types is defined by

$$T ::= X \in \mathit{TypeVars} \mid T_1 + \cdots + T_n \mid T_1 \times \cdots \times T_n \mid \mu X.T \mid T_1 \to T_2$$

where n ranges over nonnegative integers, and where in any $\mu X.T$, any free occurence of X in T must not be within a $T_1 \to T_2$ construct.

For any type T, we write $\mathit{Free}(T)$ for the set of all type variables which occur free in T.

Semantics of Types. A *set map* is a partial map from $\mathit{TypeVars}$ to sets, whose domain is finite.

For any type T, and any set map δ such that $\mathit{Free}(T) \subseteq \mathit{Domain}(\delta)$, we write $[\![T]\!]_\delta$ for the denotational semantics of T with respect to δ.

This is defined by

$$\begin{aligned}
[\![X]\!]_\delta &= \delta[\![X]\!] \\
[\![T_1 + \cdots + T_n]\!]_\delta &= \{1\} \times [\![T_1]\!]_\delta \cup \cdots \cup \{n\} \times [\![T_n]\!]_\delta \\
[\![T_1 \times \cdots \times T_n]\!]_\delta &= [\![T_1]\!]_\delta \times \cdots \times [\![T_n]\!]_\delta \\
[\![\mu X.T]\!]_\delta &= \bigcup_{k \in \mathbb{N}} \mathcal{F}^k(\{\}), \ \textit{where } \mathcal{F}(A) = [\![T]\!]_{\delta[X \mapsto A]} \\
[\![T_1 \to T_2]\!]_\delta &= [\![T_1]\!]_\delta \to [\![T_2]\!]_\delta
\end{aligned}$$

Some Standard Types. We define some standard types by abbreviations:

$$\begin{aligned}
\mathit{Empty} &= \textit{(the empty sum type)} \\
\mathit{Single} &= \textit{(the empty product type)} \\
\mathit{Bool} &= \mathit{Single} + \mathit{Single} \\
\mathit{Enum}_l &= \overbrace{\mathit{Single} + \cdots + \mathit{Single}}^{l}, \ \textit{for any } l \in \mathbb{N} \\
\mathit{Nat} &= \mu X.\mathit{Single} + X
\end{aligned}$$

We also define:

$$\overline{\mathit{false}} = (1, ()) \qquad \overline{\mathit{true}} = (2, ()) \qquad \overline{\mathit{enum}_i} = (i, ())$$
$$\overline{0} = (1, ()) \qquad \overline{i+1} = (2, \overline{i})$$

so that we have:

$$[\![\mathit{Bool}]\!]_{\{\}} = \{\overline{\mathit{false}}, \overline{\mathit{true}}\} \qquad [\![\mathit{Enum}_l]\!]_{\{\}} = \{\overline{\mathit{enum}_1}, \ldots, \overline{\mathit{enum}_l}\}$$
$$[\![\mathit{Nat}]\!]_{\{\}} = \{\overline{0}, \overline{1}, \ldots\}$$

Logical Relations. Intuitively, given a system (or specification) which uses type variables and operations on them, it gives rise to a separate labelled transition system (or Rabin automaton) for each instantiation of the type variables and the operations. Supposing we have two such instantiations and, for each type variable, a relation between the two sets by which it is instantiated, logical relations [17] enable us to lift those relations to more complex types. In particular, they will enable us to relate the two instantiations of each of the operations, and to relate states and transition labels in the two labelled transition systems (or Rabin automata).

A *relation map* is a triple (ρ, δ, δ') such that

- ρ is a partial map from *TypeVars* to relations, whose domain is finite,
- δ and δ' are set maps with the same domain as ρ, and
- $\rho[\![X]\!]$ is a relation between $\delta[\![X]\!]$ and $\delta'[\![X]\!]$, for each $X \in \mathit{Domain}(\rho)$.

Given a relation map (ρ, δ, δ'), it determines a logical relation indexed by the types T such that $\mathit{Free}(T) \subseteq \mathit{Domain}(\rho)$. For any such type T, we write $[\![T]\!]_{(\rho,\delta,\delta')}$

for the component at T of the logical relation. This is a relation between the sets $[\![T]\!]_\delta$ and $[\![T]\!]_{\delta'}$, and is defined by

$$
\begin{array}{l}
[\![X]\!]_{(\rho,\delta,\delta')} = \rho[\![X]\!] \\
[\![T_1 + \cdots + T_n]\!]_{(\rho,\delta,\delta')} = \{((i,a),(i',a')) \mid i = i' \ \wedge \ a[\![T_i]\!]_{(\rho,\delta,\delta')}a'\} \\
[\![T_1 \times \cdots \times T_n]\!]_{(\rho,\delta,\delta')} = \{(\underline{a},\underline{a'}) \mid \forall i \in \{1,\ldots,n\}.\pi_i(\underline{a})[\![T_i]\!]_{(\rho,\delta,\delta')}\pi_i(\underline{a'})\} \\
[\![\mu X.T]\!]_{(\rho,\delta,\delta')} = \bigcup_{k\in\mathbb{N}} R_k, \ \ \textit{where} \\
\quad \mathcal{H}(R,A,A') = ([\![T]\!]_{\rho[X\mapsto R],\delta[X\mapsto A],\delta'[X\mapsto A']}, [\![T]\!]_{\delta[X\mapsto A]}, [\![T]\!]_{\delta'[X\mapsto A']}) \\
\quad (R_0,A_0,A'_0) = (\{\},\{\},\{\}) \\
\quad (R_{k+1},A_{k+1},A'_{k+1}) = \mathcal{H}(R_k,A_k,A'_k) \\
[\![T_1 \to T_2]\!]_{(\rho,\delta,\delta')} = \{(f,f') \mid a[\![T_1]\!]_{(\rho,\delta,\delta')}a' \ \Rightarrow \ f(a)[\![T_2]\!]_{(\rho,\delta,\delta')}f'(a')\}
\end{array}
$$

Labelled Transition Systems. A *labelled transition system (LTS)* is a tuple $\mathcal{S} = (A, B, I, \longrightarrow)$ such that A and B are sets, $I \subseteq A$ and $\longrightarrow \subseteq A \times B \times A$.

We say that A is the set of states, B is the set of transition labels, I is the set of initial states, and $\longrightarrow$ is the transition relation.

$a \in A$ is *reachable* iff it can be reached from some $a_0 \in I$ by finitely many transitions.

$\mathcal{S}$ is *deterministic* iff, for each reachable $a \in A$, $\forall b \in B.|\{a' \mid a \xrightarrow{b} a'\}| \leq 1$.

$\mathcal{S}$ is *complete* iff, for each reachable $a \in A$, $\forall b \in B.|\{a' \mid a \xrightarrow{b} a'\}| \geq 1$.

A *run* of $\mathcal{S}$ is an infinite sequence $\underline{a} = (a_0, a_1, \ldots)$ such that $a_0 \in I$ and $\forall i \in \mathbb{N}.\exists b_i.a_i \xrightarrow{b_i} a_{i+1}$. For any infinite sequence $\underline{b} = (b_0, b_1, \ldots)$ which satisfies the previous formula, we say that $\underline{a}$ is a run of $\mathcal{S}$ on $\underline{b}$ from a_0.

Generalised Rabin Automata. The kind of automata we work with in this paper is a generalisation of (sequential) Rabin automata [20]. The generalisation consists of allowing infinitely many states, infinitely many transition labels (i.e. infinite alphabets), sets of initial states rather than only one, and nondeterministic transition relations. These are sometimes convenient, both in theory and in applications, and they do not complicate the main developments in the paper.

We regard the generalised Rabin automata to be the subclass of LTSs in which states are $(1 + 2q)$-tuples, where the final $2q$ components are booleans. The booleans are thought of as q pairs of red and green lights, so that the jth red (respectively, green) light is considered to be on in a state iff the $(2j-1)$th (respectively, $2j$th) boolean is $\overline{true}$. By adopting this presentation, we can state a number of developments only for LTSs, and have the versions for generalised Rabin automata as special cases. In particular, this is the case with the definitions of parametricity, data-independence, and data-independence with constants and predicates.

The main motivation for working with Rabin automata in this paper is that for any linear-time temporal logic formula, a corresponding deterministic Rabin automaton can be constructed [22,20], which is significant for applicability of the theorems in Section 4. Although Muller automata [20] have the same property, they are less convenient for regarding as a subclass of LTSs.

Formally, a *generalised Rabin automaton (GRA)* is an LTS $\mathcal{R} = (D, A, J, \longrightarrow)$ such that $D = C \times (\llbracket Bool \rrbracket_{\{\}})^{2q}$ for some set C and some natural number q.

A run $\underline{d} = (d_0, d_1, \ldots)$ of $\mathcal{R}$ is *successful* iff there exists $j \in \{1, \ldots, q\}$ such that $\pi_{2j-1}(h_i) = \overline{true}$ for only finitely many i and $\pi_{2j}(h_i) = \overline{true}$ for infinitely many i, where $d_i = (c_i, h_i)$.

An infinite sequence $\underline{a}$ of elements of A is *accepted* by $\mathcal{R}$ iff, for each $d_0 \in J$, there exists a run of $\mathcal{R}$ on $\underline{a}$ from d_0 which is successful.

Satisfaction of a GRA by an LTS. Given an LTS $\mathcal{S}$ and a GRA $\mathcal{R}$ such that the set of states of $\mathcal{S}$ equals the set of transition labels of $\mathcal{R}$, we say that $\mathcal{S}$ *satisfies* $\mathcal{R}$ iff each run of $\mathcal{S}$ is accepted by $\mathcal{R}$.

3 Defining Data-Independence

We will define data-independence as a special case of parametricity. The idea is that, given a system (or specification) which uses a number of type variables and operations on them (which can be thought of as abstract data types), its semantics is a family of LTSs (or GRAs), which consists of an LTS (or GRA) for each instantiation of the type variables by sets and of the operation symbols by operations on those sets. If the system (or specification) uses the type variables and the operations in a polymorphic fashion, the family will be parametric. Data-independence will be the special case where the operations are only equality tests.

We will use the term 'signature' to mean a finite set Υ of type variables together with a finite set Γ of typed operation symbols, where the types in Γ do not contain free occurences of any type variables outside Υ.

Definition 1 (signature, instantiation).

(a) *We assume TermVars is an infinite set of names for term variables.*
A type map *is a partial map from TermVars to types, whose domain is finite.*
A signature *is an ordered pair (Υ, Γ), where*
- *Υ is a finite subset of TypeVars, and*
- *Γ is a type map such that $Free(\Gamma(x)) \subseteq \Upsilon$ for each $x \in Domain(\Gamma)$.*

(b) *A* value map *is a map whose domain is a finite subset of TermVars.*
Given a type map Γ and a set map δ such that $Free(\Gamma(x)) \subseteq Domain(\delta)$ for each $x \in Domain(\Gamma)$, we say that a value map η is with respect to Γ and δ *iff*
- *$Domain(\eta) = Domain(\Gamma)$, and*
- *$\eta\llbracket x \rrbracket \in \llbracket \Gamma(x) \rrbracket_\delta$ for each $x \in Domain(\Gamma)$.*

An instantiation *of a signature (Υ, Γ) is an ordered pair (δ, η) such that*
- *δ is a set map whose domain is Υ, and*
- *η is a value map with respect to Γ and δ.* □

We can now define families of LTSs and GRAs. It is not necessary that they are indexed by all instantiations of a signature — for example, a system whose signature is $(\{X\}, [leq \mapsto ((X \times X) \to Bool)])$ may make sense only for instantiations (δ, η) such that $\eta[\![leq]\!]$ is a partial ordering.

In any family of LTSs (or GRAs), there is a type of states and a type of transition labels which are shared by all members of the family. Intuitively, these are the types of states and transition labels of the program whose semantics is the family. In a family of GRAs which is not generated by a program but by a linear-time temporal logic formula [22,20], the type of states may be a closed type, in which case its semantics is the same for each instantiation of the signature.

Although transition labels in an LTS and states (other than their lights) in a GRA are irrelevant for whether the former satisfies the latter, types of transition labels in families of LTSs and types of states in families of GRAs will play a role in relating members which correspond to different signature instantiations, and thus in defining parametric and data-independent families.

Definition 2 (families of LTSs and GRAs).

(a) A family of LTSs *is a tuple* $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *such that*
- (Υ, Γ) *is a signature,*
- T *and* U *are types with* $Free(T) \subseteq \Upsilon$ *and* $Free(U) \subseteq \Upsilon$*,*
- $\mathcal{I}$ *is a class*[1] *of instantiations of* (Υ, Γ)*, and*
- $\underline{\mathcal{S}}$ *consists of an LTS* $\mathcal{S}_{(\delta,\eta)}$ *whose set of states is* $[\![T]\!]_\delta$ *and whose set of transition labels is* $[\![U]\!]_\delta$*, for each* $(\delta, \eta) \in \mathcal{I}$*.*

(b) A family of GRAs *is a family of LTSs* $(\Upsilon, \Gamma, W, T, \mathcal{I}, \underline{\mathcal{R}})$ *such that* W *is of the form* $V \times Bool^{2q}$*.* □

A family of LTSs will be parametric iff, for any two instantiations (δ, η) and (δ', η') of its signature (Υ, Γ), and any relation map (ρ, δ, δ') on Υ which relates (δ, η) and (δ', η'), $[\![T]\!]_{(\rho,\delta,\delta')}$ is a certain kind of bisimulation with respect to $[\![U]\!]_{(\rho,\delta,\delta')}$ between the members of the family corresponding to (δ, η) and (δ', η'), where T is the type of states and U the type of transition labels.

The required kind of bisimulation is 'universal partial $[\![U]\!]_{(\rho,\delta,\delta')}$-bisimulation', and it is defined below. $[\![U]\!]_{(\rho,\delta,\delta')}$ acts as the relation between the sets of transition labels of the two LTSs. The bisimulation is called 'partial' because $[\![U]\!]_{(\rho,\delta,\delta')}$ may be a partial relation, and it is called 'universal' because it treats initial states and transition labels in a universal-quantifier manner.

Definition 3 (related instantiations). *Suppose*

- (δ, η) *and* (δ', η') *are two instantiations of a signature* (Υ, Γ)*, and*
- (ρ, δ, δ') *is a relation map.*

We say that (ρ, δ, δ') relates (δ, η) *and* (δ', η') *iff*

$$\forall x \in Domain(\Gamma).\eta[\![x]\!][\![\Gamma(x)]\!]_{(\rho,\delta,\delta')}\eta'[\![x]\!] \quad \square$$

[1] We allow $\mathcal{I}$ to be a class rather than only a set because Definitions 7 and 8 will require $\mathcal{I}$ to be as large as the class of all sets. However, this foundational issue is not important in this paper.

Definition 4 (universal partial R-bisimulation). *Suppose*

- $\mathcal{S} = (A, B, I, \longrightarrow)$ *is an LTS,*
- $\mathcal{S}' = (A', B', I', \longrightarrow')$ *is an LTS,*
- P *is a relation between* A *and* A'*, and*
- R *is a relation between* B *and* B'*.*

We say that P *is a* universal partial R-bisimulation between $\mathcal{S}$ and $\mathcal{S}'$ *iff*

(i) $\forall a, a'.aPa' \Rightarrow (a \in I \Leftrightarrow a' \in I')$,

(ii) $\forall a_1, a_1', b, b', a_2.(a_1 P a_1' \wedge a_1 \xrightarrow{b} a_2 \wedge bRb') \Rightarrow (\exists a_2'.a_1' \xrightarrow{b'}{}' a_2' \wedge a_2 P a_2')$, *and*

(iii) $\forall a_1, a_1', b, b', a_2'.(a_1 P a_1' \wedge a_1' \xrightarrow{b'}{}' a_2' \wedge bRb') \Rightarrow (\exists a_2.a_1 \xrightarrow{b} a_2 \wedge a_2 P a_2')$. □

Definition 5 (related LTSs). *Suppose*

- T *and* U *are types,*
- (ρ, δ, δ') *is a relation map such that* $Free(T), Free(U) \subseteq Domain(\rho)$,
- $\mathcal{S}$ *is an LTS with set of states* $[\![T]\!]_\delta$ *and set of transition labels* $[\![U]\!]_\delta$*, and*
- $\mathcal{S}'$ *is an LTS with set of states* $[\![T]\!]_{\delta'}$ *and set of transition labels* $[\![U]\!]_{\delta'}$*.*

We say that (ρ, δ, δ') relates $\mathcal{S}$ *and* $\mathcal{S}'$ *iff* $[\![T]\!]_{(\rho,\delta,\delta')}$ *is a universal partial* $[\![U]\!]_{(\rho,\delta,\delta')}$*-bisimulation between* $\mathcal{S}$ *and* $\mathcal{S}'$*.* □

Definition 6 (parametric family of LTSs). *A family of LTSs* $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is* parametric *iff, for any* (δ, η) *and* (δ', η') *from* $\mathcal{I}$*, and any relation map* (ρ, δ, δ')*, we have*

$$(\rho, \delta, \delta') \text{ relates } (\delta, \eta) \text{ and } (\delta', \eta') \;\Rightarrow\; (\rho, \delta, \delta') \text{ relates } \mathcal{S}_{(\delta,\eta)} \text{ and } \mathcal{S}_{(\delta',\eta')} \quad \square$$

A family of LTSs will be data-independent iff it is parametric and the operations in its signature (Υ, Γ) are only equality tests. More precisely, we define data-independence with respect to a subset Υ' of Υ, so that equality testing is allowed on type variables in Υ', but not allowed on those in $\Upsilon \setminus \Upsilon'$. The relevant instantiations of (Υ, Γ) are all those in which the equality-test symbols are instantiated by equalities.

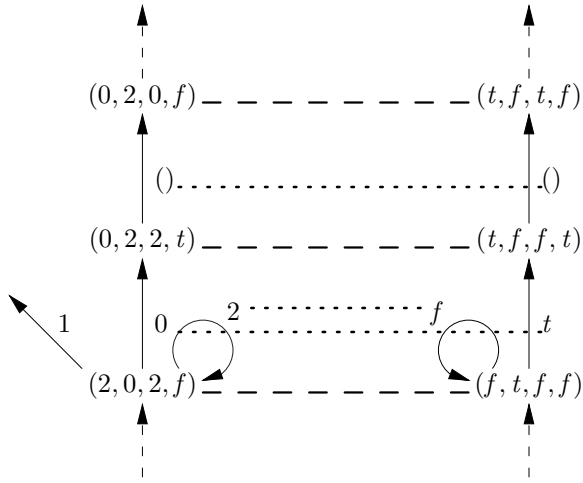
Definition 7 (data-independence). *Suppose* $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is a family of LTSs, and* $\Upsilon' \subseteq \Upsilon$*.*

$(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is* data-independent with Υ'-equalities *iff*

(i) $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is parametric,*
(ii) $Domain(\Gamma) = \{e_X \mid X \in \Upsilon'\}$,
(iii) $\Gamma(e_X) = ((X \times X) \to Bool)$*, and*
(iv) $\mathcal{I}$ *consists of all instantiations* (δ, η) *of* (Υ, Γ) *such that* $\eta[\![e_X]\!]$ *is the equality on* $\delta[\![X]\!]$ *for each* $X \in \Upsilon'$*.* □

Example 1. Consider a system which removes adjacent duplicates from an input stream, where the type of data is a type variable X. This system can be represented by a family of LTSs $(\{X\}, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ which is data-independent with $\{X\}$-equalities. Γ and $\mathcal{I}$ are given by Definition 7, $T = X \times X \times X \times \mathit{Bool}$, and $U = \mathit{Single} + X$. The first two components of a state store the two most recent outputs (partly to make possible a specification in linear-time temporal logic), whereas the remaining two components model an output channel. Transition labels of type X represent inputs.

Let δ and δ' be set maps which map X to $\{0, 1, 2\}$ and $\{t, f\}$ respectively, and let η and η' be value maps which map e_X to equalities on $\{0, 1, 2\}$ and $\{t, f\}$ respectively. Let ρ be a relation map which maps X to $\{(0, t), (2, f)\}$. Since both $\rho[\![X]\!]$ and its inverse are injective, (ρ, δ, δ') relates (δ, η) and (δ', η'). Therefore, by Definitions 7, 6 and 5, $[\![T]\!]_{(\rho,\delta,\delta')}$ is a universal partial $[\![U]\!]_{(\rho,\delta,\delta')}$-bisimulation between $\mathcal{S}_{(\delta,\eta)}$ and $\mathcal{S}_{(\delta',\eta')}$. This is illustrated on parts of $\mathcal{S}_{(\delta,\eta)}$ and $\mathcal{S}_{(\delta',\eta')}$ in the following figure.



□

The following is an extension (i.e. a weakening) of data-independence. In addition to equality-test symbols, it allows constant symbols whose types are type variables, and predicate symbols which are functions from type variables to fixed non-empty finite types. This extension has been shown important in the recent work on model checking systems which are parameterised by their network structure [5].

Definition 8 (data-independence with constants and predicates). *Suppose* $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is a family of LTSs,* $\Upsilon' \subseteq \Upsilon$, $k : \Upsilon \to \mathbb{N}$ *and* $l : \Upsilon \to (\mathbb{N} \setminus \{0\})$.

$(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is* data-independent with Υ'-equalities, k-constants and l-predicates *iff*

(i) $(\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ *is parametric,*
(ii) $\mathit{Domain}(\Gamma) = \{e_X \mid X \in \Upsilon'\} \cup \{s_{X,i} \mid i \in \{1, \ldots, k(X)\}\} \cup \{p_X \mid X \in \Upsilon\}$,

(iii) $\Gamma(e_X) = ((X \times X) \to Bool)$, $\Gamma(s_{X,i}) = X$ *and* $\Gamma(p_X) = (X \to Enum_{l(X)})$, *and*

(iv) $\mathcal{I}$ *consists of all instantiations* (δ, η) *of* (Υ, Γ) *such that* $\eta[\![e_X]\!]$ *is the equality on* $\delta[\![X]\!]$ *for each* $X \in \Upsilon'$. □

4 Reduction Theorems

For any family of LTSs $\mathcal{F}$ and any type variable X from its signature, $\kappa_X(\mathcal{F})$ (respectively, $\lambda_X(\mathcal{F})$) will denote the supremum of the number of distinct values of type X in any reachable state (respectively, transition label). For these to make sense, it will be assumed that the type of states (respectively, transition labels) contains no function-type constructs.[2] If a supremum is over an empty set, its value is 0. A supremum may also be ∞.

Notation (*Comps*). For any type variable X, any type T which contains no function-type constructs, and any a which is an element of some $[\![T]\!]_\delta$, we write $Comps_{X,T}(a)$ for the set of all components of a which correspond to free occurences of X in T. A formal definition can be found in the full version of the paper [13]. □

Notation (κ). For any family of LTSs $\mathcal{F} = (\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ such that T contains no function-type constructs, and any $X \in \Upsilon$, let

$$\kappa_X(\mathcal{F}) = \sup\{|Comps_{X,T}(a)| \mid a \text{ is a reachable state in some } \mathcal{S}_{(\delta,\eta)}\} \quad \square$$

Notation (λ). For any family of LTSs $\mathcal{F} = (\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ such that U contains no function-type constructs, and any $X \in \Upsilon$, let

$$\lambda_X(\mathcal{F}) = \sup\{|Comps_{X,U}(b)| \mid a_1 \xrightarrow{b} a_2 \text{ and } a_1 \text{ is a reachable state in some } \mathcal{S}_{(\delta,\eta)}\} \quad \square$$

Given a family of LTSs $\mathcal{F}$ (representing a system) and a family of GRAs $\mathcal{G}$ (representing a specification) which have the same signature (Υ, Γ) and are data-independent, and given two instantiations (δ, η) and (δ', η') of (Υ, Γ), the first reduction theorem states that satisfaction for (δ, η) and satisfaction for (δ', η') are equivalent provided, for any $X \in \Upsilon$, the sizes of the sets $\delta[\![X]\!]$ and $\delta'[\![X]\!]$ are either equal or no less than $\kappa_X(\mathcal{F}) + \lambda_X(\mathcal{F}) + \kappa_X(\mathcal{G})$. The types of states and transition labels are assumed to contain no function-type constructs, and the GRAs are assumed to be deterministic and complete. The theorem applies to the weakest kind of data-independence, where equality testing is allowed on all type variables.

If $\kappa_X(\mathcal{F}) + \lambda_X(\mathcal{F}) + \kappa_X(\mathcal{G})$ is finite, the theorem reduces the verification for all instantiations to the verification for a finite number of finite instantiations.

[2] This could be relaxed, in particular to allow subtypes $T_1 \to T_2$ such that T_1 and T_2 do not contain any of the free occurences of X in the overall type.

Theorem 1. *Suppose Υ is a finite subset of TypeVars.*

Suppose T, U and V are types such that $Free(T) \subseteq \Upsilon$, $Free(U) \subseteq \Upsilon$ and $Free(V) \subseteq \Upsilon$, and T, U and V contain no function-type constructs.

Suppose $\mathcal{F} = (\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ is a family of LTSs which is data-independent with Υ-equalities.

Suppose $\mathcal{G} = (\Upsilon, \Gamma, V \times Bool^{2q}, T, \mathcal{I}, \underline{\mathcal{R}})$ is a family of GRAs which is data-independent with Υ-equalities, and such that, for any $(\delta, \eta) \in \mathcal{I}$, $\mathcal{R}_{(\delta,\eta)}$ is deterministic and complete.

If $(\delta, \eta), (\delta', \eta') \in \mathcal{I}$ are such that, for any $X \in \Upsilon$, either $|\delta[\![X]\!]| = |\delta'[\![X]\!]|$ or

$$|\delta[\![X]\!]|, |\delta'[\![X]\!]| \geq \kappa_X(\mathcal{F}) + \lambda_X(\mathcal{F}) + \kappa_X(\mathcal{G})$$

then $\mathcal{S}_{(\delta,\eta)}$ satisfies $\mathcal{R}_{(\delta,\eta)}$ iff $\mathcal{S}_{(\delta',\eta')}$ satisfies $\mathcal{R}_{(\delta',\eta')}$.

Proof (outline). By forming a family of LTSs which is the composition of $\mathcal{F}$ and $\mathcal{G}$, transforming it so that its type of transition labels is *Single*, and showing that the reachable portions of the symmetry quotients [2] of the members corresponding to (δ, η) and (δ', η') are isomorphic. □

The following theorem generalises the first one by being applicable to data-independence with constants and predicates. For simplicity, it assumes that the GRAs have unique initial states, and considers only instantiations which assign mutually distinct values to the constant symbols.

Theorem 2. *Suppose Υ is a finite subset of TypeVars, $k : \Upsilon \to \mathbb{N}$ and $l : \Upsilon \to (\mathbb{N} \setminus \{0\})$.*

Suppose T, U and V are types such that $Free(T) \subseteq \Upsilon$, $Free(U) \subseteq \Upsilon$ and $Free(V) \subseteq \Upsilon$, and T, U and V contain no function-type constructs.

Suppose $\mathcal{F} = (\Upsilon, \Gamma, T, U, \mathcal{I}, \underline{\mathcal{S}})$ is a family of LTSs which is data-independent with Υ-equalities, k-constants and l-predicates.

Suppose $\mathcal{G} = (\Upsilon, \Gamma, V \times Bool^{2q}, T, \mathcal{I}, \underline{\mathcal{R}})$ is a family of GRAs which is data-independent with Υ-equalities, k-constants and l-predicates, and such that, for any $(\delta, \eta) \in \mathcal{I}$, $\mathcal{R}_{(\delta,\eta)}$ has a unique initial state and is deterministic and complete.

If $(\delta, \eta), (\delta', \eta') \in \mathcal{I}$ are such that

(i) for any $X \in \Upsilon$, $\eta[\![s_{X,1}]\!], \ldots, \eta[\![s_{X,k(X)}]\!]$ are mutually distinct, and $\eta'[\![s_{X,1}]\!], \ldots, \eta'[\![s_{X,k(X)}]\!]$ are mutually distinct,

(ii) for any $X \in \Upsilon$ and any $i \in \{1, \ldots, k(X)\}$, $\eta[\![p_X]\!](\eta[\![s_{X,i}]\!]) = \eta'[\![p_X]\!](\eta'[\![s_{X,i}]\!])$, and

(iii) for any $X \in \Upsilon$ and any $j \in \{1, \ldots, l(X)\}$, either $\nu_{\eta,X,j} = \nu_{\eta',X,j}$ or

$$\nu_{\eta,X,j}, \nu_{\eta',X,j} \geq \kappa_X(\mathcal{F}) + \lambda_X(\mathcal{F}) + \kappa_X(\mathcal{G})$$

where $\nu_{\theta,X,j} = |\theta[\![p_X]\!]^{-1}(\overline{enum_j}) \setminus \{\theta[\![s_{X,i}]\!] \mid i \in \{1, \ldots, k(X)\}\}|$,

then $\mathcal{S}_{(\delta,\eta)}$ satisfies $\mathcal{R}_{(\delta,\eta)}$ iff $\mathcal{S}_{(\delta',\eta')}$ satisfies $\mathcal{R}_{(\delta',\eta')}$.

Proof (outline). By finitely many transformations, this theorem can be reduced to Theorem 1. A transformation either replaces a predicate symbol whose range has size at least 2 by a sum of fresh type variables, or eliminates a constant symbol provided its type has a predicate symbol with the singleton range. □

5 Applications of the Approach

In this section, we show how the approach we presented can be applied to a programming language similar to UNITY [1]. A way of applying it to linear-time temporal logic can be found in the full version of the paper [13].

A program in this language consists of three sections: a declare section which declares state variables, an initially section which specifies the initial condition satisfied by state variables, and a section assign which is a finite collection of guarded assignments. For the sake of simplicity, the language does not contain inductive or function type constructs. We assume the operators f range over a finite set.

$$\begin{aligned} T &::= X \in \mathit{TypeVars} \mid T_1 + \cdots + T_n \mid T_1 \times \cdots \times T_n \\ \mathit{decl} &::= x_1{:}T; \ldots; x_n{:}T \\ \mathit{expr} &::= x \in \mathit{TermVars} \mid f(\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid e_X(\mathit{expr}_1, \mathit{expr}_2) \\ \mathit{assign} &::= [y{:}T] : \mathit{expr} :: x_1, \ldots, x_n := \mathit{expr}_1, \ldots, \mathit{expr}_n \\ \mathit{assigns} &::= \mathit{assign}_1 \parallel \cdots \parallel \mathit{assign}_p \\ \mathit{progr} &::= \mathsf{declare}\ \mathit{decl}\ \mathsf{initially}\ \mathit{expr}\ \mathsf{assign}\ \mathit{assigns}\ \mathsf{end} \end{aligned}$$

where e_X is the equality-test operator on X, i.e. its semantics is the equality on the semantics of X. In the guarded assignments, $[y{:}T]$ is a typed parameter, expr is a guard (a boolean expression), and $x_1, \ldots, x_n := \mathit{expr}_1, \ldots, \mathit{expr}_n$ is a simultaneous assignment.

Notation (signature, state type, action type). Suppose P is a program. Its signature (cf. Definition 1) is $(\Upsilon(P), \Gamma(P))$ where

- $\Upsilon(P)$ is the set of all type variables which occur free in P and
- $\Gamma(P)$ is the set of operators (f and e_X in the syntax) such that their types involve free type variables.

$\mathit{StateType}(P)$ is the product of the types of the state variables of P. $\mathit{ActType}(P)$ is the sum of the types of the parameters of P. □

We write $[\![P]\!]$ for the operational semantics of a program P. It is a family of LTSs $(\Upsilon(P), \Gamma(P), \mathit{StateType}(P), \mathit{ActType}(P), \mathcal{I}, \underline{\mathcal{S}})$ (cf. Definition 2) where $\mathcal{I}$ is the class of all instantiations (δ, η) of $(\Upsilon(P), \Gamma(P))$ such that each $\eta[\![e_X]\!]$ is the equality on $\delta[\![X]\!]$ (cf. Definition 1) and $\underline{\mathcal{S}}$ consists of an LTS $[\![P]\!]_{(\delta,\eta)}$ for each $(\delta, \eta) \in \mathcal{I}$. (A formal definition can be found in [13].)

Proposition 1. *For any program P, the family of LTSs $[\![P]\!]$ is parametric.* □

Definition 9. *Suppose P is a program and Υ' is a subset of $\Upsilon(P)$. P is said to be* syntactically data-independent with Υ'-equalities *if and only if $\Gamma(P)$ only contains equality-test operators on type variables from Υ', i.e. $\Gamma(P)$ is equal to $\{e_X \mid X \in \Upsilon'\}$.* □

It is clear that syntactic data-independence is decidable.

Proposition 2. *Suppose P is a program and Υ' is a subset of $\Upsilon(P)$. If P is syntactically data-independent with Υ'-equalities, then the family of LTSs $[\![P]\!]$ is data-independent with Υ'-equalities.* □

We show how to compute an upper approximation of the bound stated by Theorem 1.

Definition 10. *For any type variable X and any type T which contains no inductive or function type constructs, $k_{X,T}$ is defined by*

$$k_{X,Y} = \begin{cases} 1, & \text{if } Y = X \\ 0, & \text{if } Y \neq X \end{cases} \qquad \begin{array}{l} k_{X,T_1+\cdots+T_n} = max\{k_{X,T_i} \mid 1 \leq i \leq n\} \\ k_{X,T_1\times\cdots\times T_n} = \Sigma_{i=1}^{n} k_{X,T_i} \end{array}$$ □

Proposition 3. *Suppose P is a program and $X \in \Upsilon(P)$.*

$$k_{X,StateType(P)} \geq \kappa_X([\![P]\!]) \text{ and } k_{X,ActType(P)} \geq \lambda_X([\![P]\!])$$ □

Suppose two programs *Sys* (a system) and *Spec* (a specification), and a finite subset Υ of *TypeVars* are such that $\Upsilon(Sys) = \Upsilon(Spec) = \Upsilon$; *Sys* and *Spec* are both data-independent with Υ-equalities; and each $[\![Spec]\!]_{(\delta,\eta)}$ is a deterministic and complete GRA. Then, according to Theorem 1 and Proposition 3, if *Sys* satisfies *Spec* for all instantiations which map each type variable $X \in \Upsilon$ to a set of size lower than or equal to $k_{X,StateType(Sys)}+k_{X,ActType(Sys)}+k_{X,StateType(Spec)}$, then *Sys* satisfies *Spec* for any instantiation.

6 Future Work

One direction for future work is considering other extensions (i.e. weakenings) of data-independence. Another is considering cases where the types of states and transition labels may contain function-type constructs — some progress on this was made in [14]. A third direction is tree automata [20], in order to allow specifications given by formulae in branching-time temporal logics.

Acknowledgements

We are grateful to Antti Valmari for pointing out a shortcoming in a previous version of the paper, to Michael Goldsmith, Bengt Jonsson, David Naumann and Uday Reddy for useful discussions, to Philippa Broadfoot and Bill Roscoe for proof-reading, and to anonymous reviewers for their comments.

References

1. K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, 1988. 583, 593

2. E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In [6], pages 77–104, 1996. Kluwer. 592
3. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. 581
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixed Points. In *Proc. of POPL*, pages 238–252, 1977. ACM. 583
5. S. J. Creese and A. W. Roscoe. Formal Verification of Arbitrary Network Topologies. In *Proc. of PDPTA*, Volume II, 1999. CSREA Press. 582, 590
6. E. A. Emerson (editor). Formal Methods in System Design 9 (1–2) (Special Issue on Symmetry in Automatic Verification). Kluwer, 1996. 595
7. R. Hojati and R. K. Brayton. Automatic Datapath Abstraction In Hardware Systems. In *Proc. of CAV*, volume 939 of LNCS, pages 98–113, 1995. Springer Verlag. 582, 583, 584
8. R. Hojati, D. L. Dill, and R. K. Brayton. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proc. of CHDL*, 1997. 582, 583, 584
9. C. N. Ip and D. L. Dill. Better verification through symmetry. In [6], pages 41–75, 1996. 582, 583, 584
10. B. Jonsson and J. Parrow. Deciding Bisimulation Equivalences for a Class of Non-Finite-State Programs. *Information and Computation*, 107(2):272–302, 1993. 582
11. Y. Kinoshita, P. W. O'Hearn, A. J. Power, and M. Takeyama. An Axiomatic Approach to Binary Logical Relations with Applications to Data Refinement. volume 1281 of LNCS, page 191, 1997. 583
12. R. S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D.Phil. thesis, Oxford University Computing Laboratory, 1999. 582, 583, 584
13. R. S. Lazić and D. Nowak. *A Unifying Approach to Data-independence*. PRG Technical Report, Oxford University Computing Laboratory, 2000. Available at http://www.comlab.ox.ac.uk/oucl/publications/tr/index.html. 583, 584, 591, 593
14. R. S. Lazić and A. W. Roscoe. Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays. In *Proc. of INFINITY*, Report TUM-I9825, Technical University of Munich, 1998. A fuller version is the Technical Report PRG-TR-2-98, Oxford University Computing Laboratory. 594
15. R. S. Lazić and A. W. Roscoe. Data Independence with Generalised Predicate Symbols. In *Proc. of PDPTA*, Volume I, pages 319–325, 1999. CSREA Press. 583, 584
16. K. S. Namjoshi and R. P. Kurshan. *Syntactic Program Transformations for Automatic Abstraction*. In *Proc. of CAV*, volume 1855 of LNCS, 2000. Springer Verlag. 583
17. J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Proc. of the IFIP 9th World Congress*, pages 513–523, 1983. Elsevier Science Publishers B. V. (Norht-Holland). 583, 585
18. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. 583
19. A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. In *J. of Comp. Sec., Special Issue CSFW11*, page 147, 1999. IOS Press. 582
20. W. Thomas. Automata on Infinite Objects. In [21], chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990. 583, 586, 588, 594
21. J. van Leeuwen. *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*. Elsevier Science Publishers B. V., 1990. 595

22. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–345, 1986. IEEE Computer Society Press. 583, 586, 588
23. P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Conference Record of POPL*, pages 184–193, 1986. ACM. 582, 583

Chi Calculus with Mismatch

Yuxi Fu* and Zhenrong Yang

Department of Computer Science
Shanghai Jiaotong University, Shanghai 200030, China

Abstract. The theory of chi processes with the mismatch operator is studied. Two open congruence relations are investigated. These are weak early open congruence and weak late open congruence. Complete systems are given for both congruence relations. These systems use some new tau laws. The results of this paper correct some popular mistakes in literature.

1 Introduction

In recent years several publications have focused on a class of new calculi of mobile processes. These models include χ-calculus ([1,2,3,4]), update calculus ([7]) and fusion calculus ([8,10]). In a uniform terminology they are respectively χ-calculus, asymmetric χ-calculus and polyadic χ-calculus. The χ-calculus has its motivations from proof theory. In process algebraic model of classical proofs there has been no application of mismatch operator. The χ-calculus studied so far contains no mismatch operator. On the other hand the update and fusion calculi have their motivations from concurrent constraint programming. When applying process calculi to model real programming problems one finds very handy the mismatch operator. For that reason the full update and fusion calculi always have the mismatch combinator. Strong bisimulation congruence has been investigated for each of the three models. It is basically the open congruence. A fundamental difference between χ-like calculi and π-like calculi ([5]) is that all names in the former are subject to update whereas local names in the latter are never changed. In terms of the algebraic semantics, it says that open style congruence relations are particularly suitable to χ-like process calculi. Several weak observational equivalence relations have been examined. Fu studied in [1] weak open congruence and weak barbed congruence. It was shown that a sensible bisimulation equivalence on χ-processes must be closed under substitution in every bisimulation step. In χ-like calculi closure under substitution amounts to the same thing as closure under parallel composition and localization. This is the property that led Fu to introduce L-congruences ([2]). These congruence relations form a lattice under inclusion order. It has been demonstrated

* The author is funded by NNSFC (69873032) and the 863 Hi-Tech Project (863-306-ZT06-02-2). He is also supported by BASICS, Center of Basic Studies in Computing Science, sponsored by Shanghai Education Committee. BASICS is affiliated to the Department of Computer Science at Shanghai Jiaotong University.

C. Palamidessi (Ed.): CONCUR 2000, LNCS 1877, pp. 596–610, 2000.

that L-congruences are general enough so as to subsume familiar bisimulation congruences. The open congruence and the barbed congruence for instance are respectively the bottom and the top elements of the lattice. This is also true for asymmetric χ-calculus ([4]). Complete systems have been discovered for L-congruences on both finite χ-processes and finite asymmetric χ-processes ([4]). An important discovery in the work of axiomatizing χ-processes is that Milner's tau laws are insufficient for open congruences. Another basic tau law called T4

$$\tau.P = \tau.(P+[x=y]\tau.P)$$

is necessary to deal with the dynamic aspect of name update. Parrow and Victor have worked on completeness problems for fusion calculus ([8]). The system they provide for the weak hypercongruence for sub-fusion calculus *without* the mismatch operator is deficient because it lacks of the axiom T4. However their main effort in the above mentioned paper is on the full fusion calculus *with* the mismatch operator. This part of work is unfortunately more problematic. To explain what we mean by that we need to take a closer look at hyperequivalence.

Process equivalence is observational in the sense that two processes are deemed to be equal unless a third party can detect a difference between the two processes. Usually the third party is also a process. Now to observe a process is to communicate with it. In process calculi the communication happens if the observer and the observee are composed via a parallel operator. It follows that process equivalences must be closed under parallel composition. Weak hyperequivalence is basically an open equivalence. This relation is fine with the sub-fusion calculus without the mismatch combinator. It is however a bad equivalence for the full fusion calculus for the reason that it is not closed under composition. A simple counter example is as follows: Let $\approx_h$ be the hyperequivalence. Now for distinct names x, y it holds that

$$(x)ax.[x{\neq}y]\tau.P \approx_h (x)ax.[x{\neq}y]\tau.P+(x)ax.P$$

This is because the transition $(x)ax.[x{\neq}y]\tau.P+(x)ax.P \xrightarrow{a(x)} P$ can be simulated by $(x)ax.[x{\neq}y]\tau.P \xrightarrow{a(x)} \xrightarrow{\tau} P$. However

$$\overline{a}y|(x)ax.[x{\neq}y]\tau.P \not\approx_h \overline{a}y|((x)ax.[x{\neq}y]\tau.P+(x)ax.P)$$

for $\overline{a}y|((x)ax.[x{\neq}y]\tau.P+(x)ax.P) \xrightarrow{\tau} \mathbf{0}|P[y/x]$ can not be matched up by any transitions from $\overline{a}y|(x)ax.[x{\neq}y]\tau.P$. For similar reason

$$ax.[x{\neq}y]\tau.P \approx_h ax.[x{\neq}y]\tau.P+[x{\neq}y]ax.P$$

but

$$\overline{a}y|ax.[x{\neq}y]\tau.P \not\approx_h \overline{a}y|(ax.[x{\neq}y]\tau.P+[x{\neq}y]ax.P)$$

So the theory of weak equivalences of fusion calculus need be overhauled.

In the above counter examples the mismatch operator plays a crucial part. From a programming point of view the role of the mismatch combinator is to

terminate a process at run time. This is a useful function in practice and yet realizable in neither CCS nor calculi of mobile processes without the mismatch combinator. The problem caused by the mismatch combinator is mostly operational. A well known fact is that transitions are not stable under name instantiations, which render the algebraic theory difficult. The mismatch operator often creates a 'now-or-never' situation in which if an action does not happen right now it might never be allowed to happen. In the calculi with the mismatch operator processes are more sensible to the timing of actions. This reminds one of the difference between early and late semantics.

The early/late dichotomy is well known in the semantic theory of π-calculus. The weak late congruence is strictly contained in the weak early congruence in π-calculus whether the mismatch combinator is present or not. For some time it was taken for granted that there is no early and late distinction in weak open congruence. At least this is true for the calculus without the mismatch combinator. Very recently the present authors discovered to their surprise that early and late approaches give rise to two different weak open congruences in the π-calculus in the presence of the mismatch combinator. This has led them to realize the problem with the weak hyperequivalence.

In this paper we study early and late open congruences for χ-calculus with the mismatch operator. The main contributions of this paper are as follows:

- We point out that there is an early/late discrepancy in the open semantics of calculi of mobile processes with the mismatch combinator.
- We provide a correct treatment of open semantics for χ-calculus with mismatch. Our definitions of early and late congruences suggest immediately how to generalize them to fusion calculus.
- We propose two new tau laws to handle free prefix operator and one to deal with update prefix combinator. These tau laws rectify the mistaken tau laws in [8]. We point out that the new tau laws for free prefix operator subsume the corresponding laws for bound prefix operator.
- We give complete axiomatic systems for both early open congruence and late open congruence. These are the first completeness result for weak congruences on χ-like processes with the mismatch operator.

The structure of the paper is as follows: Section 2 summarizes some background material on χ-calculus. Section 3 defines two weak open congruences. Section 4 proves completeness theorem. Some comments are made in the final section.

2 The χ-Calculus with Mismatch

The π-calculus has been shown to be a powerful language for concurrent computation. From the algebraic point of view, the model is slightly inconvenient due to the presence of two classes of restricted names. The input prefix operator $a(x)$ introduces the dummy name x to be instantiated by an action induced by the prefix operator. On the other hand the localization operator (y) forces the

name y to be local, which will never be instantiated. Semantically these two restricted names are very different. The χ-calculus can be seen as obtained from the π-calculus by unifying the two classes of restricted names. The calculus studied in this paper is the χ-calculus extended with the mismatch operator. This language will be referred to as the $\chi^{\neq}$-calculus in the rest of the paper.

We will write $\mathcal{C}$ for the set of $\chi^{\neq}$-processes defined by the following grammar:

$$P := \mathbf{0} \mid \alpha[x].P \mid P|P \mid (x)P \mid [x{=}y]P \mid [x{\neq}y]P \mid P{+}P$$

where $\alpha \in \mathcal{N} \cup \overline{\mathcal{N}}$. Here $\mathcal{N}$ is the set of names ranged over by small case letters. The set $\{\overline{x} \mid x \in \mathcal{N}\}$ of conames is denoted by $\overline{\mathcal{N}}$. We have left out replication processes since we will be focusing on axiomatization of equivalences on finite processes. The name x in $(x)P$ is local. A name is global in P if it is not local in P. The global names, the local names and the names of P, as well as the notations $gn(P)$, $ln(P)$ and $n(P)$, are used in their standard meanings. In sequel we will use the functions $gn(\_)$, $ln(\_)$ and $n(\_)$ without explanation. We adopt the α-convention widely used in the literature on process algebra.

The following labeled transition system defines the operational semantics:

Sequentialization

$$\frac{}{\alpha[x].P \xrightarrow{\alpha[x]} P}Sqn$$

Composition

$$\frac{P \xrightarrow{\nu} P' \quad bn(\nu) \cap gn(Q){=}\emptyset}{P|Q \xrightarrow{\nu} P'|Q}Cmp_0 \qquad \frac{P \xrightarrow{[y/x]} P'}{P|Q \xrightarrow{[y/x]} P'|Q[y/x]}Cmp_1$$

Communication

$$\frac{P \xrightarrow{\alpha(x)} P' \quad Q \xrightarrow{\overline{\alpha}[y]} Q'}{P|Q \xrightarrow{\tau} P'[y/x]|Q'}Cmm_0 \qquad \frac{P \xrightarrow{\alpha(x)} P' \quad Q \xrightarrow{\overline{\alpha}(x)} Q'}{P|Q \xrightarrow{\tau} (x)(P'|Q')}Cmm_1$$

$$\frac{P \xrightarrow{\alpha[x]} P' \quad Q \xrightarrow{\overline{\alpha}[y]} Q' \quad x \neq y}{P|Q \xrightarrow{[y/x]} P'[y/x]|Q'[y/x]}Cmm_2 \qquad \frac{P \xrightarrow{\alpha[x]} P' \quad Q \xrightarrow{\overline{\alpha}[x]} Q'}{P|Q \xrightarrow{\tau} P'|Q'}Cmm_3$$

Localization

$$\frac{P \xrightarrow{\lambda} P' \quad x \notin n(\lambda)}{(x)P \xrightarrow{\lambda} (x)P'}Loc_0 \qquad \frac{P \xrightarrow{\alpha[x]} P' \quad x \notin \{\alpha, \overline{\alpha}\}}{(x)P \xrightarrow{\alpha(x)} P'}Loc_1 \qquad \frac{P \xrightarrow{[y/x]} P'}{(x)P \xrightarrow{\tau} P'}Loc_2$$

Condition

$$\frac{P \xrightarrow{\lambda} P'}{[x{=}x]P \xrightarrow{\lambda} P'}Mtch \qquad \frac{P \xrightarrow{\lambda} P' \quad x{\neq}y}{[x{\neq}y]P \xrightarrow{\lambda} P'}Mismtch$$

Summation

$$\frac{P \xrightarrow{\lambda} P'}{P{+}Q \xrightarrow{\lambda} P'}Sum$$

We have omitted all the symmetric rules. In the above rules the letter ν ranges over the set $\{\alpha(x), \alpha[x] \mid \alpha \in \mathcal{N} \cup \overline{\mathcal{N}}, x \in \mathcal{N}\} \cup \{\tau\}$ of actions and the letter λ over the set $\{\alpha(x), \alpha[x], [y/x] \mid \alpha \in \mathcal{N} \cup \overline{\mathcal{N}}, x, y \in \mathcal{N}\} \cup \{\tau\}$ of labels. The symbols $\alpha(x), \alpha[x], [y/x]$ represent restricted action, free action and update action respectively. The x in $\alpha(x)$ is local.

A substitution is a function from $\mathcal{N}$ to $\mathcal{N}$. Substitutions are usually denoted by σ, σ' etc.. The empty substitution, that is the identity function on $\mathcal{N}$, is written as $[]$. The result of applying σ to P is denoted by $P\sigma$. Suppose Y is a finite set $\{y_1, \ldots, y_n\}$ of names. The notation $[y \notin Y]P$ stands for $[y \neq y_1] \ldots [y \neq y_n]P$, where the order of the mismatch operators is immaterial. We will write ϕ and ψ to stand for sequences of match and mismatch combinators concatenated one after another, μ for a sequence of match operators, and δ for a sequence of mismatch operators. Consequently we write ψP, μP and δP. When the length of ψ (μ, δ) is zero, ψP (μP, δP) is just P. The notation $\phi \Rightarrow \psi$ says that ϕ logically implies ψ and $\phi \Leftrightarrow \psi$ that ϕ and ψ are logically equivalent. A substitution σ agrees with ψ, and ψ agrees with σ, when $\psi \Rightarrow x{=}y$ if and only if $\sigma(x){=}\sigma(y)$. The notations $\Longrightarrow$ and $\stackrel{\hat{\lambda}}{\Longrightarrow}$ are used in their standard meanings. A sequence $x_1, \ldots, x_n$ of names will be abbreviated to $\tilde{x}$.

The following lemma is useful in later proofs.

Lemma 1. *If $P\sigma \stackrel{\lambda}{\longrightarrow} P_1$ then there exists some P' such that $P_1 \equiv P'\sigma$.*

Notice that a substitution σ may disable an action of P. So one can not conclude $P\sigma \stackrel{\lambda\sigma}{\longrightarrow} P'\sigma$ from $P \stackrel{\lambda}{\longrightarrow} P'$. But the above lemma tells us that we may write $P\sigma \stackrel{\lambda}{\longrightarrow} P'\sigma$ once we know that $P\sigma$ can induce a λ transition.

3 Open Bisimilarities

In our view the most natural equivalence for mobile processes is the open equivalence introduced by Sangiorgi ([9]). Technically the open equivalence is a bisimulation equivalence closed under substitution of names. Philosophically the open approach assumes that the environments are dynamic in the sense that they are shrinking, expanding and changing all the time. After the simulation of each computation step, the environment might be totally different. As a matter of fact the very idea of bisimulation is to ensure that no operational difference can be detected by any dynamic environment. So closure under substitution is a reasonable requirement.

A simple minded definition of weak open bisimulation would go as follows:

> A binary relation $\mathcal{R}$ on $\mathcal{C}$ is a weak open bisimulation if it is symmetric and closed under substitution such that whenever $P\mathcal{R}Q$ and $P \stackrel{\lambda}{\longrightarrow} P'$ then $Q \stackrel{\widehat{\lambda}}{\Longrightarrow} Q'\mathcal{R}P'$ for some Q'.

As it turns out this is a bad definition for processes with the mismatch operator. Counter examples are given in the introduction. The problem here is that the

instantiation of names is delayed for any period of time. This is not always possible in $\chi^{\neq}$-calculus. To correct the above definition, one should adopt the approach that name instantiations should take place in the earliest possible occasion. This brings us to the familiar early and late frameworks.

Definition 2. *Let $\mathcal{R}$ be a binary symmetric relation on $\mathcal{C}$. It is called an early open bisimulation if it is closed under substitution and whenever $P\mathcal{R}Q$ then the following properties hold:*
(i) If $P \xrightarrow{\tau} P'$ then Q' exists such that $Q \Longrightarrow Q'\mathcal{R}P'$.
(ii) If $P \xrightarrow{[y/x]} P'$ then Q' exists such that $Q \overset{[y/x]}{\Longrightarrow} Q'\mathcal{R}P'$.
(iii) If $P \xrightarrow{\alpha[x]} P'$ then for every y some Q', Q'' exist such that $Q \Longrightarrow \xrightarrow{\alpha[x]} Q''$ and $Q''[y/x] \Longrightarrow Q'\mathcal{R}P'[y/x]$.
(iv) If $P \xrightarrow{\alpha(x)} P'$ then for every y some Q', Q'' exist such that $Q \Longrightarrow \xrightarrow{\alpha(x)} Q''$ and $Q''[y/x] \Longrightarrow Q'\mathcal{R}P'[y/x]$.
The early open bisimilarity $\approx_o^e$ is the largest early open bisimulation.

The clause (iv) is easy to understand. Its counterpart for weak bisimilarity of π-calculus is familiar. The clause (iii) calls for some explanation. In $\chi^{\neq}$-calculus free actions can also incur name updates in suitable contexts. Suppose $P \xrightarrow{\alpha[x]} P''$. Then $(x)(P|\overline{\alpha}[y].Q) \xrightarrow{\tau} P''[y/x]|Q[y/x]$. Even if $P'' \Longrightarrow P'$, one does not necessarily have $P''[y/x] \Longrightarrow P'[y/x]$. Had we replace clause (iii) by

(iii') If $P \xrightarrow{\alpha[x]} P'$ then some Q' exists such that $Q \overset{\alpha[x]}{\Longrightarrow} Q'\mathcal{R}P'$

then we would have obtained a relation to which the second counter example in the introduction applies. The similarity of clause (iii) and clause (iv) exhibits once again the uniformity of the names in χ-like calculi.

Analogously we can introduce late open bisimilarity.

Definition 3. *Let $\mathcal{R}$ be a binary symmetric relation on $\mathcal{C}$. It is called a late open bisimulation if it is closed under substitution and whenever $P\mathcal{R}Q$ then the following properties hold:*
(i) If $P \xrightarrow{\tau} P'$ then Q' exists such that $Q \Longrightarrow Q'\mathcal{R}P'$.
(i) If $P \xrightarrow{[y/x]} P'$ then Q' exists such that $Q \overset{[y/x]}{\Longrightarrow} Q'\mathcal{R}P'$.
(iii) If $P \xrightarrow{\alpha[x]} P'$ then Q'' exists such that $Q \Longrightarrow \xrightarrow{\alpha[x]} Q''$ and for every y some Q' exists such that $Q''[y/x] \Longrightarrow Q'\mathcal{R}P'[y/x]$.
(iv) If $P \xrightarrow{\alpha(x)} P'$ then Q'' exists such that $Q \Longrightarrow \xrightarrow{\alpha(x)} Q''$ and for every y some Q' exists such that $Q''[y/x] \Longrightarrow Q'\mathcal{R}P'[y/x]$.
The late open bisimilarity $\approx_o^l$ is the largest late open bisimulation.

It is clear that $\approx_o^l \subseteq \approx_o^e$. The following example shows that inclusion is strict: $a[x].[x{=}y]\tau.P + a[x].[x{\neq}y]\tau.P \approx_o^e a[x].[x{=}y]\tau.P + a[x].[x{\neq}y]\tau.P + a[x].P$ but not $a[x].[x{=}y]\tau.P + a[x].[x{\neq}y]\tau.P \approx_o^l a[x].[x{=}y]\tau.P + a[x].[x{\neq}y]\tau.P + a[x].P$.

The lesson we have learned is that we should always check if an observational equivalence is closed under parallel composition. The next lemma makes sure that this is indeed true for the two open bisimilarities.

Lemma 4. *Both $\approx_o^e$ and $\approx_o^l$ are closed under localization and composition.*

Both open bisimilarities are also closed under the prefix and match combinators. But neither is closed under the summation operator or the mismatch operator. For instance $[x{\neq}y]P \approx_o^e [x{\neq}y]\tau.P$ does not hold in general. To obtain the largest congruence contained in early (late) open bisimilarity we follow the standard approach: We say that two processes P and Q are early open congruent, notation $P \simeq_o^e Q$, if $P \approx_o^e Q$ and for each substitution σ a tau action of $P\sigma$ must be matched up by a non-empty sequence of tau actions from $Q\sigma$ and vice versa. Clearly $\simeq_o^e$ is a congruence. Similarly we can define $\simeq_o^l$.

4 Axiomatic System

In [2] completeness theorems are proved for L-bisimilarities on χ-processes without mismatch operator. The proofs of these completeness results use essentially the inductive definitions of L-bisimilarities. In the presence of the mismatch operator, the method used in [2] should be modified. The modification is done by incorporating ideas from [6]. In this section, we give the complete axiomatic systems for early and late open congruences using the modified approach. First we need to define two induced prefix operators, tau and update prefixes, as follows:

$$[y|x].P \stackrel{\text{def}}{=} (a)(\overline{a}[y]|a[x].P)$$
$$\tau.P \stackrel{\text{def}}{=} (b)[b|b].P$$

where a, b are fresh. The following are some further auxiliary definitions.

Definition 5. *Let V be a finite set of names. We say that ψ is complete on V if $n(\psi) \subseteq V$ and for each pair x, y of names in V it holds that either $\psi \Rightarrow x{=}y$ or $\psi \Rightarrow x \neq y$.*

Suppose ψ is complete on V and $n(\phi) \subseteq V$. Then it should be clear that either $\psi\phi \Leftrightarrow \psi$ or $\psi\phi \Leftrightarrow \bot$. In sequel this fact will be used implicitly.

Lemma 6. *If ϕ and ψ are complete on V and both agree with σ then $\phi \Leftrightarrow \psi$.*

Definition 7. *A substitution σ is induced by ψ if it agrees with ψ and $\sigma\sigma = \sigma$.*

Let AS denote the system consisting of the rules and laws in Appendix A plus the following expansion law:

$$P|Q = \sum_i \phi_i(\tilde{x})\pi_i.(P_i|Q) + \sum_{\gamma_j=\overline{b_j}[y_j]}^{\pi_i=a_i[x_i]} \phi_i\psi_j(\tilde{x})(\tilde{y})[a_i{=}b_j][x_i|y_j].(P_i|Q_j) +$$
$$\sum_j \psi_j(\tilde{y})\gamma_j.(P|Q_j) + \sum_{\gamma_j=b_j[y_j]}^{\pi_i=\overline{a_i}[x_i]} \phi_i\psi_j(\tilde{x})(\tilde{y})[a_i{=}b_j][x_i|y_j].(P_i|Q_j)$$

where P is $\sum_i \phi_i(\tilde{x})\pi_i.P_i$ and Q is $\sum_j \psi_j(\tilde{y})\gamma_j.Q_j$, π_i and γ_j range over $\{\alpha[x] \mid \alpha \in \mathcal{N} \cup \overline{\mathcal{N}}, x \in \mathcal{N}\}$. In the expansion law, the summand

$$\sum_{\gamma_j=\overline{b_j}[y_j]}^{\pi_i=a_i[x_i]} \phi_i\psi_j(\tilde{x})(\tilde{y})[a_i{=}b_j][x_i|y_j].(P_i|Q_j)$$

contains $\phi_i\psi_j(\tilde{x})(\tilde{y})[a_i{=}b_j][x_i|y_j].(P_i|Q_j)$ as a summand whenever $\pi_i = a_i[x_i]$ and $\gamma_j = \overline{b_j}[y_j]$.

We write $AS \vdash P = Q$ to indicate that the equality $P = Q$ can be inferred from AS. Some important derived laws of AS are given in Appendix A.

Using axioms in AS, a process can be converted to a process that contains no occurrence of the composition operator, the latter process is of special form as defined below.

Definition 8. *A process P is in normal form on $V \supseteq fn(P)$ if P is of the form $\sum_{i\in I_1} \phi_i\alpha_i[x_i].P_i + \sum_{i\in I_2} \phi_i(x)\alpha_i[x].P_i + \sum_{i\in I_3} \phi_i[z_i|y_i].P_i$ such that x does not appear in P, ϕ_i is complete on V for each $i \in I_1 \cup I_2 \cup I_3$, P_i is in normal form on V for $i \in I_1 \cup I_3$ and is in normal form on $V \cup \{x\}$ for $i \in I_2$. Here I_1, I_2 and I_3 are pairwise disjoint finite indexing sets.*

Notice that if P is in normal form and σ is a substitution then $P\sigma$ is in normal form.

The depth of a process measures the maximal length of nested prefixes in the process. The structural definition goes as follows: (i) $d(\mathbf{0}) = 0$; (ii) $d(\alpha[x].P) = 1{+}d(P)$; (iii) $d(P|Q) = d(P){+}d(Q)$; (iv) $d((x)P) = d(P)$; (v) $d([x{=}y]P) = d(P)$; (vi) $d(P{+}Q) = max\{d(P), d(Q)\}$.

Lemma 9. *For a process P and a finite set V of names such that $fn(P) \subseteq V$ there is a normal form Q on V such that $d(Q) \leq d(P)$ and $AS \vdash Q = P$.*

It can be shown that AS is complete for the strong open bisimilarity on $\chi^{\neq}$-processes. This fact will not be proved here. Our attention will be confined to the completeness of the two weak open congruences. The tau laws used in this paper are given in Figure 1. Some derived tau laws are listed in Figure 2. In what follows, we will write AS_o^l for $AS \cup \{T1, T2, T3a, T3b, T3d, T4\}$ and AS_o^e for $AS \cup \{T1, T2, T3a, T3c, T3d, T4\}$.

The next lemma discusses some relationship among the tau laws.

Lemma 10. *(i) $AS_o^e \vdash TD5$. (ii) $AS_o^l \vdash TD6$. (iii) $AS \cup \{T3c\} \vdash T3b$.*

Proof. (i) By T3c and C2, we get:

$$\begin{aligned} AS_o^e \vdash \Sigma(a(x), P, Q, \delta) &= (x)(\Sigma(a[x], P, Q, \delta){+}[x{\notin}n(\delta)]\delta a[x].Q) \\ &= \Sigma(a(x), P, Q, \delta){+}(x)[x{\notin}n(\delta)]\delta a[x].Q \\ &\overset{LD3}{=} \Sigma(a(x), P, Q, \delta){+}(x)\delta a[x].Q \\ &\overset{LD2}{=} \Sigma(a(x), P, Q, \delta){+}\delta(x)a[x].Q \end{aligned}$$

The proofs of (ii) and (iii) are omitted. □

T1	$\alpha[x].\tau.P = \alpha[x].P$	
T2	$P+\tau.P = \tau.P$	
T3a	$\alpha[x].(P+\tau.Q) = \alpha[x].(P+\tau.Q)+\alpha[x].Q$	
T3b	$\alpha[x].(P+\delta\tau.Q) = \alpha[x].(P+\delta\tau.Q)+[x\notin n(\delta)]\delta\alpha[x].Q$	$x \notin n(\delta)$
T3c	$\Sigma(\alpha[x],P,Q,\delta) = \Sigma(\alpha[x],P,Q,\delta)+[x\notin n(\delta)]\delta\alpha[x].Q$	$x \notin n(\delta)$
T3d	$[y\|x].(P+\delta\tau.Q) = [y\|x].(P+\delta\tau.Q)+\psi\delta[y\|x].Q$	
T4	$\tau.P = \tau.(P+\psi\tau.P)$	
In T3d, if $\delta \Rightarrow [u\neq v]$ then either $\psi \Rightarrow [x=u][y\neq v]$ or $\psi \Rightarrow [x=v][y\neq u]$ or $\psi \Rightarrow [y=u][x\neq v]$ or $\psi \Rightarrow [y=v][x\neq u]$ or $\psi \Rightarrow [x\neq u][x\neq v][y\neq u][y\neq v]$. In T3c, $\Sigma(\alpha[x],P,Q,\delta)$ is $\sum_{y\in Y}\alpha[x].(P_y+\delta[x=y]\tau.Q)+\alpha[x].(P+\delta[x\notin Y]\tau.Q)$.		

Fig. 1. The Tau Laws

TD1	$[x\|y].\tau.P = [x\|y].P$	
TD2	$\tau.\tau.P = \tau.P$	
TD3	$[x\|y].(P+\tau.Q) = [x\|y].(P+\tau.Q)+[x\|y].Q$	
TD4	$\tau.(P+\tau.Q) = \tau.(P+\tau.Q)+\tau.Q$	
TD5	$(x)\alpha[x].(P+\delta\tau.Q) = (x)\alpha[x].(P+\delta\tau.Q)+\delta(x)\alpha[x].Q$	$x \notin n(\delta)$
TD6	$\Sigma(\alpha(x),P,Q,\delta) = \Sigma(\alpha(x),P,Q,\delta)+\delta(x)\alpha[x].Q$	$x \notin n(\delta)$
In TD6, $\Sigma(\alpha(x),P,Q,\delta)$ is $\sum_{y\in Y}(x)\alpha[x].(P_y+\delta[x=y]\tau.Q)+(x)\alpha[x].(P+\delta[x\notin Y]\tau.Q)$.		

Fig. 2. The Derived Tau Laws

To establish the completeness theorem, some properties of AS and the open bisimilarities must be established first. The next three lemmas describe these properties.

Lemma 11. *Suppose Q is in normal form on V, ϕ is complete on V, and σ is a substitution induced by ϕ. Then the following properties hold:*
(i) If $Q\sigma \stackrel{\tau}{\Longrightarrow} Q'$ then $AS \cup \{T1,T2,T3a\} \vdash Q = Q+\phi\tau.Q'$.
(ii) If $Q\sigma \Longrightarrow \stackrel{\alpha[x]}{\longrightarrow} Q'$ then $AS \cup \{T1,T2,T3a\} \vdash Q = Q+\phi\alpha[x].Q'$.
(iii) If $Q\sigma \Longrightarrow \stackrel{\alpha(x)}{\longrightarrow} Q'$ then $AS \cup \{T1,T2,T3a\} \vdash Q = Q+\phi(x)\alpha[x].Q'$.
(iv) If $Q\sigma \stackrel{[y/x]}{\Longrightarrow} Q'$ then $AS \cup \{T1,T2,T3a,T3d\} \vdash Q = Q+\phi[y|x].Q'$.

Proof. (iv) If $Q\sigma \stackrel{[y/x]}{\longrightarrow} Q'$ then $AS \cup \{T1,T2,T3a,T3d\} \vdash Q = Q+\phi[y|x].Q'$. Suppose for example $Q\sigma \stackrel{\tau}{\Longrightarrow} Q_1\sigma \stackrel{[y/x]}{\longrightarrow} Q_2 \stackrel{\tau}{\Longrightarrow} Q'$. It is easy to see that $Q_2 \equiv Q'_2\sigma[y/x]$. Let ψ be a complete condition on $fn(Q'_2)$ that induces $\sigma[y/x]$. Suppose $\psi \Leftrightarrow \mu\delta$. Clearly $\mu\sigma[y/x]$ is true. Therefore

$$\begin{aligned} AS \cup \{T1,T2,T3a,T3d\} \vdash Q &= Q+\phi[y|x].Q_2 \\ &= Q+\phi[y|x].Q'_2\sigma[y/x] \\ &= Q+\phi[y|x].Q'_2 \\ &= Q+\phi[y|x].(Q'_2+\psi\tau.Q') \end{aligned}$$

$$
\begin{aligned}
&= Q+\phi[y|x].(Q'_2+\mu\delta\tau.Q') \\
&= Q+\phi[y|x].(Q'_2+\delta\tau.Q') \\
&= Q+\phi([y|x].(Q'_2+\delta\tau.Q')+\theta\delta\tau.Q') \\
&= Q+\phi\theta\delta[y|x].Q'
\end{aligned}
$$

in which θ is defined as in $T3d$. First of all it is easy to see that $\phi \Rightarrow \delta$ for ϕ is complete on V. So $\phi\delta \Leftrightarrow \phi$. Second we need to explain how to construct θ. It should be constructed in such a way that $\phi\theta \Leftrightarrow \phi$. Suppose $\delta \Rightarrow [u{\neq}v]$. There are five possibilities:

- $\phi \Rightarrow x{=}u$. Then we let θ contain $[x{=}u][y{\neq}v]$. If $\phi \Rightarrow y{=}v$ then σ is induced by $[x{=}u][y{=}v]$. It follows that $\sigma[y/x]$ is induced by $[u{=}v]$, which is impossible. Hence $\phi \Rightarrow y{\neq}v$.
- $\phi \Rightarrow y{=}u$ or $\phi \Rightarrow x{=}v$ or $\phi \Rightarrow y{=}v$. These cases are similar to previous case.
- $\phi \Rightarrow [x{\neq}u][x{\neq}v][y{\neq}u][y{\neq}v]$. Simply let θ contain $[x{\neq}u][x{\neq}v][y{\neq}u][y{\neq}v]$.

It is clear from the construction that $\phi \Rightarrow \theta$. Therefore $AS \cup \{T1, T2, T3a, T3d\} \vdash Q = Q+\phi\theta\delta[y|x].Q' = Q+\phi[y|x].Q'$. □

Lemma 12. *Suppose Q is a normal form on some $V = \{y_1, \ldots, y_k\} \supseteq fn(Q)$, ψ is complete on V, and σ is a substitution induced by ψ. If*

$$
\begin{aligned}
&Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'_1\sigma,\ Q'_1\sigma[y_1/x] \Longrightarrow Q_1, \\
&Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'_2\sigma,\ Q'_2\sigma[y_2/x] \Longrightarrow Q_2, \\
&\vdots \\
&Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'_k\sigma,\ Q'_k\sigma[y_k/x] \Longrightarrow Q_k, \\
&Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'_{k+1}\sigma \Longrightarrow Q_{k+1}
\end{aligned}
$$

then the following properties hold:

1. $AS \cup \{T1, T2, T3a\} \vdash Q = Q + \psi\sum_{j=1}^{k}(x)\alpha[x].(\tau.Q'_j + \psi[x{=}y_j]\tau.Q_j) + \psi(x)\alpha[x].(\tau.Q'_{k+1} + \psi[x{\notin}V]\tau.Q_{k+1}))$.
2. *If* $Q'_1 \equiv Q', \ldots, Q'_{k+1} \equiv Q'$, *then* $Q + \psi(x)\alpha[x].(\tau.Q' + \psi\sum_{j=1}^{k}[x{=}y_j]\tau.Q_j + \phi[x{\notin}V]\tau.Q_{k+1})$ *is provably equal to* Q *in* $AS \cup \{T1, T2, T3a\}$.

The proof of the above lemma is similar to that of Lemma 11.

Lemma 13. *In $\chi^{\neq}$-calculus the following properties hold:*
(i) If $P \approx^e_o Q$ then $AS^e_o \vdash \tau.P = \tau.Q$.
(ii) If $P \approx^l_o Q$ then $AS^l_o \vdash \tau.P = \tau.Q$.

Proof. By Lemma 9 we may assume that P and Q are in normal form on $V = fn(P|Q) = \{y_1, y_2, \ldots, y_k\}$. Let P be

$$\sum_{i\in I_1}\phi_i\alpha_i[x_i].P_i + \sum_{i\in I_2}\phi_i(x)\alpha_i[x].P_i + \sum_{i\in I_3}\phi_i[z_i|y_i].P_i$$

and Q be

$$\sum_{j\in J_1}\psi_j\alpha_j[x_j].Q_j+\sum_{j\in J_2}\psi_j(x)\alpha_j[x].Q_j+\sum_{j\in J_3}\psi_j[z_j|y_j].Q_j$$

We prove this lemma by induction on the depth of $P|Q$.

(i) Suppose $\phi_i\pi_i.P_i$ is a summand of P and σ is induced by ϕ_i. There are several cases:

- $\pi_i\sigma$ is an update action $[y|x]$. It follows from $P\approx_o^e Q$ that $Q\sigma\stackrel{[y/x]}{\Longrightarrow}Q'\approx_o^e P_i[y/x]\sigma$. By induction we have $AS_o^e\vdash Q'=P_i\sigma[y/x]$. By (iv) of Lemma 11

$$\begin{aligned}AS_o^e\vdash Q&=Q+\phi_i[y|x].Q'\\&=Q+\phi_i[y|x].P_i[y/x]\sigma\\&=Q+\phi_i[y|x].P_i\sigma\\&=Q+\phi_i\pi_i\sigma.P_i\sigma\\&=Q+\phi_i\pi_i.P_i\end{aligned}$$

- $\pi_i\sigma$ is a restricted action $\alpha(x)$. Since $P\approx_o^e Q$ one has the following cases:
  - For each $l\in\{1,\dots,k\}$, Q'_{i_l} and Q_{i_l} exist such that $Q\sigma\Longrightarrow\stackrel{\alpha(x)}{\longrightarrow}Q'_{i_l}\sigma$ and $Q'_{i_l}\sigma[y_l/x]\Longrightarrow Q_{i_l}\approx_o^e P_i\sigma[y_l/x]$.
  - $Q'_{i_{k+1}}$ and $Q_{i_{k+1}}$ exist such that $Q\sigma\Longrightarrow\stackrel{\alpha(x)}{\longrightarrow}Q'_{i_{k+1}}\sigma\Longrightarrow Q_{i_{k+1}}\approx_o^e P_i\sigma$.

  By Lemma 12

$$\begin{aligned}AS_o^e\vdash Q&=Q+\sum_{l=1}^{k}\phi_i(x)a[x].(\tau.Q'_{i_l}+\phi_i[x{=}y_l]\tau.Q_{i_l})\\&\quad+\phi_i(x)a[x].(\tau.Q'_{i_{k+1}}+\phi_i[x{\notin}V]\tau.Q_{i_{k+1}})\\&=Q+\sum_{l=1}^{k}\phi_i(x)a[x].(\tau.Q'_{i_l}+\phi_i[x{=}y_l]\tau.P_i\sigma[y_l/x])\\&\quad+\phi_i(x)a[x].(\tau.Q'_{i_{k+1}}+\phi_i[x{\notin}V]\tau.P_i\sigma)\\&=Q+\phi(x)a[x].P_i\\&=Q+\phi_i\pi_i.P_i\end{aligned}$$

- $\pi_i\sigma$ is a free action $\alpha[x]$. Using the fact $P\approx_o^e Q$ one has the following cases:
  - For each $l\in\{1,\dots,k\}$, Q'_{i_l} and Q_{i_l} exist such that $Q\sigma\Longrightarrow\stackrel{\alpha[x]}{\longrightarrow}Q'_{i_l}\sigma$ and $Q'_{i_l}\sigma[y_l/x]\Longrightarrow Q_{i_l}\approx_o^e P_i\sigma[y_l/x]$.
  - $Q'_{i_{k+1}}$ and $Q_{i_{k+1}}$ exist such that $Q\sigma\Longrightarrow\stackrel{\alpha[x]}{\longrightarrow}Q'_{i_{k+1}}\sigma\Longrightarrow Q_{i_{k+1}}\approx_o^e P_i\sigma$.

  Since ϕ_i is complete on V, it groups the elements of V into several disjoint classes. Assume that these classes are $[x],[a_1],\dots,[a_r]$. Let $\phi_i^{=}$ be the sequence of match operators induced by the equivalence classes $[a_1],\dots,[a_r]$. Let $\phi_i^{=x}$ be the sequence of match operators induced by the equivalence class $[x]$. Let $\phi_i^{\neq}$ be the sequence of mismatch combinators constructed as

follows: For $1 \leq p, q \leq r$ and $a \in [a_p], b \in [a_q]$, $a{\neq}b$ is in $\phi_i^{\neq}$. And let $\phi_i^{\neq x}$ be the sequence of mismatch combinators constructed as follows: For $a \in [a_1] \cup \ldots \cup [a_r]$, $a{\neq}x$ is in $\phi_i^{\neq x}$. It is clear that $\phi_i \Leftrightarrow \phi_i^{=}\phi_i^{=x}\phi_i^{\neq}\phi_i^{\neq x}$. Now V can be divided into two subsets: $V^{=x} \stackrel{\text{def}}{=} \{y \mid y \in V, \phi_i \Rightarrow y{=}x\}$; and $V^{\neq x} \stackrel{\text{def}}{=} \{y \mid y \in V, \phi_i \Rightarrow y{\neq}x\} = [a_1]\cup\ldots\cup[a_r]$. Clearly $\phi_i^{\neq x} \Leftrightarrow [x{\notin}V^{\neq x}]$.

- If $y_l \in V^{\neq x}$ then we define $\phi_{i\backslash[y_l]}^{\neq x}$ as follows: For $a \in ([a_1] \cup \ldots \cup [a_r]) \setminus [y_l]$, $a{\neq}x$ is in $\phi_{i\backslash[y_l]}^{\neq x}$. It is easy to see that $\phi_{i\backslash[y_l]}^{\neq x} \Leftrightarrow [x{\notin}(V^{\neq x} \setminus [y_l])]$. Now $\phi_i^{=}\phi_i^{=x}\phi_i^{\neq}\phi_{i\backslash[y_l]}^{\neq x}[x{=}y_l]$ is complete on V and induces $\sigma[y_l/x]$. By Lemma 11

$$\begin{aligned}
Q &= Q + \phi_i\alpha[x].Q'_{i_l}\\
&= Q + \phi_i\alpha[x].\tau.Q'_{i_l}\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{=}\phi_i^{=x}\phi_i^{\neq}\phi_{i\backslash[y_l]}^{\neq x}[x{=}y_l]\tau.Q_{i_l})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}\phi_{i\backslash[y_l]}^{\neq x}[x{=}y_l]\tau.Q_{i_l})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}[x{\notin}(V^{\neq x} \setminus [y_l])][x{=}y_l]\tau.Q_{i_l})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}[y_l{\notin}(V^{\neq x} \setminus [y_l])][x{=}y_l]\tau.Q_{i_l})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}[x{=}y_l]\tau.Q_{i_l})
\end{aligned}$$

- It is clear that $\phi_i^{=}\phi_i^{=x}\phi_i^{\neq}\phi_i^{\neq x}$ is complete on V and induces σ. One has by Lemma 11 that

$$\begin{aligned}
Q &= Q + \phi_i\alpha[x].Q'_{i_k}\\
&= Q + \phi_i\alpha[x].\tau.Q'_{i_k}\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_k} + \phi_i^{=}\phi_i^{=x}\phi_i^{\neq}\phi_i^{\neq x}\tau.Q_{i_{k+1}})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_k} + \phi_i^{\neq}\phi_i^{\neq x}\tau.Q_{i_{k+1}})\\
&= Q + \phi_i\alpha[x].(\tau.Q'_{i_k} + \phi_i^{\neq}[x{\notin}V^{\neq x}]\tau.Q_{i_{k+1}})
\end{aligned}$$

Now

$$\begin{aligned}
AS_o^e \vdash Q &= Q + \sum_{y_l \in V^{\neq x}} \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}[x{=}y_l]\tau.Q_{i_l})\\
&\quad + \phi_i\alpha[x].(\tau.Q'_{i_{k+1}} + \phi_i^{\neq}[x{\notin}V^{\neq x}]\tau.Q_{i_{k+1}})\\
&= Q + \sum_{y_l \in V^{\neq x}} \phi_i\alpha[x].(\tau.Q'_{i_l} + \phi_i^{\neq}[x{=}y_l]\tau.P_i\sigma[y_l/x])\\
&\quad + \phi_i\alpha[x].(\tau.Q'_{i_{k+1}} + \phi_i^{\neq}[x{\notin}V^{\neq x}]\tau.P_i\sigma)\\
&= Q + \phi_i[x{\notin}n(\phi_i^{\neq})]\alpha[x].P_i\\
&= Q + \phi_i\alpha[x].P_i
\end{aligned}$$

– $\pi_i\sigma$ is a tau action. If the tau action is matched by $Q\sigma \stackrel{\tau}{\Longrightarrow} Q'$ then it is easy to prove that $AS_o^e \vdash Q = Q + \phi_i\pi_i.P_i$. If the tau action is matched vacuously then $AS_o^e \vdash Q + \phi_i\pi_i.P_i = Q + \phi_i\tau.Q$.

In summary, we have $AS^e_o \vdash P+Q = Q+\Sigma_{i\in I'}\phi_i\tau.Q$ for some $I' \subseteq I$. So by T4 we get $AS^e_o \vdash \tau.(P+Q) = \tau.(Q+\Sigma_{i\in I'}\phi_i\tau.Q) = \tau.Q$. Symmetrically, we can prove $AS^e_o \vdash \tau.(P+Q) = \tau.P$. Hence $AS^e_o \vdash \tau.P = \tau.Q$.

(ii) The proof is similar to that for $\approx^e_o$. We consider only one case:

– $\pi_i\sigma$ is a restricted action $\alpha(x)$. It follows from $P \approx^l_o Q$ that some Q' exists such that the following holds:
  - For each $l \in \{1,\ldots,k\}$, Q', Q_{i_l} exists such that $Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'\sigma$ and $Q'\sigma[y_l/x] \Longrightarrow Q_{i_l} \approx^l_o P_i\sigma[y_l/x]$.
  - $Q_{i_{k+1}}$ exists such that $Q\sigma \Longrightarrow \xrightarrow{\alpha(x)} Q'\sigma \Longrightarrow Q_{i_{k+1}} \approx^l_o P_i\sigma$.

  By (ii) of Lemma 12 we get

$$\begin{aligned} AS^l_o \vdash Q &= Q+\phi_i(x)\alpha[x].(\tau.Q'+\phi_i\sum_{l=1}^{k}[x{=}y_l]\tau.Q_{i_l}+\phi_i[x{\notin}V]\tau.Q_{i_{k+1}}) \\ &= Q+\phi_i(x)\alpha[x].(\tau.Q'+\phi_i\sum_{l=1}^{k}[x{=}y_l]\tau.P_i\sigma[y_l/x]+\phi_i[x{\notin}V]\tau.P_i\sigma) \\ &= Q+\phi_i(x)\alpha[x].(\tau.Q'+\phi_i\tau.P_i) \\ &= Q+\phi_i(x)\alpha[x].P_i \end{aligned}$$

Then by a similar argument as in (i), we get $AS^l_o \vdash \tau.P = \tau.Q$. □

Theorem 14. *In $\chi^{\neq}$-calculus the following completeness results hold:*
(i) If $P \simeq^e_o Q$ then $AS^e_o \vdash P = Q$.
(ii) If $P \simeq^l_o Q$ then $AS^l_o \vdash P = Q$.

Proof. By Lemma 11 and Lemma 13, one can prove the theorem in very much the same way as the proof of Lemma 13 is done. □

5 Historical Remark

The first author of this paper has been working on χ-calculus for some years. His attention had always been on the version of χ without mismatch combinator. By the end of 1999 he started looking at testing congruence on χ-processes. In order to axiomatize the testing congruence he was forced to introduce the mismatch operator. This led him to deal with open congruences on $\chi^{\neq}$-processes, which made him aware of the fact that the open semantics for the π-calculus with the mismatch combinator has not been investigated before. So he, together with the second author, began to work on the problem for the π-calculus with the mismatch combinator. Their investigation showed that the simple-minded definition of open bisimilarity is not closed under parallel composition. It is then a small step to realize the problem of the weak hyperequivalence.

It has come a long way to settle down on the axiom T4. The first solution, proposed by the first author in an early version of [4], is the following rule:

$$\frac{P+\Sigma_{i\in I}\psi_i\tau.P = Q+\Sigma_{j\in J}\psi_j\tau.Q}{\tau.P = \tau.Q}$$

The premises of the rule is an equational formalization of $P \approx Q$. In the final version of [4] he observed that the rule is equivalent to the following law:

$$\tau.P = \tau.(P + \Sigma_{i \in I} \psi_i \tau.P)$$

Later on he realized that the above equality can be simplified to T4.

In [8] two tau laws are proposed for fusion calculus. Using the notations of [8] they can be written as follows:

$$\alpha.(P + \tilde{M}\mathbf{1}.Q) = \alpha.(P + \tilde{M}\mathbf{1}.Q) + \tilde{M}\alpha.Q \quad (1)$$

$$\iota.(P + \tilde{M}\rho.Q) = \iota.(P + \tilde{M}\rho.Q) + \tilde{M}\iota \wedge \rho.Q \text{ if } \forall x, y.(\tilde{M} \Rightarrow x \neq y) \Rightarrow \neg(x \iota y) \quad (2)$$

where α is a communication action, ι and ρ are fusion actions, and $\tilde{M}$ is a sequence of match/mismatch operators. Neither (1) nor (2) is valid. The counterexample to (1) is given in the introduction. The problem with (2) is that it is not closed under substitution. The following is an instance of (2) since the side condition is satisfied:

$$\{x{=}y\}.(P + [u{\neq}v]\mathbf{1}.Q) = \{x{=}y\}.(P + [u{\neq}v]\mathbf{1}.Q) + [u{\neq}v]\{x{=}y\}.Q$$

But if we substitute u for x and v for y we get

$$\{x{=}y\}.(P + [x{\neq}y]\mathbf{1}.Q) = \{x{=}y\}.(P + [x{\neq}y]\mathbf{1}.Q) + [x{\neq}y]\{x{=}y\}.Q$$

This equality should not hold. Our T3b is the correction of (1) while our T3d is a special case of the correct version of (2). In order for (2) to be valid, the side condition has to be internalized as it were.

References

1. Y. Fu. A Proof Theoretical Approach to Communications. *ICALP'97*, Lecture Notes in Computer Science 1256, Springer, 325–335, 1997. 596
2. Y. Fu. Bisimulation Lattice of Chi Processes. *ASIAN'98*, Lecture Notes in Computer Science 1538, Springer, 245–262, 1998. 596, 602
3. Y. Fu. Variations on Mobile Processes. *Theoretical Computer Science*, **221**: 327-368, 1999. 596
4. Y. Fu. Open Bisimulations of Chi Processes. *CONCUR'99*, Lecture Notes in Computer Science 1664, Springer, 304-319, 1999. 596, 597, 608, 609
5. R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes. *Information and Computation*, **100**: 1-40 (Part I), 41-77 (Part II), Academic Press. 596
6. J. Parrow, D. Sangiorgi. Algebraic Theories for Name-Passing Calculi. *Journal of Information and Computation*, **120**: 174-197, 1995. 602
7. J. Parrow, B. Victor. The Update Calculus. *AMAST '97*, Lecture Notes in Computer Science 1119, Springer, 389–405, 1997. 596
8. J. Parrow, B. Victor. The Tau-Laws of Fusion. *CONCUR '98*, Lecture Notes in Computer Science 1466, Springer, 99–114, 1998. 596, 597, 598, 609
9. D. Sangiorgi. A Theory of Bisimulation for π-Calculus. *CONCUR 93*, Lecture Notes in Computer Science 715, Springer, 1993. 600
10. B. Victor, J. Parrow. Concurrent Constraints in the Fusion Calculus. *ICALP '98*, Lecture Notes in Computer Science 1443, Springer, 455–469, 1998. 596

A Axiomatic System for the Strong Open Congruence

E1	$P = P$	
E2	$P = Q$	if $Q = P$
E3	$P = R$	if $P = Q$ and $Q = R$
C1	$\alpha[x].P = \alpha[x].Q$	if $P = Q$
C2	$(x)P = (x)Q$	if $P = Q$
C3a	$[x{=}y]P = [x{=}y]Q$	if $P = Q$
C3b	$[x{\neq}y]P = [x{\neq}y]Q$	if $P = Q$
C4	$P{+}R = Q{+}R$	if $P = Q$
C5	$P_0{\mid}P_1 = Q_0{\mid}Q_1$	if $P_0 = Q_0$ and $P_1 = Q_1$
L1	$(x)\mathbf{0} = \mathbf{0}$	
L2	$(x)\alpha[y].P = \mathbf{0}$	$x \in \{\alpha, \overline{\alpha}\}$
L3	$(x)\alpha[y].P = \alpha[y].(x)P$	$x \notin \{y, \alpha, \overline{\alpha}\}$
L4	$(x)(y)P = (y)(x)P$	
L5	$(x)[y{=}z]P = [y{=}z](x)P$	$x \notin \{y, z\}$
L6	$(x)[x{=}y]P = \mathbf{0}$	$x{\neq}y$
L7	$(x)(P{+}Q) = (x)P{+}(x)Q$	
L8	$(x)[y{\mid}z].P = [y{\mid}z].(x)P$	$x \notin \{y, z\}$
L9	$(x)[y{\mid}x].P = \tau.P[y/x]$	$y \neq x$
L10	$(x)[x{\mid}x].P = \tau.(x)P$	
M1	$\phi P = \psi P$	if $\phi \Leftrightarrow \psi$
M2	$[x{=}y]P = [x{=}y]P[y/x]$	
M3a	$[x{=}y](P{+}Q) = [x{=}y]P{+}[x{=}y]Q$	
M3b	$[x{\neq}y](P{+}Q) = [x{\neq}y]P{+}[x{\neq}y]Q$	
M4	$P = [x{=}y]P{+}[x{\neq}y]P$	
M5	$[x{\neq}x]P = \mathbf{0}$	
S1	$P{+}\mathbf{0} = P$	
S2	$P{+}Q = Q{+}P$	
S3	$P{+}(Q{+}R) = (P{+}Q){+}R$	
S4	$P{+}P = P$	
U1	$[y{\mid}x].P = [x{\mid}y].P$	
U2	$[y{\mid}x].P = [y{\mid}x].[x{=}y]P$	
U3	$[x{\mid}x].P = \tau.P$	
LD1	$(x)[x{\mid}x].P = [y{\mid}y].(x)P$	by U3 and L8
LD2	$(x)[y{\neq}z]P = [y{\neq}z](x)P$	by L5, L7 and M4
LD3	$(x)[x{\neq}y]P = (x)P$	by L6, L7 and M4
MD1	$[x{=}y].\mathbf{0} = \mathbf{0}$	by S1, S4 and M4
MD2	$[x{=}x].P = P$	by M1
MD3	$\phi P = \phi(P\sigma)$ where σ agrees with ϕ	by M2
SD1	$\phi P{+}P = P$	by S-rules and M4
UD1	$[y{\mid}x].P = [y{\mid}x].P[y/x]$	by U2 and M2

Author Index