

Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2422

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Tokyo

Hélène Kirchner Christophe Ringeissen (Eds.)

Algebraic Methodology and Software Technology

9th International Conference, AMAST 2002
Saint-Gilles-les-Bains, Reunion Island, France
September 9-13, 2002
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Hélène Kirchner
LORIA - CNRS & INRIA, Campus Scientifique, Bâtiment LORIA
BP 239, 54506 Vandoeuvre-lès-Nancy, France
E-mail: helene.kirchner@loria.fr

Christophe Ringeissen
LORIA - INRIA
615, rue du Jardin Botanique
54602 Villers-lès-Nancy, France
E-mail: christophe.ringeissen@loria.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algebraic methodology and software technology : 9th international conference ;
proceedings / AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France,
September 9 - 13, 2002. Hélène Kirchner ; Christophe Ringeissen (ed.). -
Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ;
Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2422)
ISBN 3-540-44144-1

CR Subject Classification (1998): F.3-4, D.2, C.3, D.1.6, I.2.3, I.1.3

ISSN 0302-9743

ISBN 3-540-44144-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

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

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10873829 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of AMAST 2002, the 9th International Conference on Algebraic Methodology and Software Technology, held during September 9–13, 2002, in Saint-Gilles-les-Bains, Réunion Island, France. The major goal of the AMAST conferences is to promote research that may lead to setting software technology on a firm mathematical basis. This goal is achieved through a large international cooperation with contributions from both academia and industry. Developing a software technology on a mathematical basis produces software that is: (a) correct, and the correctness can be proved mathematically, (b) safe, so that it can be used in the implementation of critical systems, (c) portable, i.e., independent of computing platforms and language generations, (d) evolutionary, i.e., it is self-adaptable and evolves with the problem domain.

All previous AMAST conferences, which were held in Iowa City (1989, 1991), Twente (1993), Montreal (1995), Munich (1996), Sydney (1997), Manaus (1999), and Iowa City (2000), made contributions to the AMAST goals by reporting and disseminating academic and industrial achievements within the AMAST area of interest. During these meetings, AMAST attracted an international following among researchers and practitioners interested in software technology, programming methodology, and their algebraic, and logical foundations.

There were 59 submissions of overall high quality, authored by researchers from countries including Algeria, Argentina, Australia, Belgium, Brazil, Canada, China, Denmark, France, Germany, India, Italy, Japan, Korea, New Zealand, Poland, Portugal, Russia, Spain, Sweden, Switzerland, The Netherlands, the USA, and UK. All submissions were thoroughly evaluated, and an electronic program committee meeting was held through the Internet. The program committee selected 26 regular papers plus 2 system descriptions. This volume also includes full papers of six invited lectures given by Gilles Barthe, José Fiadeiro, Dale Miller, Peter Mosses, Don Sannella, and Igor Walukiewicz.

We warmly thank the members of the program committee and all the referees for their care and time in reviewing and selecting the submitted papers, Teodor Knapik for the practical organization of the conference, and all the institutions that supported AMAST 2002: Conseil Régional de la Réunion, Conseil Général de la Réunion, Conseil Scientifique de l'Université de la Réunion, and Faculté des Sciences et Technologies de l'Université de la Réunion.

June 2002

Hélène Kirchner
Christophe Ringeissen

Program Committee Chairs

Hélène Kirchner (LORIA-CNRS Nancy)
Christophe Ringeissen (LORIA-INRIA Nancy)

Program Committee

V.S. Alagar	U. Santa Clara, USA
Egidio Astesiano	DISI-U. Genova, Italy
Michel Bidoit	ENS Cachan, France
Dominique Bolignano	Trusted Logic, France
Manfred Broy	TU Muenchen, Germany
José Fiadeiro	U. Lisbon, Portugal
Bernd Fischer	NASA Ames Research Center, USA
Kokichi Futatsugi	JAIST, Japan
Armando Haeberer	PUC-Rio, Brazil
Nicolas Halbwachs	VERIMAG, France
Anne Haxthausen	TU Lyngby, Denmark
Dieter Hutter	DFKI, U. Saarbruecken, Germany
Paola Inverardi	U. L'Aquila, Italy
Bart Jacobs	U. Nijmegen, The Netherlands
Michael Johnson	U. Macquarie Sydney, Australia
Helene Kirchner	LORIA-CNRS Nancy, France
Paul Klint	CWI and U. Amsterdam, The Netherlands
Thomas Maibaum	King's College London, UK
Zohar Manna	U. Stanford, USA
Jon Millen	SRI International, USA
Peter Mosses	U. Aarhus, Denmark
Fernando Orejas	UPC, Barcelona, Spain
Ruy de Queiroz	UFPE, Recife, Brazil
Teodor Rus	U. Iowa, USA
Christophe Ringeissen	LORIA-INRIA Nancy, France
Don Sanella	U. Edinburgh, UK
Pierre-Yves Schobbens	U. Namur, Belgium
Giuseppe Scollo	U. Verona, Italy
Andrzej Tarlecki	U. Warsaw, Poland
Martin Wirsing	LMU Muenchen, Germany

Local Organization

Teodor Knapik (U. de la Réunion)

External Reviewers

L. Aceto	R. Hennicker	I. Nunes
K. Adi	P. Hoffman	K. Ogata
D. Aspinall	E. Hubbers	M. Oostdijk
H. Baumeister	J. Juerjens	S. Petitjean
M.A. Bednarczyk	A. Knapp	J. Philipps
B. Blanc	M. Koehler	N. Piccinini
A. Boisseau	M. Konarski	E. Poll
O. Bournez	P. Kosiuczenko	A. Pretschner
R. Bruni	Y. Lakhnech	J.-F. Raskin
H. Cirstea	S. Lasota	G. Reggio
M. Colon	P. Le Gall	P. Resende
M. Debbabi	M. Leucker	S. Sankaranarayanan
D. Deharbe	L. Liquori	B. Schaetz
R. De Nicola	A. Lopes	A. Schairer
R. Diaconescu	P. Machado	H. Sipma
S. Donatelli	K. MacKenzie	K. Spies
B. Finkbeiner	H. Mantel	A. Vilbig
K. Fischer	M. Matsumoto	M. Wermelinger
G. Ferrari	M. Mejri	S. Yovine
J. Goguen	S. Merz	O. Zendra
J. Goubault-Larrecq	M. Nakamura	
M.R. Hansen	M. Nesi	

Table of Contents

Invited Papers

From Specifications to Code in CASL	1
<i>David Aspinall and Donald Sannella</i>	
Automata and Games for Synthesis	15
<i>Igor Walukiewicz</i>	
Pragmatics of Modular SOS	21
<i>Peter D. Mosses</i>	
Tool-Assisted Specification and Verification of the JavaCard Platform	41
<i>Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa</i>	
Higher-Order Quantification and Proof Search	60
<i>Dale Miller</i>	
Algebraic Support for Service-Oriented Architecture	75
<i>José Luiz Fiadeiro</i>	

Regular Papers

Fully Automatic Adaptation of Software Components Based on Semantic Specifications	83
<i>Christian Haack, Brian Howard, Allen Stoughton, and Joe B. Wells</i>	
HASCASL: Towards Integrated Specification and Development of Functional Programs	99
<i>Lutz Schröder and Till Mossakowski</i>	
Removing Redundant Arguments of Functions	117
<i>María Alpuente, Santiago Escobar, and Salvador Lucas</i>	
A Class of Decidable Parametric Hybrid Systems	132
<i>Michaël Adélaïde and Olivier Roux</i>	
Vacuity Checking in the Modal Mu-Calculus	147
<i>Yifei Dong, Beata Sarna-Starosta, C.R. Ramakrishnan, and Scott A. Smolka</i>	
On Solving Temporal Logic Queries	163
<i>Samuel Hornus and Philippe Schnoebelen</i>	
Modelling Concurrent Behaviours by Commutativity and Weak Causality Relations	178
<i>Guangyuan Guo and Ryszard Janicki</i>	

An Algebra of Non-safe Petri Boxes	192
<i>Raymond Devillers, Hanna Klaudel, Maciej Koutny, and Franck Pommereau</i>	
Refusal Simulation and Interactive Games	208
<i>Irek Ulidowski</i>	
A Theory of May Testing for Asynchronous Calculi with Locality and No Name Matching	223
<i>Prasanna Thati, Reza Ziaei, and Gul Agha</i>	
Equational Axioms for Probabilistic Bisimilarity	239
<i>Luca Aceto, Zoltán Ésik, and Anna Ingólfssdóttir</i>	
Bisimulation by Unification	254
<i>Paolo Baldan, Andrea Bracciali, and Roberto Bruni</i>	
Transforming Processes to Check and Ensure Information Flow Security ..	271
<i>Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi</i>	
On Bisimulations for the Spi Calculus	287
<i>Johannes Borgström and Uwe Nestmann</i>	
Specifying and Verifying a Decimal Representation in Java for Smart Cards	304
<i>Cees-Bart Breunesse, Bart Jacobs, and Joachim van den Berg</i>	
A Method for Secure Smartcard Applications	319
<i>Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel</i>	
Extending JML Specifications with Temporal Logic	334
<i>Kerry Trentelman and Marieke Huisman</i>	
Algebraic Dynamic Programming	349
<i>Robert Giegerich and Carsten Meyer</i>	
Analyzing String Buffers in C	365
<i>Axel Simon and Andy King</i>	
A Foundation of Escape Analysis	380
<i>Patricia M. Hill and Fausto Spoto</i>	
A Framework for Order-Sorted Algebra	396
<i>John G. Stell</i>	
Guarded Transitions in Evolving Specifications	411
<i>Dusko Pavlovic and Douglas R. Smith</i>	
Revisiting the Categorical Approach to Systems	426
<i>Antónia Lopes and José Luiz Fiadeiro</i>	
Proof Transformations for Evolutionary Formal Software Development	441
<i>Axel Schairer and Dieter Hutter</i>	
Sharing Objects by Read-Only References	457
<i>Mats Skoglund</i>	

Class-Based versus Object-Based: A Denotational Comparison 473
Bernhard Reus

System Descriptions

BRAIN: Backward Reachability Analysis with Integers 489
Tatiana Rybina and Andrei Voronkov

The Development Graph Manager MAYA 495
Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer

Author Index 503

From Specifications to Code in CASL

David Aspinall and Donald Sannella

Laboratory for Foundations of Computer Science
Division of Informatics, University of Edinburgh

Abstract. The status of the Common Framework Initiative (CoFI) and the Common Algebraic Specification Language (CASL) are briefly presented. One important outstanding point concerns the relationship between CASL and programming languages; making a proper connection is obviously central to the use of CASL specifications for software specification and development. Some of the issues involved in making this connection are discussed.

1 Introduction

The *Common Framework Initiative*, abbreviated CoFI, is an open international collaboration which aims to provide a common framework for algebraic specification and development of software by consolidating the results of past research in the area [AKBK99]. CoFI was initiated in 1995 in response to the proliferation of algebraic specification languages. At that time, despite extensive past collaboration between the main research groups involved and a high degree of agreement concerning the basic concepts, the field gave the appearance of being extremely fragmented, and this was seen as a serious obstacle to the dissemination and use of algebraic development techniques. Although many tools supporting the use of algebraic techniques had been developed in the academic community, none of them had gained wide acceptance, at least partly because of their isolated usability, with each tool using a different specification language.

The main activity of CoFI has been the design of a specification language called CASL (the *Common Algebraic Specification Language*), intended for formal specification of functional requirements and modular software design and subsuming many previous specification languages. Development of prototyping and verification tools for CASL leads to them being interoperable, i.e. capable of being used in combination rather than in isolation. Design of CASL proceeded hand-in-hand with work on semantics, methodology and tool support, all of which provided vital feedback regarding language design proposals. For a detailed description of CASL, see [ABK⁺03] or [CoF01]; a tutorial introduction is [BM01] and the formal semantics is [CoF02]. Section 2 below provides a brief taste of its main features. All CASL design documents are available from the main CoFI web site [CoF].

The main activity in CoFI since its inception has been the design of CASL including its semantics, documentation, and tool support. This work is now essentially finished. The language design is complete and has been approved by IFIP

WG1.3 following two rounds of reviewing. The formal semantics of CASL is complete, and documentation including a book containing a user's guide and reference documents are nearing completion. A selection of tools supporting CASL are available, the most prominent of these being the CASL Tool Set CATS [Mos00a] which combines a parser, static checker, L^AT_EX pretty printer, facilities for printing signatures of specifications and structure graphs of CASL specifications, with an encoding of CASL specifications into second-order logic [Mos03] providing links to various verification and development systems including Isabelle and INKA [AHMS99].

The most focussed collaborative activity nowadays is on tool development, see <http://www.tzi.de/cofi/Tools>. There is also some further work on various topics in algebraic specification in the CASL context; for two recent examples see [MS02] and [BST02b]. However, with the completion of the design activity, there has been a very encouraging level of use of CASL in actual applications. In contrast with most previous algebraic specification languages which are used only by their inventors and their students and collaborators, many present CASL users have had no connection with its design. CASL has begun to be used in industry, and applications beyond software specification include a tentative role in the OMDoc project for the communication and storage of mathematical knowledge [Koh02]. Overall, CASL now appears to be recognized as a *de facto* standard language for algebraic specification.

One important aspect of any specification language is the way that specifications relate to programs. This can be a difficult issue; see [KS98] for a discussion of some serious problems at the specification/program interface that were encountered in work on Extended ML. The approach taken in CASL, which is outlined in Section 3, involves abstracting away from the details of the programming language. This works up to a point, but it leaves certain questions unanswered. In Section 4 we briefly outline a number of issues with the relationship between CASL specifications and programs that deserve attention. Our aim here is to raise questions, not to answer them. The discussion is tentative and we gloss over most of the technical details.

2 A Taste of CASL

A CASL *basic specification* denotes a class of many-sorted partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. These are classified by signatures which list sort names, partial and total function names, and predicate names, together with profiles of functions and predicates. The sorts are partially ordered by a subsort inclusion relation. Apart from introducing components of signatures, a CASL basic specification includes axioms giving properties of structures that are to be considered as models of the specification. Axioms are in first-order logic built over atomic formulae which include strong and existential equalities, definedness formulae and predicate applications, with generation constraints added as special, non-

first-order sentences. Concise syntax is provided for specifications of “datatypes” with constructor and selector functions.

Here is an example of a basic specification:

```

free types  $Nat ::= 0 \mid$  sort  $Pos;$ 
              $Pos ::= suc(pre : Nat)$ 
op  $pre : Nat \rightarrow? Nat$ 
axioms
     $\neg def\ pre(0);$ 
     $\forall n : Nat \bullet pre(suc(n)) = n$ 
pred  $even\_ : Nat$ 
var  $n : Nat$ 
    •  $even\ 0$ 
    •  $even\ suc(n) \Leftrightarrow \neg even\ n$ 

```

The remaining features of CASL do not depend on the details of the features for basic specifications, so this part of the design is orthogonal to the rest. This is reflected in the semantics by the use of a variant of the notion of institution [GB92] called an *institution with symbols* [Mos00b]. (For readers who are unfamiliar with the notion of institution, it corresponds roughly to “logical system appropriate for writing specifications”.) The semantics of basic specifications is regarded as defining a particular institution with symbols, and the rest of the semantics is based on an arbitrary institution with symbols. An important consequence of this is that sub-languages and extensions of CASL can be defined by restricting or extending the language of basic specifications without the need to reconsider or change the rest of the language.

CASL provides ways of building complex specifications out of simpler ones — the simplest ones being basic specifications — by means of various specification-building operations. These include translation, hiding, union, and both free and loose forms of extension. A *structured specification* denotes a class of many-sorted partial first-order structures, as with basic specifications. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. Structured specifications may be named and a named specification may be generic, meaning that it declares some parameters that need to be instantiated when it is used. Instantiation is a matter of providing an appropriate argument specification together with a fitting morphism from the parameter to the argument specification. Generic specifications correspond to what is known in other specification languages as (pushout-style) parametrized specifications.

Here is an example of a generic specification (referring to a specification named PARTIAL_ORDER, which is assumed to declare the sort *Elem* and the predicate $-- \leq --$):

```

spec LIST_WITH_ORDER [PARTIAL_ORDER] =
  free type  $List[Elem] ::= nil \mid cons(hd :? Elem; tl :? List[Elem])$ 
then

```

```

local
  op  $insert : Elem \times List[Elem] \rightarrow List[Elem]$ 
  vars  $x, y : Elem; l : List[Elem]$ 
  axioms  $insert(x, nil) = cons(x, nil);$ 
            $x \leq y \Rightarrow insert(x, cons(y, l)) = cons(x, insert(y, l));$ 
            $\neg(x \leq y) \Rightarrow insert(x, cons(y, l)) = cons(y, insert(x, l))$ 
within
  op  $order[-- \leq --] : List[Elem] \rightarrow List[Elem]$ 
  vars  $x : Elem; l : List[Elem]$ 
  axioms  $order[-- \leq --](nil) = nil;$ 
            $order[-- \leq --](cons(x, l)) = insert(x, order[-- \leq --](l))$ 
end

```

Architectural specifications in CASL are for describing the modular structure of software, in contrast to structured specifications where the structure is only for presentation purposes. An architectural specification consists of a list of unit declarations, indicating the component modules required with specifications for each of them, together with a unit term that describes the way in which these modules are to be combined. Units are normally functions which map structures to structures, where the specification of the unit specifies properties that the argument structure is required to satisfy as well as properties that are guaranteed of the result. These functions are required to be persistent, meaning that the argument structure is preserved intact in the result structure. This corresponds to the fact that a software module must use its imports as supplied without altering them.

Here is a simple example of an architectural specification (referring to ordinary specifications named LIST, CHAR, and NAT, assumed to declare the sorts *Elem* and *List[Elem]*, *Char*, and *Nat*, respectively):

```

arch spec CN_LIST =
  units
     $C : CHAR ;$ 
     $N : NAT ;$ 
     $F : ELEM \rightarrow LIST[ELEM]$ 
  result  $F[C \text{ fit } Elem \mapsto Char]$  and  $F[N \text{ fit } Elem \mapsto Nat]$ 

```

More about architectural specifications, including further examples, may be found in [BST02a].

3 Specifications and Programs

The primary use of specifications is to describe programs; nevertheless CASL abstracts away from all details of programming languages and programming paradigms, in common with most work on algebraic specification. The connection with programs is indirect, via the use of partial first-order structures or

similar mathematical models of program behaviour. We assume that each program P determines a CASL signature $Sig(P)$, and the programming language at hand comes equipped with a semantics which assigns to each program P its denotation as a partial first-order structure, $\llbracket P \rrbracket \in Alg(Sig(P))$. Then P is regarded as satisfying a specification SP if $Sig(P) = Sig(SP)$ and $\llbracket P \rrbracket \in \llbracket SP \rrbracket$, where $Sig(SP)$ and $\llbracket SP \rrbracket \subseteq Alg(Sig(SP))$ are given by the semantics of CASL.

The type systems of most programming languages do not match that of CASL, and partial first-order structures are not always suitable for capturing program behaviour. In that case one may simply replace the institution that is used for basic specifications in CASL with another one that is tailored to the programming language at hand.

Example 3.1. A suitable institution for Standard ML (SML) would consist of the following components.

Signatures: These would be SML signatures, or more precisely *environments* as defined in the static semantics of SML [MTHM97] which are their semantic counterparts. Components of signatures are then type constructors, typed function symbols including value constructors, exception constructors, and substructures having signatures. The type system here is that of SML, where functions may be higher-order and/or polymorphic.

Models: Any style of model that is suitable for capturing the behaviour of SML programs could be used. For example, one could take environments as defined in the *dynamic* semantics of SML, where closures are used to represent functions.

Sentences: One choice would be the syntax used for axioms in Extended ML [KST97], which is an extension of the syntax of SML boolean expressions by quantifiers, extensional equality, and a termination predicate.

Satisfaction: If sentences are as in EML and models are as in the dynamic semantics of SML, then satisfaction of a sentence by a model is as defined in the verification semantics of EML [KST97]. \square

Here is a variant of the sorting specification shown earlier, given for the SML instantiation of CASL, by adjusting the syntax of basic specifications.

```
spec LIST_WITH_POLYORDER =
  local
    val insert : ( $\alpha \times \alpha \rightarrow bool$ )  $\rightarrow$   $\alpha \times \alpha list \rightarrow \alpha list$ 
    vars  $x, y : \alpha; l : \alpha list$ 
    axioms insert leq ( $x, nil$ ) = cons( $x, nil$ );
           leq( $x, y$ )  $\Rightarrow$  insert leq ( $x, cons(y, l)$ ) =
                cons( $x, insert leq (y, l)$ );
            $\neg$ (leq( $x, y$ ))  $\Rightarrow$  insert leq ( $x, cons(y, l)$ ) =
                cons( $y, insert leq (x, l)$ )
  within
    op order : ( $\alpha \times \alpha \rightarrow bool$ )  $\rightarrow$   $\alpha list \rightarrow \alpha list$ 
    vars  $x : \alpha; l : \alpha list$ 
```

axioms $order\ leq\ (nil) = nil;$
 $order\ leq\ (cons(x, l)) = insert\ leq\ (x, order\ leq\ l)$

Example 3.2. An institution for a simplified version of Java could be defined by combining and recasting ideas from μ Java [NOP00] and JML [LBR01].

Signatures: A signature would represent the type information for a collection of classes, including class names, types and names of fields and methods in each class, and the subclass relationship between classes.

Models: A natural choice for models is based on execution traces within an abstract version of the Java Virtual Machine [NOP00]. A trace is a sequence of states, each including a representation of the heap (the current collection of objects), as well as a program counter indicating the next method to be executed and a stack containing the actual arguments it will be invoked with.

Sentences: An appropriate choice would be a many-sorted first-order logic which has non side-effecting Java expressions as terms. When specifying object-oriented systems, it is desirable to allow both *class invariants* which express properties of the values of fields in objects, as well as *method pre-post conditions* which express the behaviour of methods. Post conditions need some way to refer to the previous state (possibilities are to use auxiliary variables as in Hoare logic [NOP00], or the *Old*($-$) function of JML [LBR01]), as well as the result value of the method for non void returning methods. It would also be possible to add constructs for specifying exceptional behaviour and termination conditions.

Satisfaction: Roughly, a class invariant is satisfied in a model if it is satisfied in every state in the execution trace, and a method pre-post condition is satisfied if the pre-condition implies the post condition for all pairs of states corresponding to the particular method's call and return points. In practice, we need to be careful about certain intermediate states where class invariants may be temporarily violated; see [LBR01] for ways of describing this, as well as ways of specifying frame conditions which restrict which part of the state a method is allowed to alter. \square

Example 3.3. An institution for Haskell-with-CASL is described in [SM02]. It relates closely to the normal first-order CASL institution, and has been studied in more detail than our sketches for Java and SML above. Here is an overview:

Signatures: These consist of Haskell-style type classes including type constructors, type aliases, and type schemes for operators. There is a way to reduce rich signatures to simpler ones close to ordinary CASL, except that higher-order functions are present.

Models: Models are over basic signatures, and given by intensional Henkin models. Like the institutions outlined above, this choice reflects a more computationally-oriented viewpoint, allowing particular models which capture operational interpretations.

Sentences and satisfaction: Full formulae are similar to those of first-order CASL, but are reduced to a restricted *internal* logic on which satisfaction is defined. \square

This institutional approach takes a model-theoretic view and says nothing about how sentences can be *proved* to hold. For this, one would require an associated *entailment system*, see [Mes89].

4 Some Unresolved Issues

Defining an institution for specifying programs in a given programming language, as in the above examples, provides a link between CASL and the programming language at hand. This gives an adequate framework for analysis of the process of developing programs from specifications by stepwise refinement using CASL architectural specifications, see e.g. [BST02a] and [BST02b].

Still, this seems to be only part of a bigger and more detailed story. Notice that the syntax of programs does not appear anywhere in the institutions outlined in Examples 3.1 and 3.2. One would expect a full account to take into consideration the structure of the programming language, rather than regarding it as a monolithic set of notations for describing a model.

4.1 Combining Specifications and Programs

We have made the point that specification structure is in general unrelated to program structure, and it often makes sense to use a completely different structure for an initial requirements specification than is used for the eventual implementation. If we are to connect the two formally, however, it is useful to have a path between them. This is what architectural specifications in CASL are intended to provide, as a mechanism for specifying implementation structure.

Architectural specifications in CASL can make use of certain *model building operators* for defining units. These are defined analogously to the specification building operators available for structured specifications. They including renaming, hiding, amalgamation, and the definition of generic units. The semantics of the model building operators is defined for an arbitrary institution with symbols; but once a specific programming language is chosen, it remains to decide how the model building operators can be realised [BST02a]. It may be that none, some, or all of them can be defined directly within the programming language.

Example 4.1. (Continuing Example 3.1) In SML, an architectural unit is a structure (possibly containing substructures) and a generic unit corresponds to a functor. The ways that units are combined in CASL correspond with the ways that structures and functors are combined in SML; it is possible to define renaming, hiding and amalgamation within the language itself. \square

Example 4.2. (Continuing Example 3.2) In Java, an architectural unit is perhaps best identified as a set of related classes belonging to the same package. Java has visibility modifiers and interfaces which control access to these units, but there is no dedicated program-level syntax for combining pieces in this higher level of organization. Instead, the operations for constructing architectural units in CASL must be simulated by *meta*-operations on Java program syntax. Moreover,

there is nothing corresponding to a generic unit: we must treat generic units as meta-level macros which are expanded at each instantiation. \square

Even if the CASL model building operators are definable within the programming language, it may be preferable to use source-level transformations to realise them. This choice will be part of explaining how to instantiate CASL to a particular programming language.

CASL provides model building operations for combining units, but has no built-in syntax for constructing basic units. One way to link to a specific programming language would be add syntax from the programming language for basic units (and perhaps also syntax for combining units, e.g. SML functors). Here's a simple example of a unit definition using SML syntax (assuming PARTIAL_ORDER has been slightly adapted for SML syntax):

```

unit PAIRORDER : PARTIAL_ORDER =
  struct
    type Elem = int  $\times$  int;
    fun leq((x1, x2), (y1, y2)) = (x1 < x2 orelse
                                   (x1 = x2 andalso y1  $\leq$  y2))
  end

```

This mechanism gives a CASL-based language for writing specifications with pieces of programs inside; such specifications induce proof obligations.

Conversely, we would also like to explain so-called “wide-spectrum” approaches to specification, in which pieces of specification are written inside programs, as exemplified by Extended ML and JML. Although adopting the wide-spectrum approach throughout a formal development may risk early commitment to a particular program structure, it is appealing for programmers because they can be introduced to specification annotations gradually, rather than needing to learn a whole new language.

If we have an institution \mathcal{I}_{PL} for a programming language PL , we can imagine a crude way of adding assertions to the language by attaching sentences ϕ from \mathcal{I}_{PL} to programs P . In reality, we would want to attach sentences more closely to the area of the program they refer to, which depends on the specific language being considered.

The two scenarios we have discussed can be visualized like this:



Here, “wide-spectrum PL ” is a programming language PL extended with specification annotations, and $CASL(\mathcal{I}_{PL})$ is a CASL extension for the institution \mathcal{I}_{PL} .

To make each side work, we need to consider how to combine the static semantics of the languages to interpret pieces of programs or specifications in their surrounding context. We have two frameworks for specifying, on the boundary of programming and specification, but they are not quite the same.

4.2 Models of Programs vs. Models of Specifications

The activities of specification and programming are quite different. Specification languages put a premium on the ability to express *properties* of functions and data in a convenient and concise way, while in programming languages the concern is with expressing *algorithms* and *data structures*. This “what vs. how” distinction leads to obvious differences in language constructs. More subtle differences arise on the level of models. As explained in Section 3, the CASL approach is to use the same kind of models for programs and for specifications. The following example from [ST96] shows how this might lead to problems.

Example 4.3. Let ϕ_{equiv} be a sentence which asserts that $equiv(n, m) = true$ iff the Turing machines with Gödel numbers n and m compute the same partial function (this is expressible in first-order logic with equality). Now consider the following specification:

```

local
  op  $equiv : Nat \times Nat \rightarrow Bool$ 
  axioms  $\phi_{equiv}$ 
within
  op  $opt : Nat \rightarrow Nat;$ 
  vars  $n : Nat$ 
  axioms  $equiv(opt(n), n) = true$ 

```

This specifies an optimizing function opt transforming TMs to equivalent TMs. (Axioms could be added to require that the output of opt is at least as efficient as its input.) If the models in use require functions to be computable, as in Examples 3.1 and 3.2, then this specification will have *no* models because there is no computable function $equiv$ satisfying ϕ_{equiv} . Yet there *are* computable functions opt having the required property, for instance the identity function on Nat . Thus this specification disallows programs that provide exactly the required functionality.¹ \square

This example demonstrates that there is a potential problem with the use of “concrete” models like those in the operational semantics of Standard ML. The problem does not disappear if one uses more “abstract” models, as in denotational semantics and Example 3.3. Such models impose restrictions on function spaces in order to interpret fixed point equations. Further restrictions are imposed in a desire to reflect more accurately the constraints that the programming language imposes, for example functions in a polymorphic language like

¹ One way out might be to use a *predicate* for $equiv$ instead of a function, extending models to interpret predicates in such a way that predicates are not required to be undecidable.

SML might be required to be *parametric*, i.e. behave uniformly for all type instances [BFSS90]. Imposing such restrictions, whatever they are, gives rise to examples like the one above. Whether or not such an example illustrates a problem that needs to be solved is a different question. There is also an effect on the meaning of quantification: $\forall f : \tau \rightarrow \tau . \phi$ is more likely to hold if the quantifier ranges over only the parametric functions [Wad89, PA93].

A different problem arises from the use of the same *signatures* for programs and specifications. For specification purposes, one might want richer signatures with auxiliary components (that are not meant to be implemented) for use in specifying the main components (that *are* meant to be implemented). An example is the use of predicates specifying class invariants in JML [LBR01]. The addition of such auxiliary components does not seem to present substantive problems, but it is not catered for by the simple view of the relationship between specifications and programs presented in Section 3.

4.3 Relationships between Levels and between Languages

A classical topic in semantics is compiler correctness, see e.g. [BL69, Mor73], where the following diagram plays a central role:

$$\begin{array}{ccc} PL & \xrightarrow{c} & PL' \\ \llbracket \cdot \rrbracket \downarrow & & \downarrow \llbracket \cdot \rrbracket' \\ M & \xleftarrow{e} & M' \end{array}$$

Here, PL is the source language, PL' is the target language, c is a compiler, $\llbracket \cdot \rrbracket$ and $\llbracket \cdot \rrbracket'$ are the semantics of PL and PL' respectively, and e is some kind of encoding or simulation relating the results delivered by $\llbracket \cdot \rrbracket$ and $\llbracket \cdot \rrbracket'$. The compiler c is said to be correct if the diagram commutes, i.e. if for any $P \in PL$ the result of interpreting P is properly related by e to the result of interpreting $c(P) \in PL'$.

Given institutions for PL and PL' , we may recast this to require compilation to preserve satisfaction of properties. One formulation would be to require that for all programs $P \in PL$ and sentences ϕ over $Sig(P)$,

$$\llbracket P \rrbracket \models \phi \quad \Longrightarrow \quad \llbracket \widehat{P} \rrbracket' \models \widehat{\phi}$$

where \widehat{P} is the result of compiling P and $\widehat{\phi}$ is a translation of the property ϕ into the terms of \widehat{P} . If the set of sentences over $Sig(P)$ is closed under negation then this is equivalent to what is obtained if we replace \Rightarrow by \Leftrightarrow , and then this is a so-called *institution encoding* [Tar00] (cf. [GR02] where the name “forward institution morphism” is used for the same concept).

An issue of great practical importance is “interlanguage working” in which programs in one programming language can work directly with libraries and components written in another. One way to achieve this is via a low-level intermediate language which is the common target of compilers for the programming

languages in question. Microsoft's .NET platform is an example, with the CIL common intermediate language [MG01]. In such a framework, faced with an architectural specification of the form:

```

arch spec ASP =
  units
    U1 : SP1 ;
    U2 : SP2
  result ... U1 ... U2 ...

```

one might consider implementing $U1$ in one language and $U2$ in a different language. If $U1$ satisfies $SP1$ and $U2$ satisfies $SP2$, then it would be possible to combine $\widehat{U1}$ and $\widehat{U2}$ as indicated in ASP to obtain a program in the common intermediate language, with (under appropriate conditions) $\widehat{U1}$ satisfying $\widehat{SP1}$ and $\widehat{U2}$ satisfying $\widehat{SP2}$.

5 Conclusion

We have considered various issues in connecting programming languages to CASL, including the possibility of connecting several languages at once to allow inter-language implementations.

Once we have characterised a setting for working with specifications and programs, there is more to do before we have a framework for formal development. Perhaps the most important question is: what do we actually want to do with our specifications?

There are various possibilities. We can use specifications as a basis for testing [Gau95]; they might be used to generate code for run-time error checking [WLAG93, LLP⁺00], or they may be used as a basis for additional static analysis [DLNS98].

The traditional hope is to be able to prove properties about specifications, and to prove that implementations satisfy specifications. A common approach for this is to connect the specification language to an already existing theorem prover. There is a range of strategies here. The extremes are represented by a *shallow embedding* which formalizes just the semantics of the represented language directly within a theorem prover's logic, and a *deep embedding*, which formalizes the syntax of the language being represented, together with a meaning function which describes its semantics within the theorem prover's logic [RAM⁺92]. Shallow embeddings have the advantage of direct reasoning within the theorem prover's logic, but require a degree of compatibility between the logic and the language being formalized. A deep embedding, by contrast, is more flexible, but typically more difficult to use, so that proof principles for the represented language may have to be derived. Deep embeddings may also allow formalization of meta-properties of the language being represented, although whether this is useful or not depends on the application. Among the existing work in connecting theorem provers to languages, the work on CATS [Mos00a] and

HasCASL [SM02] use shallow embeddings, whereas the work on μ Java [NOP00] and ASL+FPC [Asp97] use deep embeddings; in the case of μ Java the object of the formalization is to prove meta-properties about Java, rather than reason about Java programs.

References

- [ABK⁺03] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 2003. To appear. 1
- [AHMS99] S. Autexier, D. Hutter, H. Mantel and A. Schairer. System description: INKA 5.0 – a logic voyager. *Proc. 16th Intl. Conf. on Automated Deduction*. Springer LNAI 1632, 207–211, 1999. 2
- [AKBK99] E. Astesiano, B. Krieg-Brückner and H.-J. Kreowski, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999. 1
- [Asp97] D. Aspinall. *Type Systems for Modular Programming and Specification*. PhD thesis, University of Edinburgh, 1997. 12
- [BFSS90] E. Bainbridge, P. Freyd, A. Scedrov and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990. 10
- [BL69] R. Burstall and P. Landin. Programs and their proofs: An algebraic approach. B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, 17–43. Edinburgh University Press, 1969. 10
- [BM01] M. Bidoit and P. Mosses. A gentle introduction to CASL. Tutorial, WADT/CoFI Workshop at ETAPS 2001, Genova. Available from <http://www.lsv.ens-cachan.fr/~bidoit/CASL/>, 2001. 1
- [BST02a] M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 2002. To appear. 4, 7
- [BST02b] M. Bidoit, D. Sannella and A. Tarlecki. Global development via local observational construction steps. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, Warsaw. Springer LNCS, 2002. To appear. 2, 7
- [CoF] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from <http://www.brics.dk/Projects/CoFI/>. 1, 12
- [CoF01] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0.1. [Documents/CASL/v1.0.1/Summary](#), in [CoF], 2001. 1
- [CoF02] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics, version 1.0. [Documents/CASLSemantics](#), in [CoF], 2002. 1
- [DLNS98] D. Detlefs, K. R. M. Leino, G. Nelson and J. Saxe. Extended static checking. Technical Report #159, Compaq SRC, Palo Alto, USA, 1998. 11
- [Gau95] M.-C. Gaudel. Testing can be formal, too. *Intl. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, Aarhus. Springer LNCS 915, 82–96, 1995. 11
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery*, 39:95–146, 1992. 3

- [GR02] J. Goguen and G. Roşu. Institution morphisms. *Formal Aspects of Computing*, 2002. To appear. 10
- [Koh02] M. Kohlhase. OMDoc: An open markup format for mathematical documents, version 1.1. Available from <http://www.mathweb.org/omdoc/index.html>, 2002. 2
- [KS98] S. Kahrs and D. Sannella. Reflections on the design of a specification language. *Proc. Intl. Colloq. on Fundamental Approaches to Software Engineering. European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, Lisbon. Springer LNCS 1382, 154–170, 1998. 2
- [KST97] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997. 5
- [LBR01] G. Leavens, A. Baker and C. Ruby. Preliminary design of JML. Technical Report TR #98-06p, Department of Computer Science, Iowa State University, 2001. 6, 10
- [LLP⁺00] G. Leavens, K. R. M. Leino, E. Poll, C. Ruby and B. Jacobs. JML: notations and tools supporting detailed design in Java. *OOPSLA 2000 Companion*, Minneapolis, 105–106, 2000. 11
- [Mes89] J. Meseguer. General logics. *Logic Colloquium '87*, 275–329. North Holland, 1989. 7
- [MG01] E. Meijer and J. Gough. A technical overview of the common language infrastructure. Available from <http://research.microsoft.com/~emeijer/Papers/CLR.pdf>, 2001(?). 11
- [Mor73] F. Morris. Advice on structuring compilers and proving them correct. *Proc. 3rd ACM Symp. on Principles of Programming Languages*, 144–152. ACM Press, 1973. 10
- [Mos00a] T. Mossakowski. CASL: From semantics to tools. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, European Joint Conferences on Theory and Practice of Software, Berlin. Springer LNCS 1785, 93–108, 2000. 2, 11
- [Mos00b] T. Mossakowski. Specification in an arbitrary institution with symbols. *Recent Trends in Algebraic Development Techniques: Selected Papers from WADT'99*, Bonas. Springer LNCS 1827, 252–270, 2000. 3
- [Mos03] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 2003. To appear. 2
- [MS02] P. Machado and D. Sannella. Unit testing for CASL architectural specifications. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science*, Warsaw. Springer LNCS, 2002. To appear. 2
- [MTHM97] R. Milner, M. Tofte, R. Harper and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 5
- [NOP00] T. Nipkow, D. von Oheimb and C. Pusch. μ Java: Embedding a programming language in a theorem prover. F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Intl. Summer School Marktoberdorf 1999*, 117–144. IOS Press, 2000. 6, 12
- [PA93] G. Plotkin and M. Abadi. A logic for parametric polymorphism. *Proc. of the Intl. Conf. on Typed Lambda Calculi and Applications*, Amsterdam. Springer LNCS 664, 361–375, 1993. 10
- [RAM⁺92] R. Boulton, A. Gordon, M. Gordon, J. Herbert and J. van Tassel. Experience with embedding hardware description languages in HOL. *Proc. of*

- the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, Nijmegen, 129–156. North-Holland, 1992. [11](#)
- [SM02] L. Schröder and T. Mossakowski. HasCASL: Towards integrated specification and development of haskell programs. *Proc. 9th Intl. Conf. on Algebraic Methodology And Software Technology*, Reunion. Springer LNCS, this volume, 2002. [6](#), [12](#)
- [ST96] D. Sannella and A. Tarlecki. Mind the gap! Abstract versus concrete models of specifications. *Proc. 21st Intl. Symp. on Mathematical Foundations of Computer Science*, Cracow. Springer LNCS 1113, 114–134, 1996. [9](#)
- [Tar00] A. Tarlecki. Towards heterogeneous specifications. D. Gabbay and M. van Rijke, editors, *Proc. of the Intl. Conf. on Frontiers of Combining Systems (FroCoS'98)*, 337–360. Research Studies Press, 2000. [10](#)
- [Wad89] P. Wadler. Theorems for free! *Proc. of the 4th Intl. Conf. on Functional Programming and Computer Architecture*. ACM Press, 1989. [10](#)
- [WLAG93] R. Wahbe, S. Lucco, T. Anderson and S. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993. [11](#)

Automata and Games for Synthesis

Igor Walukiewicz

LaBRI, Université Bordeaux I, France

Abstract. The synthesis of controllers for discrete event systems, as introduced by Ramadge and Wonham, amounts to computing winning strategies in simple parity games. We describe an extension of this framework to arbitrary regular specifications on infinite behaviours. We also consider the extension to decentralized control. Finally, we discuss some means to compare quality of strategies.

1 Synthesis of Discrete Controllers

At the end of the eighties, Ramadge and Wonham introduced the theory of control of discrete event systems (see the survey [17] and the books [9] and [7]). In this theory a process (also called a *plant*) is a deterministic non-complete finite state automaton over an alphabet A of events, which defines all possible sequential behaviours of the process. Some of the states of the plant are termed *marked*.

The alphabet A is the disjoint union of two subsets: the set A_{cont} of controllable events and the set A_{unc} of uncontrollable events. A is also the disjoint union of the sets A_{obs} of observable events and A_{uno} of unobservable events.

A controller is a process R which satisfies the following two conditions:

- (C) For any state q of R , and for any uncontrollable event a , there is a transition from q labelled by a .
- (O) For any state q of R , and for any unobservable event a , if there is a transition from q labelled by a then this transition is a loop over q .

In other words, a controller must react to any uncontrollable event and cannot detect the occurrence of an unobservable event.

If P is a process and R is a controller, the supervised system is the product $P \times R$. Thus, if this system is in the state (p, r) and if for some controllable event a , there is no transition labelled by a from r , the controller forbids the supervised system to perform the event a . On the other hand, if an unobservable event occurs in the supervised system, the state of the controller does not change, as if the event had not occurred.

Of course, a supervised system has less behaviours than its plant alone. In particular a supervised system may be made to avoid states of the plant which are unwanted for some reason. One can also define a set of admissible behaviours of the plant, and the control problem is to find a controller R such that all behaviours of the supervised system are admissible. So the basic control problem is the following:

Given a plant P and a set S of behaviours, does there exist a controller R satisfying (C) and (O) such that the behaviours of the supervised system $P \times R$ are all in S ?

The synthesis problem is to construct such a controller if it does exist. Some variants of this problem take into account the distinction between terminal and non terminal behaviours of the plant (in [17] they are called marked and non marked behaviours).

2 Generalization of the Setting

In the approach described above the constraints on the behaviour of the supervised system say that all finite paths in the system are in some regular language, and/or that all words of a given language are paths of the system. Such types of constraints can be expressed by formulas of the modal μ -calculus: for each such constraint there is a formula Φ such that the supervised system satisfies Φ if and only if its behaviour satisfies the constraint. Therefore, one can extend (cf. [1]) the Ramadge-Wonham's approach by using any formula of the modal μ -calculus to specify the desired property of the supervised system. Hence, the control problem becomes:

Given a plant P and a formula Φ , does there exist a controller R satisfying (C) and (O) such that $P \times R$ satisfies Φ ?

It turns out that the condition (C) is also expressible in the modal μ -calculus by the formula $\nu x.(\bigwedge_{b \in A_{unc}} \langle b \rangle tt \wedge \bigwedge_{c \in A} [c]x)$. Therefore, a natural formulation of the problem can be:

(P) Given a plant P and two formulas Φ and Ψ , does there exist a controller R satisfying Ψ such that $P \times R$ satisfies Φ ?

An example of such a formula Ψ characterizing the controller is the following. Let a, c, f be three events where only c is controllable. The event f symbolizes a failure of the device which controls c so that after an occurrence of f , the event c becomes uncontrollable. The formula expressing this phenomenon is $\nu x.(\langle a \rangle x \wedge [c]x \wedge \langle f \rangle \nu y.(\langle a \rangle y \wedge \langle c \rangle y \wedge \langle f \rangle y))$. Another example is the case where only one out of two events c_1 and c_2 is controllable at a time. This is expressed by $\nu x.(\langle a \rangle x \wedge ((\langle c_1 \rangle x \wedge [c_2]x) \vee ([c_1]x \wedge \langle c_2 \rangle x)))$.

It remains to deal with the condition (O) which, unfortunately, is not expressible in the modal μ -calculus because it is not invariant under bisimulation. That is why the modal μ -calculus needs to be extended to a *modal-loop* μ -calculus. This extension consists in associating with each event a a basic proposition \circlearrowleft_a whose interpretation is that there is a transition a from the considered state q and the transition is a loop to the same state q . Having this, the condition (O) can be expressed as $\nu x.(\bigwedge_{a \in A_{obs}} [a]x \wedge \bigwedge_{a \in A_{uno}} ([a]false \vee \circlearrowleft_a))$. We can also express that an observable event becomes unobservable after a failure:

$\nu x.(\dots \wedge [a]x \wedge \langle f \rangle \nu y.(\dots \wedge ([a]false \vee \odot_a) \wedge \langle f \rangle y))$, or that at most one out of two events a and b is observable: $[a]false \vee \odot_a \vee [b]false \vee \odot_b$.

Therefore we consider problem **(P)** as the general form of a control problem when Φ and Ψ are modal-loop formulas.

It turns out, fortunately, that modal-loop μ -calculus has quite similar properties to the ordinary μ -calculus. For instance, and it is very convenient, modal automata have the same expressive power as the modal μ -calculus, and moreover, translating μ -formulas into automata and vice-versa is quite easy. One can introduce modal-loop automata [1] which are an extension of standard modal automata with the ability to test for existence of a loop. These loop automata are equivalent in expressive power to the loop μ -calculus in the same way as standard automata are equivalent to the standard μ -calculus [2]. The reason for this is simply that one can consider the property of having a loop as a local property of states, i.e., as unary predicates on states.

The two crucial properties of alternating automata, that are also shared by loop alternating automata are:

Eliminating alternation Every loop automaton is equivalent to a nondeterministic loop automaton.

Sat The emptiness of a nondeterministic loop automaton can be effectively decided and a process accepted by the automaton can be effectively constructed.

Paper [1] presents the construction of a modal-loop formula Φ/P that is satisfied by precisely those controllers R for which $P \times R \models \Phi$. This way a process R is a solution of the synthesis problem **P** if and only if $R \models (\Phi/P) \wedge \Psi$.

By the properties above, $(\Phi/P) \wedge \Psi$ can be effectively transformed into a nondeterministic loop automaton and a controller R can be synthesized. This transformation may cause an exponential blow-up in size, and the powerset construction given in [4] for dealing with the condition (O) is indeed a special case of this transformation. Checking emptiness of a nondeterministic loop automaton is essentially the same as solving the associated parity game. A winning strategy in the game gives a desired controller.

Therefore, all control problems of the form **(P)** are indeed satisfiability problems in the modal-loop μ -calculus and synthesis problems amount to finding models of modal-loop formulas. Effectiveness of solving these problems is ensured by the above properties. Indeed, finding such models consists in finding winning strategies in parity games, probably the most fascinating problem in the μ -calculus [21]. (Reciprocally, finding a winning strategy is itself a control problem: your moves are controllable and the moves of your opponent are not!)

3 Comparing Controllers

When considering program synthesis, correctness and size of the program is not the only criterion. As mentioned above, solving synthesis problem amounts to

finding a winning strategy in some game. In the framework of discrete controller synthesis of Ramadge and Wonham one is usually interested in the most permissive winning strategy, i.e., the one that allows most behaviours. This way one avoids the trivial solution which is a controller that forbids everything. In case of specifications involving infinitary conditions there may be no most permissive winning strategy. Actually it turns out that under suitable definitions most permissive strategies exist only for safety specifications. This is one of the reasons why discrete controller synthesis theory concentrates on safety properties.

Still, the setting of safety specifications and safety games may not be sufficient for some control synthesis problems. In fact, infinitary conditions such as fairness or liveness cannot be expressed in the context of safety games. This justifies the above described extension of the setting to μ -calculus expressible properties and to parity games. This class of games is sufficiently expressive, as every game with regular conditions can be encoded as a parity game [15,19].

Even though there may be no most permissive strategy in a parity game, it still may be reasonable to compare strategies by looking at the set of behaviours they allow. A strategy permitting more behaviours is better because it is less prone to transient errors (i.e. errors that do not permit a controller to make some choices for a limited period of time). Imagine that we are in such an error situation. A more permissive strategy instead of waiting until the error will be resolved may be able to take a different action. More permissive strategy is also good for modular design. Imagine that later we are going to refine computed strategy (controller). If a strategy is too restrictive then it may not allow some admissible behaviours, and hence may not allow the required refinement.

In [5] a notion of a *permissive strategy* is proposed. A strategy for a given game is permissive if it allows all the behaviours of all memoryless winning strategies in the game. It turns out that for every game there is a permissive strategy with finite memory. This strategy can be computed in $\mathcal{O}(n^{d/2+1})$ time and $\mathcal{O}(nd \log(n))$ space, where n is the number of vertices in the game and d is the number of different integers that are labelling vertices of the game. This matches known upper-bound for the simpler problem of computing a set of positions from which the given player has a winning strategy. The algorithm actually turns out to be the same as the strategy improvement algorithm of Jurdziński [8].

4 Decentralized Control

In Ramadge and Wonham setting one can also consider the synthesis of decentralized controllers [3,18]: a plant can be supervised by several independent controllers (instead of only one). The difficulty is that each controller has its own set of controllable and observable events. Hence the decentralized control problem is to find R_1, \dots, R_n such that the supervised system $P \times R_1 \times \dots \times R_n$ satisfies the specification S and for each i , R_i satisfies (C_i) and (O_i) . More generally, in our setting, a decentralized control problem is:

Given a plant P and modal-loop formulas $\Phi, \Psi_1, \dots, \Psi_n$, do there exist controllers R_i satisfying Ψ_i ($i = 1, \dots, n$) such that $P \times R_1 \times \dots \times R_n$ satisfies Φ ?

To solve this problem one can construct [1] a formula Φ/Ψ which is satisfied by a process R if and only if there exists a process P such that $P \models \Psi$ and $P \times R \models \Phi$. So, in the case when Ψ has only one model P , the meaning of Φ/Ψ is equivalent to Φ/P . If Ψ has more models then our construction works only in the case when Ψ is a formula of the standard μ -calculus, i.e., Ψ does not mention loops. Without this restriction the formula Φ/Ψ may not exist, since the set of all the processes R as described above may not be regular.

The construction of Φ/Ψ allows us to solve a decentralized control problem when at most one of the formulas Ψ_i contains loop predicates. One can show that if one allows two such formulas then the existence of a solution to the problem is undecidable [13,20,1].

5 Related Work

The results described above are presented in two papers [1,6]. The first extends the framework of Ramadge and Wonham [17] in two directions. It considers infinite behaviours and arbitrary regular specifications, while the standard framework deals only with specifications on the set of finite paths of processes. It also allows dynamic changes of the set of observable and controllable events. The second paper introduces and studies the notion of permissive strategies.

Kupferman and Vardi [10,11] consider a problem very similar to the problem **P**. They use different terminology, still it essentially amounts to the same setting but with a fixed set of observable and controllable actions. They do not consider the problem of synthesizing decentralized controllers.

Pnueli and Rosen [16] consider the problem of decentralized synthesis on given architectures. They show several decidability/undecidability results depending on the shape of the architecture. Their setting is quite different from the one we have here. It is not clear that their architectures can simulate controllability/observability assumptions. There are also architectures that cannot be expressed by controllability/observability assumptions. They consider only linear time specifications. Branching specifications in this setting are considered in [12].

Finally Maler, Pnueli and Sifakis [14] consider the problem of centralized synthesis in the presence of time constraints. They consider only linear time specifications and only the case when all the actions are observable.

References

1. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *TCS*, 2002. to appear. 16, 17, 19
2. Andr e Arnold and Damian Niwiński. *The Rudiments of the Mu-Calculus*, volume 146 of *Studies in Logic*. North-Holland, 2001. 17

3. G. Barrett and S. Lafortune. A novel framework for decentralized supervisory control with communication. In *IEEE SMC*, 1998. 18
4. A. Bergeron. A unified approach to control problems in discrete event processes. *RAIRO-ITA*, 27:555–573, 1993. 17
5. Julien Bernet and David Janin and Igor Walukiewicz. Permissive strategies: from parity games to safety games. *RAIRO*, 2002. to appear. 18
6. Julien Bernet, David Janin, and Igor Walukiewicz. Permissive strategies: From parity games to safety games. To appear in *Theoretical Informatics and Applications (RAIRO)*. 19
7. Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999. 15
8. Marcin Jurdziński. Small progress measures for solving parity games. In *STACS*, volume 1770 of *LNCS*, pages 290–301, 2000. 18
9. R. Kumar and V. K. Garg. *Modeling and control of logical discrete event systems*. Kluwer Academic Pub., 1995. 15
10. O. Kupferman and M. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, 1997. 19
11. O. Kupferman and M. Vardi. μ -calculus synthesis. In *MFCS 2000*, pages 497–507. *LNCS* 1893, 2000. 19
12. O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001. 19
13. H. Lamouchi and J. G. Thistle. Effective control synthesis for DES under partial observations. In *Proc. 39th IEEE Conf. on Decision and Control*, December 2000. 19
14. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E. W. Mayr and C. Puech, editors, *STACS'95*, volume 900 of *LNCS*, pages 229–242, 1995. 19
15. Andrzej W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Fifth Symposium on Computation Theory*, volume 208 of *LNCS*, pages 157–168, 1984. 18
16. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *31th IEEE Symposium Foundations of Computer Science (FOCS 1990)*, pages 746–757, 1990. 19
17. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77, 1989. 15, 16, 19
18. K. Rudie and W. Whonham. Think globally, act locally: Decentralized supervisory control. *IEEE Trans. on Automat. Control*, 37(11):1692–1708, 1992. 18
19. Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer-Verlag, 1997. 18
20. S. Tripakis. Undecidable problems of decentralized observation and control. In *IEEE Conference on Decision and Control*, 2001. 19
21. Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games (Extended abstract). In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 202–215, 2000. 17

Pragmatics of Modular SOS

Peter D. Mosses

BRICS* and Department of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
pdmosses@brics.dk
<http://www.brics.dk/~pdm>

Abstract. Modular SOS is a recently-developed variant of Plotkin’s Structural Operational Semantics (SOS) framework. It has several pragmatic advantages over the original framework—the most significant being that rules specifying the semantics of individual language constructs can be given *definitively*, once and for all.

Modular SOS is being used for teaching operational semantics at the undergraduate level. For this purpose, the meta-notation for modular SOS rules has been made more user-friendly, and derivation of computations according to the rules is simulated using Prolog.

After giving an overview of the foundations of Modular SOS, this paper gives some illustrative examples of the use of the framework, and discusses various pragmatic aspects.

1 Introduction

Structural Operational Semantics (SOS) [21] is a well-known framework that can be used for specifying the semantics of concurrent systems [1,9] and programming languages [10]. It has been widely taught, especially at the undergraduate level [7,20,21,22,23]. However, the modularity of SOS is quite poor: when extending a pure functional language with concurrency primitives and/or references, for instance, the SOS rules for all the functional constructs have to be completely reformulated [2].

As the name suggests, Modular SOS (MSOS) [13,14] is a variant of SOS that ensures a high degree of modularity: the rules specifying the MSOS of individual language constructs can be given *definitively*, once and for all, since their formulation is completely independent of the presence or absence of other constructs in the described language. When extending a pure functional language with concurrency primitives and/or references, the MSOS rules for the functional constructs don’t need even the slightest reformulation [17].

In denotational semantics, the problem of obtaining good modularity has received much attention, and has to a large extent been solved by introducing so-called monad transformers [11]. MSOS provides an analogous (but significantly simpler) solution for the structural approach to operational semantics.

* Basic Research in Computer Science (<http://www.brics.dk>), funded by the Danish National Research Foundation

The crucial feature of MSOS is to insist that states are merely abstract syntax and computed values, omitting the usual auxiliary information (such as environments and stores) that they include in SOS. The only place left for auxiliary information is in the labels on transitions. This seemingly minor notational change—coupled with the use of symbolic indices to access the auxiliary information—is surprisingly beneficial. MSOS rules for many language constructs can be specified independently of whatever components labels might have; rules that require particular components can access and set those components without mentioning other components at all. For instance, the MSOS rules for if-expressions do not require labels to have any particular components, and their formulation remains valid regardless of whether expressions are purely functional, have side-effects, raise exceptions, or interact with concurrent processes. Rules specifying the semantics of all individual language constructs can be given definitively in MSOS, once and for all.

MSOS is being used by the author for teaching operational semantics at the undergraduate level (5th semester) [19]. The original, rather abstract notation used for accessing and setting components of labels in MSOS rules [13,14] has been made more user-friendly, to increase perspicuity. Experimentally, derivation of computations according to MSOS rules is being simulated using Prolog, so that students can trace how the rules shown in examples actually work, and check whether their own rules for further language constructs provide the intended semantics.

Section 2 gives an overview of the foundations of MSOS, recalling the basic notions of transition systems and introducing meta-notation. Section 3 provides illustrative examples of MSOS rules for some constructs taken from Standard ML. Section 4 discusses various pragmatic aspects of MSOS, including tool support and the issue of how to choose between the small-step and big-step (“natural semantics”) styles. Section 5 concludes by indicating some topics left for future work.

2 Foundations

This section gives an overview of the foundations of MSOS. It explains the differences between conventional SOS and MSOS, and introduces the meta-notation that will be used for the illustrative examples in Sect. 3. A previous paper on MSOS [13] gives a more formal presentation, focussing on points of theoretical interest.

SOS and MSOS are both based on transition systems. As the name suggests, a transition system has a set of states Q and a set of transitions between states; the existence of a transition from Q_1 to Q_2 is written¹ $Q_1 \longrightarrow Q_2$. A *labelled* transition system has moreover a set of labels X , allowing different transitions

¹ For economy of notation, let us exploit names of sets such as Q also as meta-variables ranging over those sets, distinguishing different meta-variables over the same set by subscripts and/or primes.

$Q_1 \dashv X \rightarrow Q_2$ (usually written $Q_1 \xrightarrow{X} Q_2$) between Q_1 and Q_2 . Labels on transitions are optional in SOS, but obligatory in MSOS.

A *computation* in a transition system is a finite or infinite sequence of composable transitions $Q_i \rightarrow Q_{i+1}$. With labelled transitions $Q_i \dashv X_i \rightarrow Q_{i+1}$, the *trace* of the computation is the sequence of the labels X_i . In an *initial* transition system, computations are required to start from initial states I . In a *final* transition system, (finite) computations have to end in final states F , from which there can be no further transitions. Non-final states with no transitions are called *stuck*.

States in SOS and MSOS generally involve both *abstract syntax* (trees) and *computed values*. The abstract syntax of an entire program is used to form the initial state of a computation. Initial states involving parts of programs (declarations, expressions, statements, etc.) are required too, due to the structural nature of SOS and MSOS, whereby the transitions for a compound construct generally depend on transitions for its components. Final states for programs and their parts give computed values: declarations compute binding maps (i.e. environments), expressions compute the expected values, whereas statements compute only a fixed null value.

With the so-called *big-step* (or “natural semantics” [8]) style of SOS and MSOS, computations always go directly from initial states to final states in a single transition. The original *small-step* style favoured by Plotkin[21] requires computations to proceed gradually through intermediate states. Since initial states involve abstract syntax and final states involve computed values, it is unsurprising that intermediate states involve a mixture of these, in the form of abstract syntax trees where some nodes may have been replaced by their computed values. We refer to such mixed trees as *value-added* (abstract) syntax trees; they include pure syntax trees and computed values.

So much for the main features that SOS and MSOS have in common. Let us now consider their differences, which concern restrictions on the structure of states and labels, and whether labels affect the composability of transitions or not:

- In MSOS, states are restricted to value-added abstract syntax trees. Initial states are therefore pure abstract syntax, and final states are simply computed values. SOS, in contrast, allows states to include auxiliary components such as stores and environments.
- MSOS requires labels to be records (i.e. total maps on finite sets of indices, also known as indexed products). The components of these records are unrestricted, and usually include the auxiliary entities that would occur as components of states in SOS. Labels are often omitted altogether in SOS descriptions of sequential languages.
- In MSOS, transitions are composable only when their labels are composable, as described below. In SOS, composability of transitions is independent of their labels, and depends only on the states involved.

Each component of labels in MSOS has to be classified as inherited, variable, or observable. The classification affects composability of labels, as follows:

- When a component is classified as *inherited*, two labels are composable only when the value of this component is the same in both labels.
- A component classified as *variable* has both an initial and a final value in each label, and two labels X_1, X_2 are composable only when the final value of the component in X_1 is the same as its initial value in X_2 . When the index for the initial value of the component is i , the index for the final value is written i' .
- The possible values of a component classified as *observable* form a monoid (e.g. lists under concatenation). Its actual values do not affect composability.

When two labels X_1, X_2 are composable, their composition is determined in the obvious way, and written $X_1;X_2$.

A further significant difference between SOS and MSOS is that MSOS provides a built-in notion of *unobservable transition*, characterized simply by the components of the label: each variable component must remain constant, and each observable component must be the unit of the corresponding monoid (e.g. the empty list). The distinction between arbitrary labels X and labels U for unobservable transitions is crucial when formulating specifications in MSOS; it may also be used to define so-called observational equivalence.

Readers who are familiar with Category Theory may like to know that labels in MSOS are indeed the morphisms of a category, with the unobservable labels as identity morphisms (corresponding to the objects of the category, which are not referred to directly). Simple functors called basic label transformers can be used for adding new components to labels [13,14]. Surprisingly, it appears that this straightforward use of categories of labels had not previously been exploited in work on transition systems.

One of the most distinctive features of SOS is the way that axioms and inference rules (together referred to simply as *rules*) are used to specify transition systems that represent the semantics of programming languages. The premises and conclusions of the rules are assertions of transitions $t_1 \multimap t_2$ or $t_1 \longrightarrow t_2$, where t, t_1, t_2 are terms that may involve constructor operations, auxiliary operations, and meta-variables. Rules may also have side-conditions, which are often equations or applications of auxiliary predicates. It is common practice to write the side-conditions together with the premises of rules, and one may also use rules to defined auxiliary operations and predicates, so the general form of a rule is:

$$\frac{c_1, \dots, c_n}{c} \tag{1}$$

where each of $c, c_1, \dots, c_n (n \geq 0)$ may be a transition $t_1 \multimap t_2$ or $t_1 \longrightarrow t_2$, an equation $t_1 = t_2$, or an application $p(t_1, \dots, t_k)$ of a predicate p . Provided that the rules do not involve negations of assertions in the premises, they can be interpreted straightforwardly as inductive definitions of relations, so that assertions are satisfied in models if and only if they follow from the rules by ordinary logical inference. (The situation when negative premises are allowed is quite complicated [1], and not considered further here.)

Rules in MSOS have the same general form as in SOS. The rules specifying transitions for a programming construct are typically *structural*, having conclusions of the form $c(m_1, \dots, m_k) \rightarrow t$ where c is a constructor and m, m_1, \dots, m_k are simply meta-variables, and with premises involving one or more assertions of the form $m_i \rightarrow t'$; but note that being structural is not essential for a set of rules to specify a well-defined transition system.

MSOS provides some operations for records, finite maps, and lists; the notation corresponds closely to that used in SML and (for lists) in Prolog:

- A record with components t_1, \dots, t_n ($n \geq 0$) indexed respectively by i_1, \dots, i_n is written $\{i_1=t_1, \dots, i_n=t_n\}$; the value is independent of the order in which the components are written, and the indices must be distinct. When t' evaluates to a record not having the index i , the term $\{i=t|t'\}$ evaluates to that record extended with the value of t indexed by i (this notation is by analogy with Prolog notation for lists, and should not be confused with set comprehension). When i is a particular index, and t and t' are simply meta-variables, the equation $t'' = \{i=t|t'\}$ can be used to extract the value of the component indexed i from the record given by t'' , with t' being the rest of the record—without mentioning what other indices might be used in the record. A special dummy meta-variable written ‘...’ may be used in place of t' when the rest of the rule does not refer to t' ; an alternative is to use $t''.i$ to select the component indexed i from t'' .
- The above notation for records may also be used for constructing finite (partial) maps between sets, and for selecting values and the remaining maps. Two further operations are provided: t/t' is the map with t overriding t' ; and $\text{dom}(t)$ gives the domain of the map t (i.e. a set).
- A list with components t_1, \dots, t_n ($n \geq 0$) is written $[t_1, \dots, t_n]$.

One final point concerning rules: they can only be instantiated when all terms occurring in them have defined values. (Some of the operations used in terms may be partial, giving rise to the possibility of their applications having undefined values.) Note that applications of operations are never defined, and applications of predicates never hold, when any argument term has an undefined value.

3 Illustrative Examples

The combination of the rules given in this section provides an MSOS for a simple sub-language of Standard ML (SML). The concrete syntax of the described language is indicated by the grammar in Table 1 for declarations D , expressions E , operators O , and identifiers I . The example constructs have been chosen so as to illustrate various features of MSOS. The MSOS rules are formulated in terms of abstract syntax constructor operations whose correspondence to the concrete constructs is rather obvious.

A set of rules specifying the MSOS of a particular construct generally makes requirements on the components of labels and on various sets of values. For instance, it might be required for some construct that labels have a component

Table 1. Concrete syntax for the examples
$$\begin{aligned}
D &::= \text{val } I = E \mid \text{rec } D \\
E &::= \text{if } E \text{ then } E \text{ else } E \mid (E, E) \mid \\
&\quad E \ O \ E \mid \text{true} \mid \text{false} \mid [0 - 9]^+ \mid \\
&\quad I \mid \text{let } D \text{ in } E \text{ end} \mid \text{fn } I \Rightarrow E \mid E \ E \mid \\
&\quad \text{ref } E \mid E := E \mid ! E \mid E; E \mid \text{while } E \text{ do } E \mid \\
&\quad \text{raise } E \mid E \text{ handle } x \Rightarrow x \\
O &::= + \mid - \mid * \mid = \mid > \\
I &::= [a - z]^+
\end{aligned}$$

referred to as *bindings*, giving a map B from identifiers to values V , and that the set of values includes abstractions (representing functions). Such requirements are indicated informally below, in the interests of focussing attention on the rules themselves; the collected requirements imposed by all the constructs are listed at the end of the section.

All the rules given here are in the so-called *small-step* style. In fact MSOS does support the use of the big-step “natural semantics” style, as well as mixtures of the two styles. The general use of the small-step style will be motivated in Sect. 4.

3.1 Simple Expressions

The rules for the operational semantics of if-expressions in Table 2 require that expression values V include the truth-values *true* and *false*. The use of the variable X ranging over arbitrary labels in rule (2) reflects the fact that whatever information is processed by any step of E_1 , it is exactly the same as that processed by the corresponding step of $\text{if}(E_1, E_2, E_3)$. In contrast, the variable U in rule (3) and rule (4) ranges only over labels on unobservable steps, which are merely internal changes to configurations without any accompanying information processing. We henceforth suppress such single occurrences of U in rules, writing $\dots \longrightarrow \dots$ instead of $\dots -U \rightarrow \dots$.

Table 2. If-expressions: $\text{if } E \text{ then } E \text{ else } E$

$$E ::= \text{if}(E, E, E)$$

$$\frac{E_1 -X \rightarrow E'_1}{\text{if}(E_1, E_2, E_3) -X \rightarrow \text{if}(E'_1, E_2, E_3)} \quad (2)$$

$$\text{if}(\text{true}, E_2, E_3) -U \rightarrow E_2 \quad (3)$$

$$\text{if}(\text{false}, E_2, E_3) -U \rightarrow E_3 \quad (4)$$

Table 3. Pair-expressions: (E, E)

$$E ::= \text{pair}(E, E)$$

$$\frac{E_1 -X \rightarrow E'_1}{\text{pair}(E_1, E_2) -X \rightarrow \text{pair}(E'_1, E_2)} \quad (5)$$

$$\frac{E_2 -X \rightarrow E'_2}{\text{pair}(V_1, E_2) -X \rightarrow \text{pair}(V_1, E'_2)} \quad (6)$$

$$\text{pair}(V_1, V_2) \longrightarrow (V_1, V_2) \quad (7)$$

Clearly, rule (2) allows the computation of $\text{if}(E_1, E_2, E_3)$ to start with a computation of E_1 , and a computation of E_2 or E_3 can start only when that of E_1 terminates with computed value *true*, respectively *false*.

The rules in Table 3 require that the expressible values V are closed under formation of pairs (V_1, V_2) . Rule (6) allows the computation of E_2 to start only after E_1 has computed a value V_1 , thus reflecting sequential evaluation of the pair of expressions (following SML); replacing V_1 in that rule by E_1 would allow arbitrary interleaving of the steps of the computations of E_1 and E_2 .

The rules for binary operations shown in Table 4 make use of *operate*, which is defined in Table 5. For the latter, it is required that the expressible values V include both (integer) numbers N and the truth-values.

Table 6 is concerned with the evaluation of literal truth-values and numbers. Since the details of how to evaluate a sequence of (decimal) digits to the corresponding natural number are of no interest, let us assume that this is done already during the passage from concrete to abstract syntax, so that N is actually an integer.

Table 4. Binary operation expressions: $E O E$

$$E ::= \text{binary}(O, E, E)$$

$$\frac{E_1 -X \rightarrow E'_1}{\text{binary}(O, E_1, E_2) -X \rightarrow \text{binary}(O, E'_1, E_2)} \quad (8)$$

$$\frac{E_2 -X \rightarrow E'_2}{\text{binary}(O, V_1, E_2) -X \rightarrow \text{binary}(O, V_1, E'_2)} \quad (9)$$

$$\frac{\text{operate}(O, N_1, N_2) = V}{\text{binary}(O, N_1, N_2) \longrightarrow V} \quad (10)$$

Table 5. Binary operations: + | - | * | = | >
$$O ::= plus \mid minus \mid times \mid equals \mid greater$$

$$operate(plus, N_1, N_2) = N_1 + N_2 \quad (11)$$

$$operate(minus, N_1, N_2) = N_1 - N_2 \quad (12)$$

$$operate(times, N_1, N_2) = N_1 * N_2 \quad (13)$$

$$operate(equals, N_1, N_2) = true \text{ when } N_1 = N_2 \text{ else } false \quad (14)$$

$$operate(greater, N_1, N_2) = true \text{ when } N_1 > N_2 \text{ else } false \quad (15)$$

Table 6. Literal truth-values and numbers: true|false|[0–9]⁺

$$E ::= lit(true) \mid lit(false) \mid num(N)$$

$$lit(true) \longrightarrow true \quad (16)$$

$$lit(false) \longrightarrow false \quad (17)$$

$$num(N) \longrightarrow N \quad (18)$$

3.2 Bindings

So far, none of the rules have referred to any particular components of labels: the set of labels has been left completely open, and could even be just a singleton. Rule (19) in Table 7, however, requires that the labels have at least a component referred to as *bindings*. Let this component be “inherited”, and let its values be maps B from identifiers I to values V . (We do not bother here to make the usual notational distinction between the set of bindable—also known as denotable—values and the set V of expressible values, since for our illustrative constructs they happen to coincide.)

Rule (19) specifies that an identifier I evaluates to the value V to which it is bound by the bindings map B provided by the label U . Notice that if I is not in $dom(B)$, the rule simply cannot be applied, since one of its conditions isn’t satisfied. (Equations such as $B = U.bindings$ should be regarded formally as side-conditions, but it is notationally convenient to list them together with whatever premises the rule might have.)

Table 7. Value-identifiers: I

$$E ::= ide(I)$$

$$\frac{B = U.bindings, V = B.I}{ide(I) - U \rightarrow V} \quad (19)$$

Table 8. Let-expressions: `let D in E end`

$$E ::= \text{let}(D, E)$$

$$\frac{D -X \rightarrow D'}{\text{let}(D, E) -X \rightarrow \text{let}(D', E)} \quad (20)$$

$$\frac{\begin{array}{l} X = \{\text{bindings}=B_1|X'\}, B_2 = B/B_1, \\ X'' = \{\text{bindings}=B_2|X'\}, E -X'' \rightarrow E' \end{array}}{\text{let}(B, E) -X \rightarrow \text{let}(B, E')} \quad (21)$$

$$\text{let}(B, V) \longrightarrow V \quad (22)$$

This next illustration is particularly important. The rules in Table 8 show how block structure and nested scopes of declarations are specified in MSOS. Just as expressions E compute values V , a declaration D computes a binding map B , mapping the identifiers declared by D to values V .

The computation of $\text{let}(B, E)$ continues with that of E . The equations given as conditions in rule (21) ensure that the label X on a step for $\text{let}(B, E)$ has exactly the same components as the label X'' on a step for E , except of course for the *bindings* component, which is B_1 in X but B/B_1 (B overriding B_1) in X'' . Recall that an equation such as $X = \{\text{bindings}=B_1|X'\}$ both identifies the *bindings* component as B_1 and the record consisting of all the other fields of X as X' .

If the computation of E finally computes a value V , that is also the value computed by the let-expression, as specified by rule (22).

The rules for (non-recursive) value declarations are given in Table 9, and are completely straightforward. Closely related to value declarations are value abstractions and (call by value) applications, which are specified in Tables 10 and 11.

The value computed by an abstraction is a so-called closure, which is conveniently represented by using a let-expression to attach the current bindings B to the abstraction itself. If we didn't already have the constructor for let-expressions available, we would have to define it as auxiliary notation (or provide some other operation allowing bindings to be attached to abstractions). The auxiliary oper-

Table 9. Value-declarations: `val I = E`

$$D ::= \text{value}(I, E)$$

$$\frac{E -X \rightarrow E'}{\text{value}(I, E) -X \rightarrow \text{value}(I, E')} \quad (23)$$

$$\text{value}(I, V) \longrightarrow \{I=V\} \quad (24)$$

Table 10. Value-abstractions: $\text{fn } I \Rightarrow E$

$$\begin{array}{c}
E ::= \text{abstraction}(I, E) \\
U = \{\text{bindings}=B|\dots\} \\
\hline
\text{abstraction}(I, E) -U \rightarrow \text{abs}(\text{let}(B, \text{abstraction}(I, E)))
\end{array} \tag{25}$$

ation $\text{abs}(E)$ constructs a value from an arbitrary expression E —we cannot use the expression E itself as a value, since values should always be final states for computations.

Once E_1 has been evaluated to an abstraction value V_1 , and E_2 to an argument value V_2 , rule (28) replaces $\text{application}(V_1, V_2)$ by the appropriate expression, and the computation may continue. Assuming that all the identifiers free in E are bound by B , the application-time bindings provided by U are not used (directly) in that computation: the rules have provided static scopes for bindings.

The constructs that we have specified so far allow the declaration of (call by value) functions, using a combination of value-declarations and value-abstractions, but they do not (directly) support recursive function declarations. The construct $\text{recursive}(D)$ specified in Table 12 has the effect of making ordinary function declarations D recursive (it could easily be extended to mutually-recursive declarations).

The auxiliary operation $\text{unfold}(D, B)$ is defined by the (equational) rule (31). It returns a binding map where D has been inserted at the appropriate place in the closure, using a further level of let-expression. (It isn't intended for use on bindings of non-function values such as numbers and pairs, but for completeness, rule (32) specifies that it would have no effect on them.) Rule (33) specifies the result of an application when the abstraction incorporates a recursive declaration $\text{recursive}(B_1)$; the actual unfolding of the recursive declaration is left to the (first step of) the subsequent computation.

Table 11. Abstraction-applications: $E E$

$$\begin{array}{c}
E ::= \text{application}(E, E) \\
E_1 -X \rightarrow E'_1 \\
\hline
\text{application}(E_1, E_2) -X \rightarrow \text{application}(E'_1, E_2)
\end{array} \tag{26}$$

$$\begin{array}{c}
E_2 -X \rightarrow E'_2 \\
\hline
\text{application}(V_1, E_2) -X \rightarrow \text{application}(V_1, E'_2)
\end{array} \tag{27}$$

$$\begin{array}{c}
V_1 = \text{abs}(\text{let}(B, \text{abstraction}(I, E))) \\
\hline
\text{application}(V_1, V_2) \longrightarrow \text{let}(B, \text{let}(\{I=V_2\}, E))
\end{array} \tag{28}$$

Table 12. Recursive declarations: `rec D`

$$D ::= \text{recursive}(D)$$

$$\frac{D -X \rightarrow D'}{\text{recursive}(D) -X \rightarrow \text{recursive}(D')} \quad (29)$$

$$\frac{B' = \text{unfold}(\text{recursive}(B), B)}{\text{recursive}(B) \longrightarrow B'} \quad (30)$$

$$\frac{V = \text{abs}(\text{let}(B, E)), V' = \text{abs}(\text{let}(B, \text{let}(\text{recursive}(B_1), E)))}{\text{unfold}(\text{recursive}(B_1), \{I=V\}) = \{I=V'\}} \quad (31)$$

$$\frac{V \neq \text{abs}(\dots)}{\text{unfold}(\text{recursive}(B_1), \{I=V\}) = \{I=V\}} \quad (32)$$

$$\frac{V_1 = \text{abs}(\text{let}(B, \text{let}(\text{recursive}(B_1), \text{abstraction}(I, E)))}{\text{application}(V_1, V_2) \longrightarrow \text{let}(B, \text{let}(\text{recursive}(B_1), \text{let}(\{I=V_2\}, E)))} \quad (33)$$

Rules for declarations analogous to those illustrated here could be given in conventional SOS, using transitions of the form $B \vdash E \longrightarrow E'$ (after defining a suitable notion of composition for such transitions). The main pragmatic advantages of our MSOS rules will become apparent only in the next sections, where we shall allow expressions to have side-effects and to raise exceptions, without any reformulation of the rules given so far.

3.3 Stores

The imperative constructs described in this section are taken almost unchanged from SML, and involve so-called references to values. References correspond to simple variables, and can be created, updated, and dereferenced. The set of created references, together with the values to which they refer, is represented by a store S mapping locations L to values V . A location is itself a value, and can be stored, as well as bound to identifiers.

Some of the rules given here require that labels X have not only a *store* component, giving the store at the beginning of a transition labelled X , but also a *store'* component, giving the (possibly different) store at the end of the transition. When a transition labelled X_1 is followed immediately by a transition labelled X_2 in a computation, the *store'* component of X_1 has to be the same as the *store* component of X_2 . Moreover, the *store* and *store'* components of an unobservable transition labelled U have to be the same as well.

Table 13 gives the rules for reference creation, which is combined with initialization. L in rule (35) can be any location that is not already in use in S . Notice the use of the variable U , which ensures that the extension of S by the association of L with V is the only observable effect of the transition labelled X .

The rules for assignment given in Table 14 correspond closely to those in Table 13, except that the assignment computes the null value $()$. Table 15 gives

Table 13. Reference-expressions: $\text{ref } E$

$$\begin{array}{c}
E ::= \text{reference}(E) \\
\frac{E - X \rightarrow E'}{\text{reference}(E) - X \rightarrow \text{reference}(E')}
\end{array} \tag{34}$$

$$\frac{X = \{\text{store}=S, \text{store}'=S'|U\}, L \notin \text{dom}(S), S' = \{L=V|S\}}{\text{reference}(V) - X \rightarrow L} \tag{35}$$

Table 14. Assignment-expressions: $E := E$

$$\begin{array}{c}
E ::= \text{assignment}(E, E) \\
\frac{\text{assignment}(E_1, E_2) - X \rightarrow \text{assignment}(E'_1, E_2)}{E_1 - X \rightarrow E'_1}
\end{array} \tag{36}$$

$$\frac{\text{assignment}(L_1, E_2) - X \rightarrow \text{assignment}(L_1, E'_2)}{E_2 - X \rightarrow E'_2} \tag{37}$$

$$\frac{X = \{\text{store}=S, \text{store}'=S'|U\}, L_1 \in \text{dom}(S), S' = \{L_1=V_2|S\}}{\text{assignment}(L_1, V_2) - X \rightarrow ()} \tag{38}$$

the rules for (explicit) dereferencing; and Table 16 describes expression sequencing, where the value of the first expression is simply discarded. The rule for while-expressions in Table 17 is standard in (small-step) SOS, involving both an if-expression and a sequence-expression (a direct description not involving other constructs seems to be elusive).

Despite the fact that the illustrated imperative programming constructs allow expressions to have (side-)effects, no reformulation at all is required for the rules given in the preceding section: they remain well-formed and continue to describe the intended semantics. This is in marked contrast with conventional SOS, where stores would be added explicitly to configurations, requiring all transitions $B \vdash E \longrightarrow E'$ to be reformulated as something like $B \vdash (E, S) \longrightarrow$

Table 15. Dereferencing-expressions: $! E$

$$\begin{array}{c}
E ::= \text{dereference}(E) \\
\frac{E - X \rightarrow E'}{\text{dereference}(E) - X \rightarrow \text{dereference}(E')}
\end{array} \tag{39}$$

$$\frac{U = \{\text{store}=S|\dots\}, S = \{L=V|\dots\}}{\text{dereference}(L) - U \rightarrow V} \tag{40}$$

Table 16. Sequence-expressions: $E; E$

$$E ::= \text{sequence}(E, E)$$

$$\frac{E_1 -X \rightarrow E'_1}{\text{sequence}(E_1, E_2) -X \rightarrow \text{sequence}(E'_1, E_2)} \quad (41)$$

$$\text{sequence}(V_1, E_2) \longrightarrow E_2 \quad (42)$$

Table 17. While-expressions: `while` E `do` E

$$E ::= \text{while}(E_1, E_2)$$

$$\frac{\text{while}(E_1, E_2) \longrightarrow}{\text{if}(E_1, \text{sequence}(E_2, \text{while}(E_1, E_2)), ())} \quad (43)$$

(E', S') . Actually, the “store convention” adopted in the Definition of SML [10] would avoid the need for such a reformulation, but it has a rather informal character; the incorporation of stores as components of labels in MSOS achieves a similar result, completely formally.

3.4 Exceptions

The description of exception raising and handling in conventional SOS usually involves giving further rules for all the *other* constructs in the language being described, allowing exceptions raised in any component to be “propagated” upwards until an exception handler is reached. Such rules are quite tedious, and undermine modularity. The “exception convention” adopted in the Definition of SML [10] allows the exception propagation rules to be left implicit, but has the disadvantage that the set of presented rules has to be expanded considerably to make the propagation rules explicit before they can be used for deriving computations, etc.

Fortunately, a technique recently proposed by Klin (a PhD student at BRICS in Aarhus) avoids the need for exception propagation rules altogether, and gives excellent modularity. The key idea is illustrated in Table 18 and Table 19: labels on transitions are required to have a further component that simply indicates whether an exception is being raised by the transition or not, and if so, gives the value associated with the raised exception. An exception handler monitors the label of every step of the computation of its body, checking the extra component.

To focus attention on the main features of the new technique, and for brevity, only a simplified indiscriminate exception handler is described here; it is however straightforward to describe SML’s exception handlers, which involve pattern-matching.

Table 18. Exception-raising: `raise E`

$$E ::= \text{raise}(E)$$

$$\frac{E -X \rightarrow E'}{\text{raise}(E) -X \rightarrow \text{raise}(E')} \quad (44)$$

$$\frac{X = \{\text{raising}=[V]|U\}}{\text{raise}(V) -X \rightarrow ()} \quad (45)$$

Table 19. Exception-handling: `E handle x=>x`

$$E ::= \text{handle}(E)$$

$$\frac{E -X_1 \rightarrow E', X_1 = \{\text{raising}=[V]|X'\}, X = \{\text{raising}=[]|X'\}}{\text{handle}(E) -X \rightarrow V} \quad (46)$$

$$\frac{E -X \rightarrow E', X = \{\text{raising}=[]|\dots\}}{\text{handle}(E) -X \rightarrow \text{handle}(E')} \quad (47)$$

$$\text{handle}(V) \longrightarrow V \quad (48)$$

Let us use a list $[V]$ with the single component V to indicate the raising of an exception with value V , and the empty list $[]$ to indicate the absence of an exception. Transitions that raise exceptions are always observable.

In Table 18, rule (45) merely sets the *raising* component of X to the list $[V]$. When the raised exception occurs during the computation of $\text{handle}(E)$, rule (46) detects the exception and abandons the computation of E , giving V as the computed value. The *raising* component of the label X is set to the empty list, to reflect that the exception has now been handled at this level. The complementary rule (47) deals with a normal step of the computation of E where no exception is raised.

Suppose however that an exception is raised with no enclosing handler. Then the step indicated in rule (45) is completed, and the computation continues normally, as if the raise-expression had merely computed the null value $()$, although the raising of the exception is observable from the label of that step. To get the intended effect that an unhandled exception should stop the entire program, the entire program should always be enclosed in an extra *handle*.

The technique illustrated here is not restricted to MSOS: it could be used in conventional (small-step) SOS as well. However, in a conventional SOS of a programming language, transitions are usually unlabelled, and the amount of reformulation that would be required to add labels may be a considerable disincentive to exploiting the new technique. In MSOS, transitions are always labelled, and adding a new component to labels doesn't require any reformulation of rules.

3.5 Interaction

Lack of space precludes illustration here of the MSOS of constructs for interactive input and output, process spawning, message passing, and synchronization. An MSOS for the core of Concurrent ML has been given (using a more abstract notation) in a previous paper [17]: the rules are quite similar in style to the conventional SOS rules for concurrent processes, where configurations are generally syntactic, and transitions are labelled, just as in MSOS. What is remarkable with the MSOS rules is that allowing expressions to spawn processes and communicate with other processes requires no reformulation at all of rules given for purely functional expressions.

3.6 Summary of Requirements

The requirements made by the rules given in this section are summarized in Table 20.

Table 20. Summary of requirements

Values:	$V ::= true \mid false \mid N \mid (V, V) \mid abs(E) \mid L \mid ()$ $B ::= Map(I, V)$ $S ::= Map(L, V)$
Labels:	$X ::= \{ \text{inherited } bindings : B;$ $\text{variable } store, store' : S;$ $\text{observable } raising : List(V) \}$

4 Pragmatic Aspects

The illustrative examples given in the preceding section should have given a reasonable impression of how MSOS works. Let us now consider some important pragmatic aspects of MSOS, concerning modularity, tool support, and the choice between big-step and small-step rules.

4.1 Modularity

The main pragmatic advantage of MSOS over conventional SOS is that the rules describing the intended semantics for each programming construct can be given *definitively*, once and for all. When gradually building up a language, it is never necessary to go back and reformulate previously-given rules to accommodate

the addition of new constructs. Moreover, when different languages include the same constructs (e.g. SML and Java both have if-expressions), the MSOS rules describing the semantics of the common constructs may be reused verbatim.

Such features justify the use of the word “modular” in connection with the MSOS framework. The benefits of this kind of modularity are especially apparent when using MSOS to teach operational semantics, and one may hope that they might even encourage language designers to use formal semantics to record their decisions during a design process.

4.2 Tool Support

Another pragmatic aspect that is particularly important for applications of formal semantics in teaching and language design is tool support [6]. This is needed both for developing and browsing (large) semantic descriptions, as well as for validating descriptions by running programs according to their specified semantics. Some tool support for MSOS has already been developed by Braga et al. [3,4] using the Maude system for Rewriting Logic, but much remains to be done.

In fact it is quite easy to write MSOS rules directly in Prolog. Some examples are given in Table 21, and a Prolog program implementing all the MSOS rules given in this paper is available at <http://www.brics.dk/~pdm/AMAST-02/>.² Both records and finite maps are conveniently represented in Prolog as lists of equations, and the Prolog predicates `member(M,L)` and `select(M,L1,L2)` are used to implement conditions concerning labels, bindings, stores, etc. Unary predicates are used to check that variables range only over the intended sets, e.g., `unobs(U)` ensures that `U` is indeed an unobservable label, and `val(V)` checks that `V` is a computed values.

A definite clause grammar for the described language allows programs to be written in concrete syntax, and mapped directly to the abstract syntax constructors used in the rules. The user may determine some of the components of the label on the first transition, and select which components are to be shown at the end of the computation. When the rules are non-deterministic (e.g. describing interleaving or concurrency) the user may investigate successive solutions. The efficiency of modern Prolog implementations (the author uses SWI-Prolog, which is freely available from <http://www.swi-prolog.org/> for most popular platforms) is such that running small test programs using the clauses corresponding to their MSOS rules takes only a few seconds on a typical lap-top.

The Prolog clauses have just as good modularity as the MSOS rules. Tracers and debuggers (SWI-Prolog provides one with an efficient graphical user interface) allow the user to follow changes to configurations at the top level, as well as in-depth derivations of individual steps.

At the time of writing this paper, it remains to be seen whether students will find the Prolog implementation useful for getting a good operational understanding of the MSOS rules, and whether they will find it easy to extend it with implementations of their own rules.

² The author has had little experience of programming in Prolog, and would appreciate suggestions for improvements to the code.

Table 21. Examples of MSOS rules in Prolog

```

pair(E1,E2) ---X---> pair(E1_,E2) :-
    E1 ---X---> E1_.
pair(EV1,E2) ---X---> pair(EV1,E2_) :-
    val(EV1),
    E2 ---X---> E2_.
pair(EV1,EV2) ---U---> (EV1,EV2) :-
    val(EV1), val(EV2), unobs(U).

assignment(E1,E2) ---X---> assignment(E1_,E2) :-
    E1 ---X---> E1_.
assignment(L1,E2) ---X---> assignment(L1,E2_) :-
    val(L1), E2 ---X---> E2_.
assignment(L1,V2) ---X---> null :-
    val(L1), val(V2),
    select(store=S,X,X_), select(L1=_,S,S_),
    select(store_= [L1=V2|S_],X_,U), unobs(U).

```

4.3 Why not Big Steps?

All the MSOS rules illustrated in Sect. 3 are small-step rules: each step that a construct can take depends on at most one of its components taking a corresponding step. However, MSOS does allow big-step rules as well, where a step represents an entire sub-computation, going straight to a final state—often depending on all its components taking corresponding steps.

For example, consider the big-step rule for pair-expressions given in Table 22. Label composition, written $X_1;X_2$, has to be used explicitly in big-step rules (recall that in small-step MSOS, computations require adjacent labels to be composable, but composition is not needed in the rules themselves). The specified order of composition of labels indicates that the corresponding computation steps may be taken in the same order.

Given that MSOS supports both small-step and big-step rules, which should one prefer? Changing from the one style of rule to the other requires major reformulation, so it is important to make a good choice and stick to it.

Table 22. Big-step pair-expressions: (E, E)

$$\begin{aligned}
 E &::= \text{pair}(E, E) \\
 \frac{E_1 -X_1\rightarrow V_1, E_2 -X_2\rightarrow V_2}{\text{pair}(E_1, E_2) -X_1;X_2\rightarrow (V_1, V_2)} & \quad (49)
 \end{aligned}$$

In his Aarhus lecture notes on SOS [21], Plotkin generally used small-step rules—although by employing the transitive closure of the transition relation in the conditions of some rules, he obtained the effect of mixing the two styles of rules (e.g. a single small step for a statement depended on complete sequences of small steps for its component expressions). Moreover, much of the work on semantics of concurrent processes (CCS, CSP, etc.) is based on small-step rules.

On the other hand, Kahn [8] has advocated exclusive use of big-step rules, and this style was adopted by Milner et al. for the Definition of SML [10]. The pure big-step style has also been widely used in specifying type systems and static semantics.

This author’s view is that big-step rules should be used *only* for constructs whose computations always terminate normally: no divergence, no raising of exceptions. This includes literal constants and types, but excludes almost all other programming constructs. (As usual, we are focussing here on dynamic semantics; for static semantics, the recommendation would be to use big-step rules for all constructs.) The reason for the restriction to terminating computations is that big-step semantics is usually based on finite derivations, and non-terminating computations get completely ignored. Regarding exceptions, big-step rules appear to be inherently non-modular: adding exceptions to the described language requires adding rules specifying propagation of exceptions. (The novel technique illustrated in Sect. 3.4 works only for small-step rules.)

It seems to be a widely-held belief that the description of interleaving constructs actually requires small-step rules. Suppose however that one makes a slight generalization to MSOS, allowing not only single records X but also arbitrary (finite) sequences of records $X_1 \dots X_n$ as labels on transitions. It turns out that big-step rules for interleaving can then be given quite straightforwardly, by allowing the sequences in the labels to be arbitrarily interleaved (i.e., “shuffled”), and restricting the labels on transitions for the entire program to be composable sequences. This technique appears to depend entirely on the use of MSOS: in SOS, the inclusion of auxiliary entities in states makes it awkward (if not impossible) to give an analogous treatment of interleaving. In view of the arguments given above in favour of small-step rules, however, the development of techniques for use in big-step rules is not so interesting from a pragmatical point of view.

5 Conclusion

This paper has given an overview of the foundations of MSOS, and has illustrated the use of MSOS to describe some simple functional and imperative programming constructs. It has also presented a novel technique for the modular description of exception-handling. Finally, it has discussed particular pragmatic aspects such as modularity, tool support, and the choice between small- and big-step rules.

The development of MSOS was originally stimulated by the author’s dissatisfaction with the lack of modularity in his own SOS for Action Notation, the semantic notation used in Action Semantics [12]. An MSOS for Action Notation has been given [15] using CASL, the Common Algebraic Specification Language,

for meta-notation. The modularity of this MSOS description was particularly useful during the redesign of Action Notation [18]. An MSOS of ML concurrency primitives has been provided [15] to facilitate a comparison of MSOS with SOS and evaluation-context (reduction) semantics for the same language. MSOS has not so far been used for giving a complete description of a major programming language, and it remains to be seen whether any pragmatic problems would arise when scaling up.

The continuing development of MSOS is motivated by the aim of optimizing the pragmatic aspects of the structural approach to operational semantics, partly for use when teaching semantics [19]. Several topics are in need of further work:

- For truly definitive descriptions of individual language constructs, a universal language-independent abstract syntax has to be established.
- A library of MSOS modules (with accompanying Prolog implementations) should be made available.
- Existing systems supporting animation and validation of SOS [5] might be adapted to support MSOS.
- A type-checker for MSOS should be provided.
- The study of bisimulation and other equivalences for MSOS, initiated in [13,14], should be continued.

Readers who might be interested in contributing to the development of MSOS by working on these or other topics are requested to let the author know.

Acknowledgements The author is grateful to the organizers of AMAST 2002 for the invitation to talk on the topic of Modular SOS. Olivier Danvy and Jørgen Iversen suggested improvements to drafts of this paper; remaining infelicities are the sole responsibility of the author. Bartek Klin suggested the novel technique for dealing with exceptions. Leonardo de Moura suggested implementing MSOS directly in Prolog.

References

1. L. Aceto, W. J. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 1: Basic Theory, pages 197–292. Elsevier, 2001. 21, 24
2. D. Berry, R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, pages 119–129. ACM, 1992. 21
3. C. de O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rua Marquês de São Vicente 255, Gávea, Rio de Janeiro, RJ, Brazil, September 2001. <http://www.inf.puc-rio.br/~cbraga>. 36
4. C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST 2000*, volume 1816 of *LNCS*, pages 407–421. Springer-Verlag, 2000. 36
5. P. H. Hartel. LETOS - a lightweight execution tool for operational semantics. *Software – Practice and Experience*, 29(15):1379–1416, Sept. 1999. 39

6. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, Mar. 2000. 36
7. M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990. 21
8. G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987. 23, 38
9. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990. 21
10. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 21, 33, 38
11. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990. 21
12. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992. 38
13. P. D. Mosses. Foundations of modular SOS. Research Series RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. <http://www.brics.dk/RS/99/54/>; full version of [14]. 21, 22, 24, 39
14. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at <http://www.brics.dk/RS/99/54/>. 21, 22, 24, 39, 40
15. P. D. Mosses. A modular SOS for Action Notation. Research Series RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. <http://www.brics.dk/RS/99/56/>. Full version of [16]. 38, 39
16. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at <http://www.brics.dk/RS/99/56/>. 40
17. P. D. Mosses. A modular SOS for ML concurrency primitives. Research Series RS-99-57, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. <http://www.brics.dk/RS/99/57/>. 21, 35
18. P. D. Mosses. AN-2: Revised action notation—syntax and semantics. Available at <http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/>, Oct. 2000. 39
19. P. D. Mosses. Fundamental concepts and formal semantics of programming languages. Lecture Notes. Version 0.2, available from <http://www.brics.dk/~pdm/>, Sept. 2002. 22, 39
20. H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, Chichester, UK, 1992. 21
21. G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Dept. of Computer Science, Univ. of Aarhus, 1981. 21, 23, 38
22. K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995. 21
23. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. 21

Tool-Assisted Specification and Verification of the JavaCard Platform

Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa*

INRIA Sophia-Antipolis, France
{gilles.barthe,pierre.courtieu}@inria.fr
{guillaume.dufay,simao.desousa}@inria.fr

Abstract. Bytecode verification is one of the key security functions of the JavaCard architecture. Its correctness is often cast relatively to a defensive virtual machine that performs checks at run-time, and an offensive one that does not, and can be summarized as stating that the two machines coincide on programs that pass bytecode verification. We review the process of establishing such a correctness statement in a proof assistant, and focus in particular on the problem of automating the construction of an offensive virtual machine and a bytecode verifier from a defensive machine.

1 Introduction

Smartcards are small devices designed to store and process confidential data, and can act as enabling tokens to provide secure access to applications and services. They are increasingly being used by commercial and governmental organizations as bankcards, e-purses, SIM cards in mobile phones, e-IDs, *etc.*, and are expected to play a key role in enforcing trust and confidence in e-payment and e-government [22].

Open platform smartcards are new generation smartcards with increased flexibility. Such smartcards:

- integrate on-board a virtual machine that abstracts away from any hardware and operating system specifics so that smartcard applications, or *applets*, can be programmed in a high-level language (before being compiled and loaded onto the card);
- are multi-applications, in that several applets can coexist and communicate (securely) on the same card, and support post-issuance, in that new applets may be loaded onto already deployed smartcards.

The flexibility of open platform smartcards is at the same time a major asset and a major obstacle to their deployment. On the one hand, writing applets

* On leave from University of Beira Interior-Portugal, and partially supported by the Portuguese research grant sfrh/bd/790/2000.

in a high-level language reduces the cost and time to market new applications, and the possibility of running several applets on a single card opens the way for novel applications. On the other hand, such smartcards introduce the possibility to load malicious applets that exploit weaknesses of the platform to read, modify or erase confidential data of other applets, or simply to block the card.

JavaCard and the JavaCard Platform. JavaCard [26] is the standard programming language for multi-applications smartcards. Launched in 1996 by Sun, JavaCard brings to the resource-constrained environment of open platform smartcards many benefits of Java: object-orientation, safety through typing, portability. Here is a brief description of JavaCard and of its differences with Java:

- JavaCard applets are written in a subset of the Java language that omits some items such as large datatypes, arrays of arrays, finalization or threads. Furthermore, JavaCard applets use the JavaCard API, which defines and constrains the interface between an applet and its environment on the card;
- JavaCard applets are compiled down to class files, and then converted to the *CAP file* format. Conversion involves a number of optimizations suggested by smartcard constraints, e.g. names are replaced by tokens and class files are merged together on a package basis;
- compiled JavaCard applets go through a bytecode verifier BCV which checks that applets are correctly formed and well-typed, and that they do not attempt to perform malicious operations during their execution, see [33]. Until recently, applets were verified off-card and, in case of a successful verification, signed and loaded on-card. However, on-card bytecode verification, which circumscribes the trusted computing base to the smartcard, should come to play a prominent role in the future [13,14,34];
- JavaCard applets are executed by the JavaCard Runtime Environment JCRE. The JCRE contains the JavaCard Virtual Machine JCVM, provides support for the JavaCard API and invokes the services of a native layer to perform low-level tasks such as communication management. The JCVM is a stack-based abstract machine as the Java Virtual Machine JVM but some features, like the dynamic class loading mechanism of the JVM, are unsupported;
- JavaCard claims some features that are not present, at least not in the same form, in Java: for example, JavaCard 2.1. introduced firewalls to ensure applet isolation, and JavaCard 2.2. introduced logical channels and a simplified form of remote method invocation.

Further technical information on JavaCard may be found in publicly available specifications, white papers and books [16,43,44,45,46].

Formal methods for smartcards. It is commonly agreed that a high degree of reliability is required if smartcards are to gain a widespread acceptance as trusted computing devices, and that the highest degree of reliability can only be achieved

by the use of formal methods to formally design, test and verify smartcard platforms and applications. This view is also present in the Common Criteria [21], that define a unified scheme for evaluating the security of IT products, and impose that evaluations at the highest levels, i.e. from EAL5 to EAL7, rely heavily on formal methods.

There are many facets to formal verification for smartcards, each of which comes with its own objectives and techniques. For example:

- program verification at source level may help developers to ensure that their applets are correct. One definite advantage of such a scenario is that the developer (or an expert in formal methods working with him) can provide a formal specification of the program, and then check/prove that the applet is correct with respect to its specification. There is a wide range of formalisms to express such specifications, such as predicate, temporal and dynamic logics or interface specification languages [27]. The corresponding verification techniques include run-time checking and unit testing [17], static checking [32], model-checking [19], and interactive theorem proving [9];
- program verification at bytecode level may help to check that it is safe to load and execute an applet whose origin is unknown or doubtful. As one cannot expect code consumers to engage in program verification, it is important that this kind of verification uses automatic techniques such as static analyses and model-checking [11,39], or relies on a scheme which delegates the verification tasks to the code producer [36].

Platform verification, which is the focus of our work, aims at establishing that a platform for smartcard is correct in the sense that it adheres to some acceptable security policy. Such a security policy is usually formulated as a set of constraints like “a reference cannot be cast to an integer”, or “the virtual machine cannot jump to an arbitrary address”, and in the case of the JavaCard platform should include constraints such as type safety and applet isolation. The purpose of platform verification is to prove formally that these constraints are enforced by well-identified security functions. One important such security function is bytecode verification (it guarantees properties such as type correctness, code containment, proper object initialization, absence of stack overflows); proving its correctness is a prerequisite for Common Criteria evaluations at highest levels, and our starting point.

The correctness of bytecode verification is stated as a correspondence between two flavors of virtual machines [15,30,40]:

- a defensive JCVM, which manipulates typed values and performs structural and type checking at run-time. The defensive JCVM is close to Sun’s specification, but is inefficient in terms of execution speed and memory consumption;
- an offensive JCVM, which manipulates untyped values and relies on successful bytecode verification to eliminate structural and type checking at run-time. The offensive JCVM is close to an implementation;

The correctness statement can be summarised by the slogan: “the offensive and defensive JCVMs coincide on programs that pass bytecode verification”. Formulating precisely and proving the statement requires:

- modeling the defensive and offensive JCVMs and formalizing the notion of correspondence between the two virtual machines in terms of an abstraction function α_{do} that maps states of the defensive machine to states of the offensive machine;
- showing that both machines coincide on programs whose execution on the defensive JCVM does not raise a type error;
- modeling the BCV as a dataflow analysis of an abstract JCVM that only manipulates types and showing that the defensive JCVM does not raise a type error for programs that pass bytecode verification.

In the context of Common Criteria evaluations at the highest levels, such models and proofs are required to be formal and are best developed with the help of proof assistants.

CertiCartes [7,8] is an in-depth feasibility in proving the correctness of bytecode verification for JavaCard 2.1, using the proof assistant Coq [18]. Currently CertiCartes contains executable specifications of the three JCVMs (defensive, offensive and abstract) and of the BCV, and a proof of the correctness of the BCV. The Coq development is structured in two separate modules:

1. the first module JCVM includes the construction of three virtual machines and their cross-validation. This development represents the bulk of the effort behind CertiCartes;
2. the second module BCV includes the construction and validation of the bytecode verifier. This development is largely independent from the definition of the virtual machines, which are taken as parameters in the construction of the BCV.

An external evaluation by Gemplus Research Labs [29] suggests that CertiCartes models the JavaCard platform at a reasonable level of detail, and proposes some concrete steps to make CertiCartes amenable to certification. It is not an avenue that we are intent on pursuing, in particular because the amount of work involved is considerable: [24] reports that six man years were necessary to carry a formalization of JavaCard 2.1 in full detail.

Assessment. Our experiment with CertiCartes corroborates the common views that machine checked verification of standard programming languages is becoming viable but sorely lacks of accurate tool support for specifying and proving in the large. What seems more interesting of CertiCartes is that it validates an original methodology which is well-suited for reasoning about typed low-level languages, and which can be applied to novel type systems for Java(Card) [4,23,31,42]. Furthermore, our work suggested that it is possible to develop appropriate tool support for this methodology, and in particular for the construction of the offensive and abstract JCVM from the defensive one, and for their cross-validation.

Jakarta [6] is an experimental toolset for specifying and reasoning about the JavaCard Platform. The overall objective is to provide:

- a front-end, independent from a particular proof assistant, for describing virtual machines and subjecting such descriptions to transformations such as abstractions, refinements and optimizations;
- a compiler that translates these specifications in a format usable by proof assistants, and mechanisms for proving the correctness of these transformations in such proof assistants.

Our current focus is to provide tool support for the construction of an offensive and abstract JCVM from a description of a defensive JCVM, and for their cross-validation in a proof assistant. It may appear to be an over-specific goal but the defensive JCVM has many instantiations that are determined by the property to be enforced statically through typing. These properties typically relate to object initialization, information flow, resource control, locks, *etc*—we shall return to this point in the conclusion. Nevertheless type safety is the prime example of property to be enforced, and the running example of this paper.

In what follows, we shall: describe the methodology behind CertiCartes; explain how Jakarta has been used for extracting the offensive and abstract JCVMs from the defensive one; summarize ongoing work on automating the cross-validation between the three virtual machines.

2 An Overview of CertiCartes

As indicated in the introduction, CertiCartes is structured in two separate modules: a first module JCVM, which includes the construction of three virtual machines and their cross-validation, and a second module BCV, which includes the construction and validation of the BCV from the abstract JCVM. The next paragraphs briefly describe the contents of CertiCartes; [7,8] provide further details.

2.1 Module JCVM

Modeling programs. Programs are formalized in a neutral mathematical style based on (first-order, non-dependent) datatypes and record types, and the corresponding machinery: case analysis and structural recursion for datatypes, formation and selection for record types. For example, programs are represented by the record type:

```
Record jcprogram := {
  interfaces : (list Interface);
  classes    : (list Class);
  methods    : (list Method)
}.
```

where the types `Interface`, `Class` and `Method` are themselves defined as record types. For simplicity, we only deal with closed programs hence (relevant parts of) the packages `java.lang` and `javacard.framework` are an integral part of programs.

Modeling memory. Memory is modeled in a similar way. For example, states are formalized as a record consisting of the heap (containing the objects created during execution), the static fields image (containing static fields of classes) and a stack of frames (environments for executing methods). Formally:

```
Record state := {
  hp : heap;
  sh : sheap;
  st : stack
}.
```

In order to account for abrupt termination (that may arise because of uncaught exceptions or because the program being executed is ill-formed), we also introduce a type of return states, defined as a sum type:

```
Inductive rstate :=
  Normal   : state → rstate |
  Abnormal : exception → state → rstate.
```

The definition of (return) states and of all components of the memory model are parameterized by a generic type of values, which we leave unspecified. The memory model will later be instantiated to the defensive, offensive and abstract JCVMs. The following table gives the notion of value attached to each JCVM:

Virtual Machine	Defensive	Offensive	Abstract
Value	(Type, Number)	Number	Type

Modeling the defensive JCVM. First, we instantiate the memory model with a notion of typed value. We thus obtain a type of defensive states `dstate` and return defensive states `rdstate`.

Then, we model the defensive semantics of each JavaCard bytecode `b` as a function `dexec_b`: `dstate` \rightarrow `rdstate`. Typically, the function `dexec_b` extracts values from the state, performs type verification on these values, and extends/updates the state with the results of executing the bytecode.

Finally, one-step defensive execution is modeled as a function `dexec`: `dstate` \rightarrow `rdstate` which inspects the state to extract the JavaCard bytecode `b` to be executed and then calls the corresponding function `dexec_b`.

Modeling the offensive JCVM. First, we instantiate the memory model with a notion of untyped value so as to obtain a type of offensive states `ostate` and return offensive states `rostate`. Then, one-step offensive execution `oexec`: `ostate` \rightarrow `rostate` is modeled in the same way as one-step defensive execution, but all verifications and operations related to typing are removed.

Modeling the abstract JCVM. The abstract JCVM that is used for bytecode verification operates at the level of types. Formally, we define the type `astate` of abstract states as a record type with two fields: an abstract frame and an abstract static fields image (which are obtained by instantiating values to types in the generic notions of frame and of static fields image). The memory model is thus simplified because: (1) the abstract JCVM operates on a method per method basis, so only one frame is needed; (2) the type of objects is stored in the operand stack so the heap is not needed.

One-step abstract execution is non-deterministic because branching instructions lead to different program points and hence different states. We use a list to collect all return states. Therefore one-step abstract execution is modeled as a function `aexec: astate → (list rstate)`, where `astate` is the type of return abstract states.

Cross-validation of the JCVMs is a prerequisite for establishing the correctness of the BCV, and involves showing that the offensive and defensive JCVMs coincide on programs that do not raise a type error when executed on the defensive JCVM; and that every program that raises a type error when executed on the defensive JCVM also raises a typing error when executed with the abstract JCVM. These statements are nicely expressed in terms of abstract interpretation. Indeed, the offensive and abstract JCVMs can be viewed as abstractions of the defensive JCVM. Likewise, the correctness statements can be viewed as expressing that the offensive and abstract JCVMs are sound abstractions of the defensive JCVM. Formally, we define two abstractions functions:

```
alpha_do : dstate → ostate
alpha_da : dstate → astate
```

together with their lifting to return states:

```
alpha_do_rs : rdstate → rostate
alpha_da_rs : rdstate → rastate
```

and prove that, under suitable constraints, the diagrams of Figures 1 and 2 commute (where the arrow on the right-hand side of Figure 2 means that the abstraction of the return defensive state is, up to subtyping, an element of the list of return abstract states).

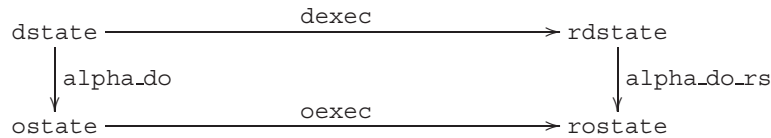


Fig. 1. Commutative diagram of defensive and offensive execution

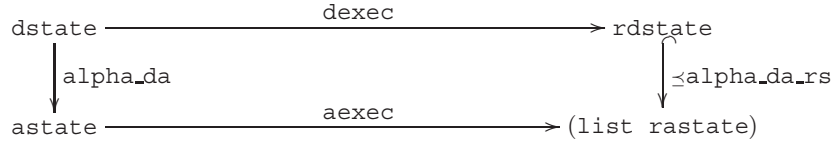


Fig. 2. Commutative diagram of defensive and abstract execution

The commutation of both diagrams is proved by case analyses, first on the bytecode to be executed and second on the state of the defensive virtual machine, and finally by equational reasoning.

Summary The module JCVM represents most of the effort behind CertiCartes. However, the construction of the offensive and abstract JCVMs relies on abstract interpretation and can be automated. Likewise, the cross-validation of the JCVM relies on case analyses and equational reasoning and can also be automated.

2.2 Module BCV

The module BCV constructs a certified bytecode verifier from an abstract virtual machine using a formalization of Kildall’s algorithm that is similar to the one presented in [28,37]. Our formalization of Kildall’s algorithm takes as input:

- a type `astate` of abstract states and a well-founded relation `lt_astate` over `astate`;
- an abstract virtual machine `aexec: astate → (list astate)` which is monotone w.r.t. `lt_astate`;

and produces as output:

- an executable function `bcv: jcprogram → bool`;
- a proof that a program `p` such that `bcv p = true` does not raise any typing error when executed on the abstract virtual machine `aexec`.

Additionally, the module BCV contains a justification of method-by-method verification, i.e. a proof that a program is well-typed if all its methods are. Together with the proofs of the commuting diagrams, these results ensure that bytecode verification is correct.

Summary The module BCV is largely independent from the specifics of the virtual machine under consideration, and hence could be applied to other settings with minimal effort.

3 Constructing Certified Abstractions with Jakarta

The objective of Jakarta is to provide an interactive environment for transforming virtual machines and for verifying the correctness of these transformations in proof assistants. Examples of such transformations include refinements and optimizations, but to date our work focuses on abstractions. In what follows, we present the abstraction mechanism of Jakarta, and describe how it has been used to extract the offensive and abstract JCVMs from the defensive JCVM.

3.1 Current Focus

The current focus with Jakarta is to provide automatic support for the construction and cross-validation of the virtual machines for an arbitrary property P . Concretely, we want the Jakarta toolset to take as input:

- a P -defensive virtual machine `dexec`: `dstate` \rightarrow `dstate` that manipulates typed values (the notion of type being determined by P) and enforces P through run-time type-checks;
- two abstraction functions:

```
alpha_da : dstate  $\rightarrow$  astate
alpha_do : dstate  $\rightarrow$  ostate
```

where `astate` and `ostate` respectively denote some suitably defined types of abstract and offensive states;

- transformation scripts which provide instructions on how to conduct the abstraction of the P -defensive virtual machine into a P -abstract (resp. P -offensive) one;

and produces as output a P -offensive and a P -abstract virtual machines:

```
oexec : ostate  $\rightarrow$  orstate
aexec : astate  $\rightarrow$  (list arstate)
```

together with the proofs that they constitute sound transformations of the P -defensive virtual machine.

3.2 An Overview of Jakarta

The Jakarta Specification Language JSL is a small language for describing virtual machines in a neutral mathematical style. JSL is a polymorphically typed language whose execution model is based on term rewriting [3,10].

A JSL theory is given by a set of declarations that introduce first-order, polymorphic datatypes and their constructors, and by a set of function definitions that introduce function symbols and fix their computational meaning via conditional rewrite rules of the form:

$$l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$$

A specificity of JSL is to require that the right-hand sides of the conditions, i.e the r_i s, should be patterns with fresh variables (recall that a pattern is a linear term built from variables and constructors). Such a format of rewriting seems unusual but is closely related to pattern-matching in functional programming and proof assistants. Figure 3 gives the JSL specification of the `ifnull` bytecode.

The Jakarta Prover Interface provides a means to translate JSL specifications to proof assistants and theorem provers. Currently, we support translations from JSL to Coq, Isabelle [38], PVS [41] and Spike [12]. These translations may produce programs which are considered ill-formed by the target system (proof assistants often reject partial, non-deterministic or non-terminating functions), but it has not turned out to be a problem in practice.

```

1  data valu_prim = VReturnAddress nat | VBoolean z |
2                    VByte z | VShort z | VInt z.
3
4  data valu_ref = VRef_null | VRef_array type0 heap_idx |
5                    VRef_instance cap_class_idx heap_idx |
6                    VRef_interface cap_interf_idx heap_idx .
7
8  data valu = VPrim valu_prim | VRef valu_ref.
9
10 data list á = Nil | Cons á (list á).
11
12 data exc á = Value á | Error.
13
14 function ifnull : bytecode_idx → dstate → rdstate :=
15   (stack_f state) → Nil
16     ⇒ (ifnull branch state) → (abortCode State_error state);
17
18   (stack_f state) → (Cons h lf),
19   (head (opstack h)) → (Value v),
20   v → (VPrim v0)
21     ⇒ (ifnull branch state) → (abortCode Type_error state);
22
23   (stack_f state) → (Cons h lf),
24   (head (opstack h)) → (Value v),
25   v → (VRef vr),
26   (res_null vr) → True
27     ⇒ (ifnull branch state)
28       → (update_frame (update_pc branch
29                        (update_opstack (tail (opstack h)) h)) state);
30
31   (stack_f state) → (Cons h lf),
32   (head (opstack h)) → (Value v),
33   v → (VRef vr),
34   (res_null vr) → False
35     ⇒ (ifnull branch state)
36       → (update_frame (update_pc (S (p_count h))
37                        (update_opstack (tail (opstack h)) h)) state);
38
39   (stack_f state) → (Cons h lf),
40   (head (opstack h)) → Error
41     ⇒ (ifnull branch state) → (abortCode Opstack_error state).

```

Fig. 3. Example of JSL specification

Furthermore, we provide a translation from Coq to JSL so as to reuse existing Coq developments and a translation from JSL to OCaml [35] for an efficient execution of JSL specifications.

The Jakarta Transformation Kit JTK provides a means to perform automatic abstractions on JSL specifications, and is described in more detail in the next subsections. Finally, the *Jakarta Automation Kit* aims at facilitating the process of cross-validating virtual machines in proof assistants, and is discussed in Subsection 3.6.

3.3 The Jakarta Transformation Kit

Principles The abstraction \hat{S} of a JSL theory S is constructed in three successive steps:

1. the user provides for every datatype T of S a corresponding abstract datatype \hat{T} together with an abstraction function α_T from T to \hat{T} ; abstraction functions are written in JSL, but are only allowed to use unconditional rewriting;
2. the user builds a transformation script that guides the abstraction process in places where automatic procedures are inaccurate; the purpose and effect of this script is detailed below.
3. based on the inputs of steps 1 and 2 above, the JTK constructs automatically for every function $f : T_1 \rightarrow \dots \rightarrow T_n \rightarrow U$ of S its abstract counterpart $\hat{f} : \hat{T}_1 \rightarrow \dots \rightarrow \hat{T}_n \rightarrow \hat{U}$. Unless indicated otherwise by the abstraction script, the function \hat{f} is constructed by a syntactic translation on the rewrite rules defining f , followed by a cleaning phase where vacuous conditions and rules are removed.

Most of the abstraction process is handled automatically but there are however situations where user interaction is required. It is the purpose of the transformation script to facilitate such interactions through a carefully designed set of commands.

Commands fall into three categories:

discarding commands specify which patterns in expressions are unwanted and must be deleted. Discarding commands are used for example to remove all verifications and operations related to typing from the the offensive JCVM;

substitution commands specify coercions and substitutions that should be used during the translation of rewrite rules. Substitution commands are used for example to replace dynamic lookup by static lookup in the abstract semantics of method invocation;

post-processing commands deal with functions which may have become non-deterministic or non-total during abstraction. Post-processing commands are used for example to collect all possible return states into lists during the construction of the abstract JCVM.

Each command is of the general form:

$$\langle \text{command} \rangle ::= \langle \text{action} \rangle \langle \text{target} \rangle \text{list} [\text{by } \langle \text{expr} \rangle] [\text{in } \langle \text{scope} \rangle \text{list}]$$

- The optional field *scope* determines in which part of the JSL specification the action must be performed; by default the scope of a command is the whole specification.
- The field *target* specifies which syntactic compounds trigger the action.
- The optional field *expr* is used in substitution commands to specify the syntactic compounds by which targets must be replaced.

The form of the *scope* and *target* fields is defined as follows, where $\langle f \rangle$ stands for function names:

$$\begin{aligned} \langle \text{target} \rangle & ::= \langle f \rangle \mid \langle f \rangle . \langle \text{loc} \rangle \mid \langle f \rangle @ \langle \text{nat} \rangle \mid \langle \text{expr} \rangle \\ \langle \text{scope} \rangle & ::= \langle f \rangle \mid \langle f \rangle . \langle \text{loc} \rangle \\ \langle \text{loc} \rangle & ::= \langle \text{nat} \rangle \mid \langle \text{nat} \rangle . \langle \text{loc} \rangle \end{aligned}$$

We illustrate the meaning of target and scope by the following examples:

- $f.2$ denotes the second rewriting rule of the definition of the function f ;
- $f.2.3$ denotes the third condition of $f.2$. By convention, $f.2.0$ stands for the conclusion of $f.2$;
- $f.2.3.1$ denotes the left hand side of $f.2.3$ and $f.2.3.2$ stands for the right hand side of the same;
- $f@i$ denotes the i th argument of f .

3.4 Synthesis of the Offensive JCVM

Figure 4 contains the script used to generate from the defensive JCVM (which is over 10000 lines long) the offensive JCVM (5000 lines). The script begins with a preamble, which specifies the function to abstract, the name of the resulting function (at line 3) and the abstraction functions (at line 4). Then the remaining of the script is divided in two blocks of discarding commands (between lines 8 and 17) and substitution commands (between lines 21 and 26); there are no post-processing commands for this abstraction.

The block between lines 8 and 17 uses the discarding commands **delete** and **drop**.

- The command **delete** is used to suppress all occurrences of the target within the specified scope. For example at line 8, the script forces type verification operations (i.e. `type_verification...`) to be removed everywhere in the specification. By propagation, these commands may lead to rules being deleted, in particular because rules whose conclusion has been removed are also removed.
- The command **drop** is used to remove some arguments from the signature of a function. Such a command is needed to clean up code that is made vacuous by the abstraction. For example, the function `getFieldObj` takes as third argument some typing information which becomes vacuous in the offensive JCVM. Therefore we instruct the argument to be removed at line 17.

```

1  (* Preamble *)
2
3  build oexec from dexec using
4  abstraction functions alpha_do_prim alpha_do ...
5
6  (* Discarding commands *)
7
8  delete type_verification signature_verification Type_error
9         types_match test_exception_aastore assignment_compatible
10
11 delete test_exception_putfield_ref.2 test_exception_astore.3
12        test_exception_checkcast.4 test_exception_checkcast.5
13        test_exception_putstatic_ref.2
14
15 delete t0 in putstatic putfield_obj
16
17 drop getfield_obj@3 getstatic@1 tableswitch@1 lookupswitch@1
18
19 (* Substitution commands *)
20
21 replace get_inst_from_heap by oget_inst_from_heap
22 replace v2t by (Ref Ref_null)
23 replace (trhi2vr (vr2tr vr) (absolu (v2z v))) by v in putfield_obj.36.0
24
25 select vp2z@1 v2z@1 vr2z@1 vr2hi@1 extr_from_opstack@2
26 select tpz2vp@2 in adr_aload res_opstack getfield_obj putfield_obj ...

```

Fig. 4. Offensive abstraction script

The block between lines 21 and 26 uses the substitution commands **replace** and **select**.

- **replace** allows to replace the target by the argument specified in the **by** field. This argument can be a function name or an expression. This command allows the user to override the expression computed by the standard abstraction mechanism. We use this command at line 21 to provide a definition of the function `oget_inst_from_heap`, which takes as argument a heap `h` and a value `v`, and extracts from `h` the instance referenced by `v`. In this case, the purpose of the substitution is to provide a direct definition of the function; it is possible, but less convenient in this case, to use discarding commands for achieving the same definition. Observe that only three such substitutions are required to obtain a correct specification of the offensive JCVM.
- **select** allows to optimize the specification by projecting a function call onto one of its arguments. Such an optimization is useful when the abstraction produces a function `f` that behaves as a projection function, i.e. such that $f(x_1, \dots, x_n) = x_i$ for all $x_1 \dots x_n$. An example of such function is given by `vp2z`. In the defensive JCVM, this function maps a typed value to its corresponding integer. In the offensive JCVM, the function behaves as the identity, so `(vp2z n)` is replaced by `n` as instructed at line 25. Note that the command affects the computational behavior of the functions, but not their meaning.

```

1  function alpha_da_prim : valu_prim → avalu_prim :=
2  ⇒ alpha_da_prim (VReturnAddress v) → aReturnAddress v ;
3  ⇒ alpha_da_prim (VBoolean v) → aBoolean ;
4  ⇒ alpha_da_prim (VByte v) → aByte ;
5  ⇒ alpha_da_prim (VShort v) → aShort ;
6  ⇒ alpha_da_prim (VInt v) → aInt .
7
8  function alpha_da : dstate → astate :=
9  ⇒ alpha_da (Build_state sh hp s)
10 → Build_astate (alpha_da_sheap sh) (alpha_da_stack s).

```

Fig. 5. Abstraction functions (excerpts)

3.5 Synthesis of the Abstract JCVM

Figure 5 contains some of the abstraction functions used to generate the abstract JCVM (which is around 5000 lines long) from the defensive JCVM. Observe how subtleties of the abstract JCVM are captured by the abstraction functions. For example:

- the abstract JCVM must keep track of return addresses to handle subroutines, and forget all other values: the definition of `alpha_da_prim` reflects this desiderata;
- abstract values do not carry any component that corresponds to the heap: this is reflected in the definition of `alpha_da`.

Figure 6 contains (excerpts of) the script used to generate the abstract JCVM (which is around 5000 lines long); the whole script is approximately 300 lines long. The preamble is immediately followed by a set of commands that act on the whole specification. These commands are used to discard operations and verifications involving the heap or values, for example `res_null`, `throwException` and `abortMemory` at line 10. Such commands are sufficient for handling various significant bytecodes, including `load`, `store`, `new`, `astore`, `goto`, `jsr` and `ret`.

The remaining of the script takes care of all bytecodes whose abstract semantics must be fine-tuned. This includes all bytecodes for method invocation and memory management, which are treated by discarding and substitution commands. We concentrate on branching bytecodes, as they are responsible for non-determinism and require the use of post-processing commands.

- The bytecodes `ifnull` and `ifnonnull` pop the top value of the operand stack and check whether it is null to determine the next program counter. The abstraction renders these functions non-deterministic. We handle this non-determinism with the command **determine** at line 18, whose effect is to transform the bytecodes into deterministic functions of target type `list astate`. The command, which implements the usual strategy of collecting the return values of a non-deterministic function into a list, also propagates the transformation to the function `aexec`.
- The bytecodes `if_scmp_cond`, `if_acmp_cond` and `if_cond` pop two values from the operand stack and compare them in order to determine the


```

1  (* Preamble *)
2
3  build aexec from dexec using
4  abstraction functions alpha_da_prim alpha_da ...
5
6  (* General commands *)
7
8  ...
9
10 delete throwException res_null abortMemory
11
12 ...
13
14 (***** Commands for branching bytecodes *****)
15
16 ...
17
18 determine ifnull ifnonnull
19 drop res_scompare@1,2,3 res_acompare@1,2,3
20 determine res_scompare res_acompare
21 ...

```

Fig. 6. Typed abstraction script (excerpts)

next program counter. This comparison is performed by the auxiliary functions `res_scompare` and `res_acompare`, but the abstraction renders these functions non-deterministic. A similar solution using **determine** applies, see line 20, but for technical reasons not discussed here, some arguments of these functions must be dropped before calling **determine**; the notation **drop** `f@1,2,3` at line 19 is a shorthand for **drop** `f@1 f@2 f@3`.

3.6 Automating Cross-Validation of Virtual Machines

In order to automate the proofs of cross-validation of the three JCVMS in Coq, we have designed specific reasoning tactics that generate induction principles for reasoning about complex recursive functions. These tactics yield drastic improvements; in particular, proof scripts are significantly easier to construct, and their size is reduced by 90% [5]. We expect that further automation is possible, and in particular that equational reasoning can be automated by adapting existing connections between Coq and external tools, such as Elan [1].

In a different line of work, we have been experimenting cross-validation of the JCVMS using Spike [12], an automatic theorem prover based on implicit induction. Currently Spike has been used to establish automatically the commutation properties for over two thirds of the instruction set, but we expect that further optimizations on Spike should allow us to handle even more instructions. This experience, which will be reported elsewhere, suggests that first-order theorem proving techniques could be advantageously exploited in the context of platform verification for smartcards.

3.7 Summary

Jakarta is at an early stage of development, but preliminary results support the claim laid in [6] that the construction and cross-validation of virtual machines could be automated to a large extent.

We have made good our promise to extract the offensive and abstract JCVMs from the defensive JCVM. One line of future research is to do likewise for novel type systems for Java(Card), see e.g. [4,23,31,42]. More generally, we are interested in understanding whether our methodology applies to other typed low-level languages [2,20,47].

Another line of future research is to enhance Jakarta with new functionalities. Support for refinement is a requirement for incremental specification, and our most immediate goal. A longer-term goal is to devise mechanisms that exploit the transformation logs produced by the JTK to guide the automatic construction of cross-validation proofs.

4 Conclusion

Formal verification of the JavaCard platform is the key to Common Criteria evaluations at the highest levels and a challenge for proof assistants. A recent survey [25] highlights the importance that formal methods have had in understanding and improving Sun's specifications of the Java(Card) platforms. But at the same time [25] observes a need to build appropriate environments that provide coherent support for formal specification, verification and provably correct implementations of the JavaCard platform. Such environments should incorporate general-purpose tools for developing and managing large-scale specifications and proofs, and specialized tools that are explicitly designed to reason about smartcard platforms. The design of such environments is a largely open avenue, and a promising field of research.

Acknowledgments

We are especially grateful to Gustavo Betarte and Manuela Pereira for extensive comments on a draft of the paper.

References

1. C. Alvarado and Q.-H. Nguyen. ELAN for equational reasoning in COQ. In J. Despeyroux, editor, *Proceedings of LFM'00*, 2000. Rapport Technique INRIA. 55
2. D. Aspinall and A. Compagnoni. Heap-bound assembly language. Manuscript, 2001. 56
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. 49

4. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002. 44, 56
5. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In C. Muñoz and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. To appear. 55
6. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001. 45, 56
7. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002. 44, 45
8. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001. 44, 45
9. J. van den Berg and B. Jacobs. The LOOP Compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001. 43
10. M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002. 49
11. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic Purse Applet Certification. In S. Schneider and P. Ryan, editors, *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2000. 43
12. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, January 1997. 50, 55
13. L. Casset. Development of an Embedded Verifier for JavaCard ByteCode using Formal Methods. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of FME'02*, *Lecture Notes in Computer Science*, 2002. To appear. 42
14. L. Casset, L. Burdy, and A. Requet. Formal Development of an Embedded Verifier for JavaCard ByteCode. In *Proceedings of DSN'02*. IEEE Computer Society, 2002. 42
15. L. Casset and J.-L. Lanet. A Formal Specification of the Java Byte Code Semantics using the B Method. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proceedings of Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999. 43
16. Z. Chen. *Java Card for Smart Cards: Architecture and Programmer's Guide*. The Java Series. O'Reilly, 2000. 42
17. Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In B. Magnusson, editor, *Proceedings of ECOOP'02*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, 2002. 43
18. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.2*, January 2002. 44
19. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'00*, pages 439–448. ACM Press, 2000. 43

20. K. Crary and G. Morrisett. Type structure for low-level programming languages. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings of ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, 1999. 56
21. Common Criteria. <http://www.commoncriteria.org>. 43
22. e-Europe SmartCards. See <http://eeurope-smartcards.org>. 41
23. S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, November 1999. 44, 56
24. E. Giménez and O. Ly. Formal modeling and verification of the java card security architecture: from static checkings to embedded applet execution, 2002. Talk delivered at the Verificard'02 meeting, Marseille, 7-9 January 2002. Slides available at <http://www-sop.inria.fr/lemme/verificard/2002/programme.html>. 44
25. P. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001. 56
26. JavaCard Technology. <http://java.sun.com/products/javacard>. 42
27. JML Specification Language. <http://www.jmlspecs.org>. 43
28. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 2002. Submitted. 48
29. Gemplus Research Labs. Java Card Common Criteria Certification Using Coq. Technical Report, 2001. 44
30. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998. 43
31. C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 200x. To appear. 44, 56
32. K. R. M. Leino. Extended Static Checking: A Ten-Year Perspective. In R. Wilhelm, editor, *Informatics–10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175, 2001. 43
33. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001. 42
34. X. Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *Proceedings e-Smart 2001*, volume 2140, pages 150–164. Springer-Verlag, 2001. 42
35. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, 2000. 51
36. G. C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997. 43
37. T. Nipkow. Verified Bytecode Verifiers. In F. Honsell and M. Miculan, editors, *Proceedings of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer-Verlag, 2001. 48
38. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. 50
39. J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *Proceedings of the ESORICS'98*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998. 43

40. A. Requet. A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of FMICS'00*, pages 29–46, 2000. 43
41. N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997. 50
42. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999. 44, 56
43. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Platform Security*, 2001. Technical White Paper. 42
44. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Application Programming Interface (API)*, 2002. 42
45. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Runtime Environment (JCRE) Specification*, 2002. 42
46. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Virtual Machine Specification*, 2002. 42
47. K. N. Swadi and A. W. Appel. Typed machine language and its semantics. Manuscript, 2001. 56

Higher-Order Quantification and Proof Search^{*}

Dale Miller

Computer Science and Engineering, 220 Pond Lab, Pennsylvania State University
University Park, PA 16802-6106, USA
dale@cse.psu.edu

Abstract. Logical equivalence between logic programs that are first-order logic formulas holds between few logic programs, partly because first-order logic does not allow auxiliary programs and data structures to be hidden. As a result of not having such abstractions, logical equivalence will force these auxiliaries to be present in any equivalence program. Higher-order quantification can be used to hide predicates and function symbols. If such higher-order quantification is restricted so that operationally, only hiding is specified, then the cost of such higher-order quantifiers within proof search can be small: one only needs to deal with adding new eigenvariables and clauses involving such eigenvariables. On the other hand, the specification of hiding via quantification can allow for novel and interesting proofs of logical equivalence between programs. This paper will present several examples of how reasoning directly on a logic program can benefit significantly if higher-order quantification is used to provide abstractions.

1 Introduction

One of the many goals of declarative programming, and particularly, logic programming, should be that the very artifact that is a program should be a flexible object about which one can reason richly. Examples of such reasoning might be partial and total correctness, various kinds of static analysis, and program transformation.

One natural question to ask in the logic programming setting is, given two logic programs, \mathcal{P}_1 and \mathcal{P}_2 , written as logical formulas, is it the case that \mathcal{P}_1 entails \mathcal{P}_2 and vice versa, the notation for which we will write as $\mathcal{P}_1 \dashv\vdash \mathcal{P}_2$. In other words, are these two programs logically equivalent formulas. If this property holds of two programs, then it is immediate that they prove the same goals: for example, if $\mathcal{P}_1 \vdash G$ and $\mathcal{P}_2 \vdash \mathcal{P}_1$ then $\mathcal{P}_2 \vdash G$. If provability of a goal from a program is described via cut-free proofs, as is often the case for logic programming [MNPS91], then cut-elimination for the underlying logic is needed to support this most basic of inferences.

^{*} This work was supported in part by NSF grants CCR-9912387, CCR-9803971, INT-9815645, and INT-9815731 and a one month guest professorship at L'Institut de Mathématiques de Luminy, University Aix-Marseille 2 in February 2002.

But how useful is the entailment relation $\mathcal{P}_1 \dashv\vdash \mathcal{P}_2$? The answer to this depends a great deal on the logic in which these logic programs are situated. If, for example, \mathcal{P}_1 and \mathcal{P}_2 are first-order Horn clause programs (that is, conjunctions of universally quantified clauses), and entailment is taken as that for first-order classical logic, then this relationship holds for very few pairs of programs \mathcal{P}_1 and \mathcal{P}_2 . The main reasons that this classical, first-order logic equivalence is nearly trivial and hence nearly worthless are outlined in the next three subsections.

1.1 Entailment Generally Needs Induction

Most interesting equivalences of programs will almost certainly require induction. Logics and type systems that contain induction are vital for reasoning about programs and such logics have been studied and developed to a great extent. In this paper, however, we will not be concerned with induction so that we can focus on two other important limiting aspects of logical entailment.

1.2 Classical Logic Is Poor at Encoding Dynamics

Classical logic does not directly support computational dynamics, at least, not directly at the logic level. Truth in classical logic is forever; it is immutable. Computations are, however, dynamic. To code computation in classical logic, all the dynamics of a computation must be encoded as terms that are given as arguments to predicates. That is, the dynamics of computations are buried in non-logical contexts (i.e., with in atomic formulas). As a result, the dynamic aspects of computation are out of reach of logical techniques, such as cut-elimination.

This situation improves a bit if intuitionistic logic is used instead. The notion of truth in Kripke models, which involve possible worlds, allows for some richer modeling of dynamics: enough, for example, to provide direct support for scoping of modules and abstract datatypes [Mil89b, Mil89a]. If one selects a substructural logic, such as linear logic, for encoding logic programs, then greater encoding of computational dynamics is possible. In linear logic, for example, it is possible to model a switch that is now on but later off and to assign and update imperative programming like variables [HM94] as well as model concurrent processes that evolve independently or synchronize among themselves [AP91, Mil96]. Here, counters, imperative variables, and processes are all represented as formulas and not as terms within the scope of a predicate (a non-logical constant). As a result of capturing more of the computational dynamics happening at the level of logic, logical equivalence will be less trivial and more interesting. All the examples that we shall present in this paper will make use of linear logic programming.

1.3 Needed Abstractions Are not Present in First-Order Logics

First-order logics do not provide means to hide or abstract parts of a specification of a computation. For example, if one specifies two different specifications for sorting in first-order Horn clauses, each of with different auxiliary predicates

and data structures, all of these are “visible” in first-order logic, and logical equivalence of two such programs will insist that all predicates, even auxiliary ones, must be equivalent. Logic does provide means of abstracting or hiding parts of specification and even entire data structures. This mechanism is quantification, and to hide predicates and data structures, higher-order quantification is needed. This paper will focus on illustrating how such quantification can be used to specify computation and to help in reasoning about program equivalence.

2 Quantification at Function and Predicate Types

A first-order logic allows quantification over only the syntactic category individuals. Higher-order logics generally allow for quantification, also, of function symbols (quantification at function type) or predicate symbols (quantification at predicate type) or both. For the sake of being concrete, we shall assume that the logic considered here is based on a simple type discipline, and that the type of logical formulas is o (following Church’s Simple Theory of Types [Chu40]). Given this typing, if a quantifier binds a variable of type $\tau_1 \rightarrow \dots \tau_n \rightarrow \tau_0$, where τ_0 is a primitive type, then that quantifier binds a variable at *predicate type* if τ_0 is o and binds a variable at *function type* otherwise.

Logic programming languages that allow for quantification over function symbols and enrich the term language with λ -terms provide support for the *higher-order abstract syntax* approach to representing syntactic expressions involving binders and scope [PE88]. As is familiar with, say, implementations of λ Prolog, Isabelle, and Elf, higher-order unification is adequate for discovering substitutions for quantifiers during proof search in languages containing quantification of individual and function types. A great deal of energy has gone into the effective implementation of such systems, including treatments of higher-order unification, explicit substitution, and search (see, for example, [NM99]). Higher-order unification [Hue75] is rather complicated, but much of higher-order abstract syntax can be maintained using a much weaker notion of unification [Mil91].

Quantification at predicate type is, however, a more complex problem. As is well known, cut-free proofs involving formulas with predicate quantification do not necessarily have the sub-formula property: sometimes predicate expressions (denoting sets and relations) are required that are not simply rearrangements of subformulas present in the end sequent. Higher-order unification alone will not generate enough substitutions of predicate type to yield completeness. Just to convince ourselves that computing predicate substitutions must be genuinely hard, imagine stating the partial correctness of a simple imperative program written using, say, assignment and interaction. Using Hoare logic for such a programming language, the correctness of a looping program is easily written by allowing predicate quantification to represent the quantification of an invariant for the loop. It is, indeed, hard to image a mechanism that would be complete for computing invariants of looping programs.

There has been some interesting work on attempts to find occasions when higher-order substitutions can be automated. See, for example, the work on set

$$\begin{array}{c}
\frac{\Delta \longrightarrow G[y/x]}{\Delta \longrightarrow \forall x_\tau.G} \forall R \qquad \frac{\Delta, D[t/x] \longrightarrow G}{\Delta, \forall x_\tau.D \longrightarrow G} \forall L \\
\frac{\Delta \longrightarrow G[t/x]}{\Delta \longrightarrow \exists x_\tau.G} \exists R \qquad \frac{\Delta, D[y/x] \longrightarrow G}{\Delta, \exists x_\tau.D \longrightarrow G} \exists L
\end{array}$$

Fig. 1. Inference rules for quantifiers. In both the $\exists R$ and $\forall L$ rules, t is a term of type τ . In the $\exists L$ and $\forall R$ rules, y is a variable of type τ that is not free in the lower sequent of these rules

variables [Ble79, Fel00, Dow93]. The logic programming language λ Prolog allows some uses of higher-order quantification (used for higher-order programming), but restricts such quantification so that the necessary predicate substitutions can be done using higher-order unification [NM90].

Figure 1 presents the sequent calculus rules for the universal and existential quantifiers. Notice that the substitution term t can be a λ -term (of type τ) and in the case that that quantification is of predicate type, the term t may contain logical connectives. As a result, the formulas $G[t/x]$ and $D[t/x]$ might have many more logical connectives and quantifiers than the formulas $\forall x.D$ and $\exists x.G$. Of course, if a sequent calculus proof does not contain the $\forall L$ and $\exists R$ inference rules at predicate type, then the treatment of higher-order quantifiers is quite simple: the only other possible introduction rules are $\exists L$ and $\forall R$ and they are simply instantiated (reading a proof bottom up) by a new “eigenvariable” y . The Teyjus implementation [NM99] of λ Prolog, for example, gives a direct and effective implementation of such eigenvariable generation during proof search. It is possible to define rich collections of higher-order formulas for which one can guarantee that only predicate quantification only occurs with $\exists L$ and $\forall R$. Most of the example of logic programs that we present below have this property.

Notice that if \mathcal{P} is such that all universal quantifiers of predicate type occur negatively and all existential quantifiers of predicate type occur positively, then there are no occurrences of $\forall L$ or $\exists R$ in a cut-free proof of the sequent $\mathcal{P} \longrightarrow A$ where A is, say, an atomic formula. If, however, \mathcal{P}_1 and \mathcal{P}_2 are two programs with the above restriction, cut-free proofs of the sequent $\mathcal{P}_1 \longrightarrow \mathcal{P}_2$ may contain occurrences of $\forall L$ and $\exists R$. Such sequents will generally require some substitutions that might be difficult to produce by simple automation. This is consistent with the expectation that inferences between programs require some genuine insights to establish.

We now proceed to present some examples of reasoning with higher-order quantification. All of our examples involve some use of linear logic as well as higher-order quantification. We will not attempt to describe the basics of linear logic, but rather refer the reader to, say, [Gir87, Tro92]. Our first two examples will make use of linear implication, written as \multimap (and as the converse \multimap) and intuitionistic implication, written as \Rightarrow . The multiplicative conjunction \otimes appears as well. Our last example, starting in Section 5 will also make use of the multiplicative disjunction \wp .

3 Reversing a List Is Symmetric

While much of the motivation for designing logic programming languages based on linear logic has been to add expressiveness to such languages, linear logic can also help shed some light on conventional programs. In this section we consider the linear logic specification for the reverse of lists and formally show, by direct reasoning on the specification, that it is a symmetric relation [Mil].

Let the constants nil and $(\cdot :: \cdot)$ denote the two constructors for lists. To compute the reverse of two lists, make a place for two piles on a table. Initialize one pile to the list you wish to reverse and initialize the other pile to be empty. Next, repeatedly move the top element from the first pile to the top of the second pile. When the first pile is empty, the second pile is the reverse of the original list. For example, the following is a trace of such a computation.

$$\begin{array}{c|c} (a :: b :: c :: nil) & nil \\ \hline (b :: c :: nil) & (a :: nil) \\ \hline (c :: nil) & (b :: a :: nil) \\ \hline nil & (c :: b :: a :: nil) \end{array}$$

In more general terms: if we wish to reverse the list L to get K , first pick a binary relation rv to denote the pairing of lists above (this predicate will not denote the reverse); then start with the atom $(rv L nil)$ and do a series of backchaining over the clause

$$rv P (X :: Q) \multimap rv (X :: P) Q$$

to get to the formula $(rv nil K)$. Once this is done, K is the result of reversing L . That is, if from the two formula

$$\begin{array}{l} \forall P \forall X \forall Q (rv P (X :: Q) \multimap rv (X :: P) Q) \\ (rv nil K) \end{array}$$

one can prove $(rv L nil)$, then the reversing of L is K . This specification is not finished for several reasons. First, L and K are specific lists and are not quantified anywhere. Second, the relation $reverse L K$ must be linked to this sub-computation using the auxiliary predicate rv . Third, we can observe that of the two clauses for rv above, the first clause (the recursive one) can be used an arbitrary number of times during a computation while the second clause (the base case) can be used exactly once. Since we are using elements of higher-order linear logic here, linking the rv sub-computation to $(reverse L K)$ can be done using nested implications, the auxiliary predicate rv can be hidden using a higher-order quantifier, and the distinction between the use pattern for the inductive and base case clauses can be specified using different implications. The entire specification of reverse can be written as the following single formula.

$$\forall L \forall K [\forall rv ((\forall X \forall P \forall Q (rv P (X :: Q) \multimap rv (X :: P) Q)) \Rightarrow rv nil K \multimap rv L nil) \multimap reverse L K]$$

Notice that the clause used for repeatedly moving the top elements of lists is to the left of an intuitionistic implication (so it can be used any number of times)

while the formula $(rv\ nil\ K)$, the base case of the recursion, is to the left of a linear implication (must be used once).

Now consider proving that reverse is symmetric: that is, if $(reverse\ L\ K)$ is proved from the above clause, then so is $(reverse\ K\ L)$. The informal proof of this is simple: in the table tracing the computation above, flip the rows and the columns. What is left is a correct computation of reversing again, but the start and final lists have exchanged roles. This informal proof is easily made formal by exploiting the meta-theory of higher-order quantification and of linear logic. A more formal proof proceeds as follows. Assume that $(reverse\ L\ K)$ can be proved. There is only one way to prove this (backchaining on the above clause for $reverse$). Thus the formula

$$\forall rv((\forall X\forall P\forall Q(rv\ P\ (X :: Q) \multimap rv\ (X :: P)\ Q)) \Rightarrow rv\ nil\ K \multimap rv\ L\ nil)$$

is provable. Since this universally quantified expression is provable, any instance of it is also provable. Thus, instantiate it with the λ -expression $\lambda x\lambda y(rv\ y\ x)^\perp$ (the swapping of the order of the arguments is one of the flips of the informal proof and the negation yields the other flip). The resulting formula

$$(\forall X\forall P\forall Q(rv\ (X :: Q)\ P)^\perp \multimap (rv\ Q\ (X :: P)^\perp)) \Rightarrow (rv\ K\ nil)^\perp \multimap (rv\ nil\ L)^\perp$$

can be simplified by using the contrapositive rule for negation and linear implication, which yields

$$(\forall X\forall P\forall Q(rv\ Q\ (X :: P) \multimap rv\ (X :: Q)\ P) \Rightarrow rv\ nil\ L \multimap rv\ K\ nil).$$

If we now universally generalize on rv we again have proved the body of the reverse clause, but this time with L and K switched.

This proof exploits the explicit hiding of the auxiliary predicate rv by providing a site into which a “re-implementation” of the predicate can be placed. Also notice that this proof does not have an explicit reference to induction. It is unlikely to expect that many proofs of interesting properties involving list manipulation predicates can be done like this and without reference to induction. This example simply illustrates an aspect of higher-order quantification and linear logic in reasoning direction with specifications.

4 Two Implementations of a Counter

For another example of how higher-order quantification can be used to form abstractions and to enhance reasoning about code, consider the two different specifications E_1 and E_2 of a simple counter object in Figure 2 [Mil96]. Each of these specifications describe a counter that encapsulates state and responds to two “methods”, namely, get and inc , for getting and incrementing the encapsulated value. Notice that in both of these specifications, the state is the linear atomic formula $(r\ n)$ (n is the integer denoting the counter’s value) and the two methods are specified by two clauses that are marked with a ! (that is, they can

$$\begin{aligned}
E_1 &= \exists r[(r \ 0) \otimes \\
&\quad !\forall K\forall V(\text{get } V \ K \multimap r \ V \otimes (r \ V \multimap K)) \otimes \\
&\quad !\forall K\forall V(\text{inc } V \ K \multimap r \ V \otimes (r \ (V + 1) \multimap K))] \\
E_2 &= \exists r[(r \ 0) \otimes \\
&\quad !\forall K\forall V(\text{get } (-V) \ K \multimap r \ V \otimes (r \ V \multimap K)) \otimes \\
&\quad !\forall K\forall V(\text{inc } (-V) \ K \multimap r \ V \otimes (r \ (V - 1) \multimap K))]
\end{aligned}$$

Fig. 2. Two specifications of a global counter

be invoked any number of times). Notice also that the predicate r is existentially quantified, thus properly encapsulating the state of the counter.

Viewed from a proof search point-of-view, these two specifications store the counter on the left side of the sequent as a linear logic assumption. The counter is then updated by “destructively” reading and then “rewriting” the atom used to store that value. The differences between these two implementations is that in the second of these implementations the *inc* method actually decrements the internal representation of the counter: to compensate for this choice both methods return the negative of that internal value. The use of \otimes , $!$, and \exists in Figure 2 is for convenience in displaying these specifications. If we write E_1 as $\exists r(R_1 \otimes !R_2 \otimes !R_3)$, then using simple “curry/uncurry” equivalences in linear logic we can rewrite $E_1 \multimap G$ to the equivalent formula $\forall r(R_1 \multimap R_2 \Rightarrow R_3 \Rightarrow G)$. These specifications are encoded using continuation passing style: the variable K ranges over continuations.

Although these two specifications of a global counter are different, they should be equivalent in some sense. Although there are several ways that the equivalence of such counters can be proved (for example, trace equivalence), the specifications of these counters are, in fact, *logically* equivalent. In particular, the entailments $E_1 \vdash E_2$ and $E_2 \vdash E_1$ are provable in linear logic. The proof of each of these entailments proceeds (in a bottom-up fashion) by choosing an eigen-variable, say s , to instantiate the existential quantifier in the left-hand specification and then instantiating the right-hand existential quantifier with $\lambda x.s \ (-x)$. The proof of these entailments must also use the equations

$$\{-0 = 0, -(x + 1) = -x - 1, -(x - 1) = -x + 1\}.$$

Clearly, logical equivalence is a strong equivalence: it immediately implies no logical context can tell the difference between these two implementations.

5 Multiset Rewriting in Proof Search

To provide some more examples of direct reasoning with logical specification, we will first describe how certain aspects of security protocols can be specified in linear logic. To discuss specifications of security protocols, we first introduce multiset rewriting and how that can be encoded in proof search.

To model multiset rewriting we shall use a subset of linear logic similar to the *process clauses* introduced by the author in [Mil93]. Such clauses are simply described as follows: Let G and H be formulas composed of \perp , \wp , and \forall . (Think of the \wp connective as the multiset constructor and \perp as the empty multiset.) Process clauses are closed formulas of the form $\forall \bar{x}[G \multimap H]$ where H is not \perp and all free variables of G are free in H . These clause have been used in [Mil93] to encode a calculus similar to the π -calculus. A nearly identical subset of linear logic has also been proposed by Kanovich [Kan92, Kan94]: if you write process clauses in their contrapositive form (replacing the connectives \wp , \forall , \perp , and \multimap with \otimes , \exists , $\mathbf{1}$, and \multimap , respectively) you have what Kanovich called *linear Horn clauses*.

The multiset rewriting rule $a, b \Rightarrow c, d, e$ can be implemented as a backchaining step over the clause $c \wp d \wp e \multimap a \wp b$. That is, backchaining using this linear logic clause can be used to justify the inference rule

$$\frac{\Psi; \Delta \longrightarrow c, d, e, \Gamma}{\Psi; \Delta \longrightarrow a, b, \Gamma},$$

with the proviso that $c \wp d \wp e \multimap a \wp b$ is a member of Ψ . We can interpret this fragment of a proof as a rewriting of the multiset a, b, Γ to the multiset c, d, e, Γ by backchaining on the clause displayed above. Using the Forum presentation of linear logic [Mil96], this inference rule can be justified with the following proof fragment.

$$\frac{\frac{\frac{\Psi; \Delta \longrightarrow c, d, e, \Gamma}{\Psi; \Delta \longrightarrow c, d \wp e, \Gamma}}{\Psi; \Delta \longrightarrow c \wp d \wp e, \Gamma} \quad \frac{\frac{\Psi; \cdot \xrightarrow{a} a}{} \quad \frac{\Psi; \cdot \xrightarrow{b} b}{} }{\Psi; \cdot \xrightarrow{a \wp b} a, b}}{\Psi; \Delta \xrightarrow{c \wp d \wp e \multimap a \wp b} a, b, \Gamma}}{\Psi; \Delta \longrightarrow a, b, \Gamma}$$

The sub-proofs on the right are responsible for deleting from the right-hand context one occurrence each of the atoms a and b while the subproof on the left is responsible for inserting one occurrence each of the atoms c , d , and e .

Example 1. Consider the problem of Alice wishing to communicate a value to Bob. The clause

$$\forall x[(a \ x) \wp b \multimap a' \wp (b' \ x)]$$

illustrates how one might synchronize Alice's agent $a \ x$ with Bob's agent b . In one, atomic step, the synchronization occurs and the value x is transfer from Alice, resulting in her continuation a' , to Bob, resulting in his continuation $b' \ x$. If a server is also involved, one can imagine the clause being written as

$$\forall x[(a \ x) \wp b \wp s \multimap a' \wp (b' \ x) \wp s],$$

assuming that the server's state is unchanged through this interaction.

As this example illustrates, synchronization between agents is easy to specify and can trivialize both the nature of communication and the need for security

protocols entirely. (For example, if such secure communications is done atomically, there is no need for a server s in the above clause.) While the clause in this example might *specify* a desired communication, it cannot be understood as an actual implementation in a distributed setting. In distributed settings, synchronization actually only takes place between agents and networks. Our main use of multiset rewriting will involve more restricted clauses with weaker assumptions about communications: these will involve synchronizations between agents and network messages (a model for asynchronous communications) and not between agents and other agents (a model of synchronous communications).

In general, however, the body of clauses are allowed to have universal quantification: since $(\forall x.Px) \wp Q$ is linear logically equivalent to $\forall x(Px \wp Q)$, we can assume such bodies are in prenex normal form (provided that x is not free in Q). Backchaining over the clause

$$\forall x_1 \dots \forall x_i [a_1 \wp \dots \wp a_m \multimap \forall y_1 \dots \forall y_j [b_1 \wp \dots \wp b_n]]$$

can be interpreted as multiset rewriting but where the variables y_1, \dots, y_j are instantiated with eigenvariables of the proof.

6 Security Protocols in Proof Search

We shall briefly outline the multiset rewriting framework proposed by Cervesato, *et. al.* in MSR [CDL+99, CDL+00]. As we have seen, universal quantification is used to handle schema variables (such as the x variable in Example 1) as well as eigenvariables: when the latter are treated properly in a sequent calculus setting, the proviso on their newness can be used to model the notions of freshness and newness in security protocols for nonces, session keys, and encryption keys.

For the sake of concreteness and simplicity, data and messages on a network will be represented as terms of type *data*. We shall use a tupling operator $\langle \cdot, \cdot \rangle$ that has type $data \rightarrow data \rightarrow data$ to form composite data objects and the empty tuple $\langle \rangle$ will have the type *data*. Expressions such as $\langle \cdot, \cdot, \dots, \cdot \rangle$ will be assumed to be built from pairing, associated to the right.

As suggested in the previous section, not just any clause makes sense in a security protocol. Various restrictions on the occurrence of predicates within clauses must be made. For example, we need to avoid that agents synchronize directly with other agents and we must avoid that one agent becomes another agent. (In Section 8 we show a different syntax for protocol clauses which will not need these various restrictions.) In general, the specification of an action possible by Alice is given by (the universal closure of) a clause of the form

$$a S \wp [[M_1]] \wp \dots \wp [[M_p]] \multimap \forall n_1 \dots \forall n_i [a' S' \wp [[M'_1]] \wp \dots \wp [[M'_q]],$$

where p , q , and i are non-negative integers. This clause indicates that Alice with memory S (represented by the atom $a S$) inputs the p network messages $[[M_1]], \dots, [[M_p]]$ and then, in a context where n_1, \dots, n_i are new symbols, becomes the continuation a' with new memory S' and with q new output messages

$[[M'_1]], \dots, [[M'_d]]$. Two variants of this clause can also be allow: the first is where the atom $a s$ is not present in the *head* of the clause (such clauses encode agent creation) and the second is where the atom $a' S'$ is not present in the *body* of the clause (such clauses encode agent deletion).

7 Encryption as an Abstract Datatype

We now illustrate our first use of higher-order quantification at a non-predicate type. In particular, we shall model encryption keys as eigenvariables of higher-order type $data \rightarrow data$. Clearly, this is a natural choice of type since it is easy to think of encryption as being being a mapping from data to data: a particular implementation of this via 64-bit string and some particular encryption algorithm is, of course, abstracted away at this point. Building data objects using higher-type eigenvariables is a standard approach in logic programming for modeling abstract datatypes [Mil89a]. In order to place this higher-type object within data, we introduce a new constructor \cdot° of type $(data \rightarrow data) \rightarrow data$ that explicitly coerces an encryption function into data.

Explicit quantification of encryption keys can be used to also describe succinctly the static distribution of keys within agents. Consider, for example, the following specification.

$$\begin{aligned} \exists k_{as} \exists k_{bs} [& a \langle M, S \rangle \quad \circ - \quad a S \mathfrak{F} [[k_{as} M]]. \\ & b T \mathfrak{F} [[k_{bs} M]] \circ - \quad b \langle M, T \rangle. \\ & s \langle \rangle \mathfrak{F} [[k_{as} P]] \circ - \quad s \langle \rangle \mathfrak{F} [[k_{bs} P]]. \end{aligned}$$

(Here as elsewhere, quantification of capital letter variables is universal with scope limited to the clause in which the variable appears.) In this example, Alice (a) communicates with Bob (b) via a server (s). To make the communications secure, Alice uses the key k_{as} while Bob uses the key k_{bs} . The server is memoryless and only takes on the role of translating messages encrypted for Alice to messages encrypted for Bob. The use of the existential quantifiers helps establish that the occurrences of keys, say, between Alice and the server and Bob and the server, are the only occurrences of those keys. Even if more principals are added to this system, these occurrences are still the only ones for these keys. Of course, as protocols are evaluated (that is, a proof is searched for), keys may extrude their scope and move freely around the network and into the memory of possible intruders. This dynamic notion of scope extrusion is similar to that found in the π -calculus [MPW92] and is modeled here in linear logic in a way similar to an encoding given in [Mil93] for an encoding of the π -calculus into linear logic.

Example 2. To illustrate the power of logical entailment using quantification at higher-order type for encryption keys, consider the following two clauses:

$$a \circ - \forall k. [[(k m)]] \quad \text{and} \quad a \circ - \forall k. [[(k m')]].$$

These two clauses specify that Alice can take a step that generates a new encryption key and then outputs either the message m or m' encrypted. Since Alice

has no continuation, no one will be able to decode this message. It should be the case that these two clauses are equivalent, but in what sense? It is an easy matter to show that these two clauses are actually logically equivalent. A proof that the first implies the second contains a subproof of the sequent

$$\forall k. [[(k\ m')]] \longrightarrow \forall k. [[(k\ m)]],$$

and this is proved by introducing the eigenvariable, say c , on the right and the term $\lambda w.(c\ m)$ on the left.

Public key encryption can be encoded using a “key” of type $data \rightarrow data \rightarrow data$ as the following clause illustrates:

$$\begin{aligned} \exists k. [a\ S\ \mathfrak{A} [[(k\ N\ M)]] \circ - a' \langle S, M \rangle. \\ \text{pubkey\ alice}\ M \circ - \forall n. [[(k\ n\ M)]]]. \end{aligned}$$

We have introduced an auxiliary predicate for storing public keys: this is reasonable since such keys should be something that can be looked up in a registry. For example, the following clause describes a method for Bob to send Alice a message using her public key.

$$\forall M \forall S. b \langle M, S \rangle \circ - b' S \mathfrak{A} \text{pubkey\ alice}\ M.$$

Every instance of the call to *pubkey* places a new nonce into the encrypted data and only Alice has the full key that makes it possible for her to ignore this nonce and only decode the message.

For a more interesting example, we specify the Needham-Schroeder Shared Key protocol (following [SC01]) in Figure 3. Notice that two shared keys are used in this example and that the server creates a new key that is placed within data and is then used for Alice and Bob to communicate directly. It is easy to show that this protocol implements the specification (taken from Example 1):

$$\forall x [(a\ x) \mathfrak{A} b \mathfrak{A} s \circ - a' \mathfrak{A} (b' x) \mathfrak{A} s].$$

That is, the formula displayed in Figure 3 logically entails the above displayed formula. The linear logic proof of this entailment starts with the multiset $(a\ c)$, b, s on the right of the sequent arrow (for some “secret” eigenvariable c) and then reduces this back to the multiset $a', (b' c), s$ simply by “executing” the logic program in Figure 3. Notice that the \forall used in the bodies of clauses in this protocol are used both for nonce creation (at type *data*) and encryption key creation (at type $data \rightarrow data$).

8 Abstracting over Internal States of Agents

Existential quantification over program clauses can also be used to hide predicates encoding agents. In fact, one might argue that the various restrictions on sets of process clauses (no synchronization directly with atoms encoding agents,

$$\begin{array}{l}
\exists k_{as} \exists k_{bs} \{ \\
\quad a S \quad \circ - \quad \forall na. a \langle na, S \rangle \wp [\langle alice, bob, na \rangle]. \\
\quad a \langle N, S \rangle \wp [(k_{as} \langle N, bob, K, En \rangle)] \quad \circ - \quad a \langle N, K, S \rangle \wp [En]. \\
\quad a \langle Na, Key^\circ, S \rangle \wp [(Key Nb)] \quad \circ - \quad a \langle \rangle \wp [(Key \langle Nb, S \rangle)]. \\
\quad \quad b \langle \rangle \wp [(k_{bs} \langle Key^\circ, alice \rangle)] \quad \circ - \quad \forall nb. b \langle nb, Key^\circ \rangle \wp [(Key nb)]. \\
\quad b \langle Nb, Key \rangle \wp [(Key \langle Nb, S \rangle)] \quad \circ - \quad b S. \\
\quad \quad s \wp [\langle alice, bob, N \rangle] \quad \circ - \quad \forall k. s \wp [(k_{as} \langle N, bob, k^\circ, k_{bs} \langle k^\circ, alice \rangle)]]. \\
\}
\end{array}$$

Fig. 3. Encoding the Needham-Schroeder protocol

no agent changing into another agent, etc) might all be considered a way to enforce locality of predicates. Existential quantification can, however, achieve this same notion of locality, but much more declaratively.

First notice that such quantification can be used to encode 3-way synchronization using 2-way synchronization via a hidden intermediary. For example, the following entailment is easy to prove in linear logic.

$$\exists x. \left\{ \begin{array}{l} a \wp b \circ - x \\ x \wp c \circ - d \wp e \end{array} \right\} \quad \dashv\vdash \quad a \wp b \wp c \circ - d \wp e$$

In a similar fashion, intermediate states of an agent can be taken out entirely. For example,

$$\exists a_2, a_3. \left\{ \begin{array}{l} a_1 \wp [[m_0]] \circ - a_2 \wp [[m_1]] \\ a_2 \wp [[m_2]] \circ - a_3 \wp [[m_3]] \\ a_3 \wp [[m_4]] \circ - a_4 \wp [[m_5]] \end{array} \right\} \quad \dashv\vdash$$

$$a_1 \wp [[m_0]] \circ - ([[m_1]] \circ - ([[m_2]] \circ - ([[m_3]] \circ - ([[m_4]] \circ - ([[m_5]] \wp a_4))))))$$

The changing of polarity that occurs when moving to the premise of a $\circ -$ flips expressions from one doing an output (e.g., $[[m_1]]$) to one doing an input (e.g., $[[m_2]]$), etc. This suggests an alternative syntax for agents where $\circ -$ alternates between agents willing to input and output of messages. To that end, consider the following syntactic categories of linear logic formulas:

$$H ::= A \mid \perp \mid H \wp H \mid \forall x. H$$

$$D ::= H \mid H \circ - D \mid \forall x. D$$

If A denotes the class of atomic formulas encoding network messages, then formulas belonging to the class H denote bundles of messages that are used as either input or output. Formulas belonging to the class D can have very deep nesting of implications and that nesting changes phases from input to output and back to input. Notice that the universal quantifier can appear in two modes: one mode it is used to generate new eigenvariables and in the other mode it is used to bind to parts of input messages (as in value-passing CCS).

To illustrate an example of this style of syntax, consider first declaring local all agent predicates in the Needham-Schroeder Shared Key protocol in Figure 3.

$$\begin{array}{l}
(\text{Out}) \quad \forall na. [[\langle alice, bob, na \rangle]] \multimap \\
(\text{In}) \quad (\forall Kab \forall En. [[kas \langle na, bob, Kab^\circ, En \rangle]] \multimap \\
(\text{Out}) \quad ([[En]] \multimap \\
(\text{In}) \quad (\forall NB. [[(KabNB)]] \multimap \\
(\text{Out}) \quad ([[Kab(NB, secret)]])))). \\
\\
(\text{Out}) \quad \perp \multimap \\
(\text{In}) \quad (\forall Kab. [[(kbs(Kab^\circ, alice)]] \multimap \\
(\text{Out}) \quad (\forall nb. [[(Kabnb)]] \multimap \\
(\text{In}) \quad ([[Kab(nb, secret)]] \multimap \\
(\text{Cont}) \quad (b \text{ secret}))). \\
\\
(\text{Out}) \quad \perp \Leftarrow \\
(\text{In}) \quad (\forall N. [[\langle alice, bob, N \rangle]] \multimap \\
(\text{Out}) \quad (\forall key. [[kas \langle N, bob, key^\circ, kbs(key^\circ, alice) \rangle]])).
\end{array}$$

Fig. 4. Encodings of Alice, Bob, and the server (respectively)

This then yields the logically equivalent presentation in Figure 4. (Actually, to get this logical equivalence, two clauses must be added to Figure 3 that allows for the server's creation and the server's deletion.) There three formulas are displayed: the first represents Alice, the second Bob, and the final one the server. If these three formulas were placed on the right-hand side of a sequent arrow (with no theory assumed on the right) then Alice will output a message and move to the left-side of the sequent arrow ($\multimap R$). Bob and the server output nothing and move to the left-hand side as well. At that point, the server will need to be chosen for a $\multimap R$, which will cause it to input the message that Alice sent and then move its continuation to the right-hand side. It will then immediate output another message, and so on.

The style of specification given in Figure 4 is essentially like that of process calculus. Notice that the only essential difference between this style of specification and the one offered for MSR is how one views continuations of agents: in MSR, they are named formulas while in the process style, they are directly presented via nesting. This difference also forces a difference in the presentation of agent memory: in MSR, this was encoded as terms associated to the agent name while in the process style presentation, it is implicitly encoded via the scope of various bound variables over the process's continuation.

If one skips a phase, the two phases adjacent to the skipped phase can be contracted as follows:

$$p \multimap (\perp \multimap (q \multimap k)) \equiv p \wp q \multimap k$$

The relationship between these two styles of agents has already been observed more generally. A *bipolar formula* is a linear logic formula in which no

asynchronous connective (\wp , \forall , $\&$, etc) is in the scope of a synchronous connective (\otimes , \exists , \oplus , etc). Andreoli [And92] showed that if we allow the addition of constants, arbitrary formulas of linear logic can be “compiled” to a collection of bipolar forms: basically, when the nesting alternates once, introduce new constants and have these be defined to handle the meaning of the next alternation, and so on. In essence, the compilation of the formula in Figure 4 yields the formulas in Figure 3: the new constants introduced by compilation are the names used to denote agent continuation. The formulas used in MSR are bipolars: for example, $Q_1 \wp \dots \wp Q_m \multimap P_1 \wp \dots \wp P_n$ is logically equivalent to $Q_1 \wp \dots \wp Q_m \wp (P_1^\perp \otimes \dots \otimes P_n^\perp)$.

9 Conclusion

We have shown that abstractions in the specification of logic programs via quantification at higher-order types can improve the chances that one can perform interesting inferences directly on the logic specification. Induction and co-induction will certainly be important, if not central, to establishing most logical entailments involving logic specifications. Abstractions of the form we have discussed here, however, will most likely play an important role in any comprehensive approach to reasoning about logic programming specifications.

References

- [And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. 73
- [AP91] J. M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991. 61
- [Ble79] W. W. Bledsoe. A maximal method for set variables in automatic theorem-proving. In *Machine Intelligence 9*, pages 53–100. John Wiley & Sons, 1979. 63
- [CDL⁺99] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A meta-notation for protocol analysis. In R. Gorrieri, editor, *Proceedings of the 12th IEEE Computer Security Foundations Workshop — CSFW’99*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press. 68
- [CDL⁺00] Iliano Cervesato, Nancy A. Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In P. Syverson, editor, *13th IEEE Computer Security Foundations Workshop — CSFW’00*, pages 35–51, Cambridge, UK, 3–5 July 2000. IEEE Computer Society Press. 68
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940. 62
- [Dow93] Gilles Dowek. A complete proof synthesis method for the cube of type systems. *Journal of Logic and Computation*, 3(3):287–315, 1993. 63

- [Fel00] Amy Felty. The calculus of constructions as a framework for proof search with set variable instantiation. *Theoretical Computer Science*, 232(1-2):187–229, February 2000. 63
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 63
- [HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. 61
- [Hue75] Gérard Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975. 62
- [Kan92] Max Kanovich. Horn programming in linear logic is NP-complete. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 200–210. IEEE Computer Society Press, June 1992. 67
- [Kan94] Max Kanovich. The complexity of Horn fragments of linear logic. *Annals of Pure and Applied Logic*, 69:195–241, 1994. 67
- [Mil] Dale Miller. An overview of linear logic programming. To appear in a book on linear logic, edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott. Cambridge University Press. 64
- [Mil89a] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press. 61, 69
- [Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):79–108, January 1989. 61
- [Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991. 62
- [Mil93] Dale Miller. The π -calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the 1992 Workshop on Extensions to Logic Programming*, number 660 in LNCS, pages 242–265. Springer-Verlag, 1993. 67, 69
- [Mil96] Dale Miller. Forum: A multiple-conclusion specification language. *Theoretical Computer Science*, 165(1):201–232, September 1996. 61, 65, 67
- [MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. 60
- [MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, pages 1–40, September 1992. 69
- [NM90] Gopalan Nadathur and Dale Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, October 1990. 63
- [NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS. 62, 63
- [PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988. 62
- [SC01] Paul Syverson and Iliano Cervesato. The logic of authentication protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume LNCS 2171. Springer-Verlag, 2001. 70

- [Tro92] Anne S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992. 63

Algebraic Support for Service-Oriented Architecture

(Extended Abstract)

JosÉLuiz Fiadeiro

ATX Software S. A. and LabMOLñUniversity of Lisbon
Alameda AntŪnio SÈrgio 7 ñ 1 C, 2795-023 Linda-a-Velha, Portugal
jose@fiadeiro.org

Abstract. The Net-based economy, now fuelled by the increasing availability of wireless communication, is imposing a new architectural model for software systems which, mirroring what is already happening in the business domain, is based on *services* instead of products. The trend is to support this model through the use of object-oriented approaches, but two essential questions have barely been addressed so far: How *new* is this Software Technology? What does it require from Algebraic Methodology?

1 On the Challenges Raised by Service-Oriented Architectures

In the literature, *Web Services* in general are being promoted as a technology for *dynamic -business*, the next generation of the Internet *culture* in which a shift is made from B2C to B2B [8]. This shift puts an emphasis on program-to-program interaction, which is quite different from the user-to-program interaction that characterised (thin) clients interacting with business applications in the B2C model. In particular, initiatives that were typical of the user side, like *searching* (although not necessarily *surfing* Ö), have now to be performed on the business side, which means supported by software. Readings on the technologies that have been made available in the past few months will normally emphasise this shift from server-to-server, static, linear interaction to dynamic, mobile and unpredictable interactions between machines operating on a network, and identify it as one of the challenges that needs to be met in full for the architecture to impose itself in its full potential.

The models that have been proposed in the meanwhile, namely the Service Oriented Architectures based on publish/find/bind, address these issues directly in terms of technological solutions that can be supported immediately, at varying levels, by open source software. The pillars of these solutions are XML ñ The Extensible Markup Language from the World Wide Web Consortium (W3C) for the definition of portable structured data; SOAP ñ The Simple Object Access Protocol, an XML-based lightweight protocol for the exchange of information in decentralized, distributed environments; WSDL ñ The Web Services Design Language, an XML vocabulary that provides a standard way for service providers to describe the format of requests and response messages for remote method invocations; and UDDI ñ The Universal De-

sign, Discovery, and Integration specification, a standard for B2B interoperability that facilitates the creation, design, discovery, and integration of Web-based services.

The best known initiatives in this area are Microsoft Biztalk, HP e-speak, and Sun Jini. We believe that these efforts lack sufficient generality and, even, imagination in exploring the full potential of the service-oriented paradigm. In order to understand what exactly is necessary to support the engineering and deployment of Web Services, and, hence, what is still required in terms of research and development effort, we have to characterise exactly what this new architecture is about and how it relates to the software development methodologies and supporting technologies that are available.

Web Services have been often characterised as *self-contained*, modular applications that can be described, published, located, and invoked over a network, generally the Web [8]. Building applications (in fact, new services that can themselves be published) is a dynamic process that consists in locating services that provide the basic functionalities that are required, and *orchestrating* them, i.e. establishing collaborations between them, so that the desired global properties of the application can emerge from their joint behaviour. Hence, this new architecture is often characterised by three fundamental aspects for which support is required: *publishing* services so that they can be used by other services; *finding* services that meet required properties; and *binding* to services after they are located to achieve the required integration.

Integration is another keyword in this process, often found married to *orchestration* or *marshalling*: application building in service-oriented architectures is based on the composition of services that have to be discovered and *marshalled* dynamically at run-time. Therefore, one of the characteristics of the service-oriented paradigm is, precisely, the ability that it requires for interconnections to be established and revised dynamically, in run-time, without having to suspend execution, i.e. without interruption of *service*. This is what is usually called *late* or *just-in-time* integration (as opposed to compile or design time integration).

2 Why Object-Oriented Techniques Cannot Address Them

Although it seems clear that Web Services are the *next step* in what has been the *evolution* of Software Engineering, we consider that this step is not itself an *evolution* of existing concepts and techniques. We believe that the move from object to service oriented systems is a true shift of paradigms, one that should be fully and formally characterised so that both methodologies and supporting technologies can be developed to take maximum profit of its potential.

The main reason for our disagreement to what is generally perceived as an evolution is precisely the fact that the shift from objects/components to services is reflected fundamentally on the interactions. Web Services require flexible architectures that are able to make the resulting systems amenable to change. For this purpose, interactions cannot be hardwired in the code that implements the services, which leads to systems that are too tightly coupled for the kind of dynamics required by the Web. The view exposed in section 1 is of systems that are in a continuous process of reconfiguration due to the fact that services need to establish, dynamically, the collaborations that allow them to provide the functionalities that are being requested by some

other service. If such collaborations are not modelled directly as first-class entities that can be manipulated by a process of dynamic reconfiguration, the overhead that just-in-time integration and other operational aspects of this new architecture represent will not lead to the levels of agility that are required for the paradigm to impose itself.

However, traditionally, interactions in OO are based on *identities*, in the sense that, through clientship, objects interact by invoking the methods of specific objects (instances) to get something specific done. This implies that any unanticipated change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established.

On the contrary, interactions in the service-oriented approach should be based on the design of what needs to be done, thus decoupling the *what one wants to be done* from *who does it*. This translates directly to the familiar characterisation of Web Services as *late binding* or, better, *just-in-time binding*. It is as if collaborations in OO were shifted from instance-to-instance to instance-to-interface, albeit with a more expressive notion of interface. This is why, in our opinion, Web Services are, indeed, beyond OO methodology and technology, especially in what concerns the support that needs to be provided for establishing and managing interactions.

3 Orchestration through Coordination

Web services are granular software components that can be used as building blocks for distributed applications or for the assembly of business processes. They reflect a new service-oriented architectural approach based on the notion of building applications by discovering and orchestrating network-available services, or just-in-time integration of applications. With Web Services, application design becomes a matter of describing the capabilities of network services to offer certain functionalities through their computational capabilities, and describing the orchestration of these collaborators in terms of mechanisms that coordinate their joint behaviour. At runtime, application execution is a matter of translating the collaborator requirements into input for a discovery mechanism, locating a collaborator capable of providing the right service, and superposing the coordination mechanisms that will orchestrate the required interactions.

This informal design makes clear that the algebraic methodology that we have been developing around architectural principles in general [4] plays a fundamental role in enabling service-oriented architectures. This methodology is based on the separation between what in systems is concerned with the computations that are responsible for the functionality of the services that they offer and the mechanisms that coordinate the way components interact, a paradigm that has been developed in the context of so-called Coordination Languages and Models [5]. We brought together concepts and techniques from Software Architectures (the notion of connector [1]), Parallel Program Design (the notion of superposition [6]), and Distributed Systems (techniques for supporting dynamic reconfiguration [7]) that are now integrated in a collection of semantic primitives that support the modelling of systems that are flexible and more amenable to change. These primitives allow, precisely, for the kind of *just-in-time*

binding required by Web Services to be performed in a non-intrusive, compositional way. The technology that we have been promoting supports dynamic configuration, which is exactly what is required for this form of binding.

The underlying methodology of 'coordination-based' development is also essential for service-oriented systems in the sense that it externalises interactions as connectors that can be dynamically, i.e. at run-time, superposed on system components, and encourages developers to identify dependencies between components in terms of *services* rather than identities. The identification of the components to which coordination contracts are applicable is made through 'interfaces' that identify the properties that they need to exhibit rather than the classes to which they have to belong.

4 Algebraic Support

In previous publications [4], we have already shown that the separation between coordination and computation can be materialised through a functor $\mathbf{sig}: \mathbf{DES} \rightarrow \mathbf{SIG}$ mapping (service) designs to signatures, forgetting their computational aspects. We base this separation on a framework consisting of:

- a category \mathbf{DES} of designs (or abstract descriptions) in which systems of interconnected components are modelled through diagrams;
- for every set CD of designs, a set $Conf(CD)$ consisting of all well-formed configurations. Each such configuration is a diagram in \mathbf{DES} that is guaranteed to have a colimit. Typically, $Conf$ is given through a set of rules that govern the interconnection of components in the formalism.

The fact that the computational side does not play any role in the interconnection of systems can be captured by the following properties of the functor \mathbf{sig} :

- \mathbf{sig} is faithful;
- \mathbf{sig} lifts colimits of well-formed configurations;
- \mathbf{sig} has discrete structures;

together with the following condition on the well-formed configuration criterion

given any pair of configuration diagrams $\mathbf{dia}_1, \mathbf{dia}_2$ s.t. $\mathbf{dia}_1; \mathbf{sig} = \mathbf{dia}_2; \mathbf{sig}$, either both are well-formed or both are ill-formed.

The fact that \mathbf{sig} is faithful (i.e., injective over each hom-set) means that morphisms of systems cannot induce more relationships than those that can be established between their underlying signatures.

The requirement that \mathbf{sig} lifts colimits means that, given any well-formed configuration expressed as a diagram $\mathbf{dia}: I \rightarrow \mathbf{DES}$ of designs and colimit $(\mathbf{sig}(S_i) \rightarrow \theta)_{i:I}$ of the underlying diagram of signatures, i.e. of $(\mathbf{dia}; \mathbf{sig})$, there exists a colimit $(S_i \rightarrow S)_{i:I}$ of the diagram \mathbf{dia} of designs whose signature part is the given colimit of signatures, i.e. $\mathbf{sig}(S_i \rightarrow S) = (\mathbf{sig}(S_i) \rightarrow \theta)$. This means that if we interconnect system components through a well-formed configuration, then any colimit of the underlying diagram of signatures establishes an signature for which a computational part exists that captures the joint behaviour of the interconnected components.

The requirement that **sig** has discrete structures means that, for every signature $\theta:SIG$, there exists a design $s(\theta):DES$ such that, for every signature morphism $f:\theta\rightarrow sig(S)$, there is a morphism $g:s(\theta)\rightarrow S$ in **DES** such that $sig(g)=f$. That is to say, every signature θ has a ‘realisation’ (a discrete lift) as a system component $s(\theta)$ in the sense that, using θ to interconnect a component S , which is achieved through a morphism $f:\theta\rightarrow sig(S)$, is tantamount to using $s(\theta)$ through any $g:s(\theta)\rightarrow S$ such that $sig(g)=f$. Notice that, because **sig** is faithful, there is only one such g , which means that f and g are, essentially, the same. That is, sources of morphisms in diagrams of designs are, essentially, signatures.

These properties constitute what we call a *coordinated formalism*. More precisely, we say that **DES** is *coordinated over SIG through the functor sig*. In a coordinated formalism, any interconnection of systems can be established via their signatures, legitimating the use of signatures as channels in configuration diagrams. By requiring that any two configuration diagrams that establish the same interconnections at the level of signatures be either both well-formed or both ill-formed, the fourth property ensures that the criteria for well-formed configurations do not rely on the computational parts of designs.

5 A Simple Example

To illustrate the approach that we started developing in [2] as an extension to the UML [3], consider a very simple example from banking. One of the services that financial institutions have been making available in recent times is what we could call a ‘flexible package’ – the ability to coordinate deposits and debits between two accounts, typically a checking and a savings account, so that the balance of one of the accounts is maintained between an agreed minimal and maximal amount by making automatic transfers to and from the other account. The service that is offered is the detection of the situations in which the transfers are required and their execution in a transactional mode. Because this behaviour should be published as a service that customer applications looking for flexible account management should be able to bind to, the following aspects should be ensured.

One of the services that financial institutions have been making available in recent times is what we could call a ‘flexible package’ – the ability to coordinate deposits and debits between two accounts, typically a checking and a savings account, so that the balance of one of the accounts is maintained between an agreed minimal and maximal amount by making automatic transfers to and from the other account. The service that is offered is the detection of the situations in which the transfers are required and their execution in a transactional mode. Because this behaviour should be published as a service that customer applications looking for flexible account management should be able to bind to, the following aspects should be ensured.

In the first place, the service cannot be offered for specific accounts. It is the binding process that should be responsible for instantiating, at execution time, the service to the relevant components. Nor should it be offered for specific object classes: the service itself should be able to be described independently of the technol-

ogy used by the components to which it will be bound. Instead, it is the find/bind process that should be able to make the adaptation that is needed between the technologies used for implementing the service and the ones used by the components to which it is going to be bound. This adaptation can itself be the subject of an independent publish/find/bind process that takes place at another level of abstraction, or be offered for default combinations by the service itself. This is important to enable the implementation of the service itself to evolve, say in order to take advantage of new technologies, without compromising the bindings that have been made.

Hence, in our approach, the description of the applications to which the service can be bound is made of what we call *coordination interfaces*. For instance, in the case of the flexible package, two coordination interfaces are required: one catering for the account whose balance is going to be managed, typically a checking account, and the other for the *savings* account. The trigger/reaction mode of coordination that our approach supports requires that each coordination interface identifies which events produced during system execution are required to be detected as triggers for the service to react, and which operations must be made available for the reaction to superpose the required effects. The nature of triggers and operations can vary. Typical events that constitute triggers are calls for operations/methods of instance components, and typical operations are those made public through their APIs. Another class of events that we have found useful as triggers is the observation of changes taking place in the system. For such changes to be detected, components must make available methods through which the required observations can be made (something that the mechanism of find/bind must check), and a detection mechanism must be made available in the implementation platform to enable such changes to be effectively monitored (something that is not universally provided).

The two coordination interfaces that we have in mind can be described as follows:

```
coordination interface savings-account
import types money;
operations
  balance():money;
  debit(a:money)  post balance(a) = old balance()-a
  credit(a:money) post balance(a) = old balance()+a
end
```

Notice how the properties of the operations that are required are specified in an abstract way in terms of pre and post-conditions.

```
coordination interface checking-account
import types money;
triggers balance():money;
reactions
  balance():money;
  debit(a:money)  post balance(a) = old balance()-a
  credit(a:money) post balance(a) = old balance()+a
end
```

The difference between the two interfaces lies in the inclusion of the trigger in the checking-account. This is because it is changes on the balance of checking account that will trigger the service to react. More specifically, we are requiring from the

component that will be bound to this interface that it makes changes in the balance to be detected by the flexible-package service.

The second important requirement is that the service itself be described only on the basis of these coordination interfaces and its own operations, and in terms of the properties that it ensures when it is bound to components that comply with the interfaces. This description can be made in terms of what we call a *coordination law*:

```

coordination law flexible-package
interfaces  c:checking-account, s:savings-account
attributes  minimum,maximum:money
coordination
  when c.balance()<minimum
  do    s.debit(min(s.balance(),maximum-c.balance())
        and c.credit(min(s.balance(),maximum-c.balance()))
  when c.balance()>maximum
  do    c.debit(c.balance()-maximum)
        and s.credit(c.balance()-maximum)
end contract

```

Each coordination rule in the law identifies, under *when*, a trigger and, under *do*, the reaction to be performed on occurrence of the trigger. In the cases at hand, these reactions define sets of actions that we call the *synchronisation sets* associated with the rules. As already mentioned, the reactions are executed as transactions, which means that the synchronisation sets are executed atomically. In what concerns the language in which the reactions are defined, we normally use an abstract notation for defining the synchronisation set as above.

When the interfaces are bound to specific components, their behaviour is coordinated by an instance of the law, establishing what we call a *coordination contract*. The behaviour specified through the rules is superposed on the behaviour of the components without requiring the code that implements them to be changed.

References

1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3), 1997, 213-249.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in *UML'99 ñ Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
3. G.Booch, J.Rumbaugh and I.Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley 1998.
4. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a interface?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999.
5. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.
6. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.

7. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", in *4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.
8. *Web Services architecture overview ñ the next stage of evolution for e-business*, September 2000, www-106.ibm.com/developerswork/web/library.

Fully Automatic Adaptation of Software Components Based on Semantic Specifications^{*}

Christian Haack², Brian Howard¹, Allen Stoughton¹, and Joe B. Wells²

¹ Kansas State University

<http://www.cis.ksu.edu/santos/>

² Heriot-Watt University

<http://www.cee.hw.ac.uk/ultra/>

Abstract. We describe the design and methods of a tool that, based on behavioral specifications in interfaces, generates simple adaptation code to overcome incompatibilities between Standard ML modules.

1 Introduction

1.1 The Problem of Unstable Interfaces

The functionality of current software systems crucially depends on the stability of module interfaces. Whereas implementations of interfaces may change, currently deployed software technology requires that the interfaces themselves remain stable because changing an interface without changing all of the clients (possibly around the entire world) results in failure to compile. However, in practice it is often the case that interfaces gradually change and improve. It may, for example, turn out that it is useful to add a few extra parameters to a function in order to make it more generally applicable. Or, it may be more convenient to give a function a different, but isomorphic, type than initially specified. In this paper, we address the issue of unstable interfaces in the context of Standard ML (SML) and propose a language extension that allows changing module interfaces in certain simple but common ways without needing the module clients to be modified.

As a simple example, consider the interface of a module that contains a sorting function for integer lists. Using SML syntax, such an interface looks like this:¹

```
signature SORT = sig val sort : int list -> int list end
```

Suppose that in a later version this sorting function is replaced by a more general function that works on lists of arbitrary type and takes the element ordering as a parameter:

^{*} This work was partially supported by NSF/DARPA grant CCR-9633388, EPSRC grant GR/R 41545/01, EC FP5 grant IST-2001-33477, NATO grant CRG 971607, NSF grants CCR 9988529 and ITR 0113193, Sun Microsystems equipment grant EDUD-7826-990410-US.

¹ Modules are called *structures* in SML and interfaces (the types of modules) are called *signatures*.

```
signature SORT = sig
  val sort : ('a * 'a -> bool) -> 'a list -> 'a list
end
```

In the example, `'a` is a type variable. `sort` is now a polymorphic function, and has all types that can be obtained by instantiating `'a`. Implementations of the more general of these two interfaces can easily be transformed into implementations of the more special one: Just apply the general sorting function to the standard ordering on the integers. However, the SML compiler does not recognize this. Programs that assume the special interface will fail to compile when the interface is replaced by the general one.

One way to ensure compatibility with the rest of the system when an interface is changed is to always keep the existing functions. For example, when a function (and its type specification) is generalized, the older and more specialized version can be kept. However, this method results in redundancy in the interfaces, which is hardly desirable. Therefore, with currently deployed software technology, it is probably better to change the interfaces, rely on the compiler to report the places where the old interface is assumed, and then fix those places by hand to work with the new interface. Such a manual adaptation is not hard for someone familiar with the programming language, but requires a human programmer and is not automated. It is this step of bridging gaps between interfaces that our tool automates. It adapts a software component by synthesizing wrapper code that transforms implementations of its interface into implementations of a different, but closely related, interface.

1.2 The AxML Language for Synthesis Requests

Extending Interfaces by Semantic Specifications. In the design of our tool, we were guided by the following two principles: Firstly, component adaptation should solely be based on information that is explicitly recorded in interfaces. Secondly, the synthesized interface transformations should transform semantically correct implementations into semantically correct implementations. The emphasis in the second principle is on *semantically correct* as opposed to *type correct*. Considering these two principles, it becomes clear that we have to add semantic specifications into interfaces.² For this purpose, we have designed the language AxML (“**A**nother extended **M**L”), an extension of SML with specification axioms.

Specification axioms are certain formulas from predicate logic. In order to convey an idea of how they look, we extend the interface from above by a specification axiom. As basic vocabulary, we use the following atomic predicates:

```
linOrder : ('a * 'a -> bool) -> prop
sorted : ('a * 'a -> bool) -> 'a list -> prop
bagEq : 'a list -> 'a list -> prop
```

² One could view these semantic specifications as part of the type of a module. However, in this paper we use the following terminology: We refer to the traditional SML types as types, whereas we refer to the AxML specification axioms as semantic specifications.

The type `prop` represents the two-element set of truth values, and predicates denote truth-valued functions. The intended meanings of the above atomic predicate symbols are summarized in the following table:

<code>linOrder f</code>		<code>f</code> is a linear order
<code>sorted f xs</code>		<code>xs</code> is sorted with respect to <code>f</code>
<code>bagEq xs ys</code>		<code>xs</code> and <code>ys</code> are equal as multisets

The AxML system has been deliberately designed to never need to know the meaning of these predicate symbols. Instead, component adaptation works correctly for any valid assignment of meanings to predicate symbols.

Using this vocabulary, we now extend the general `SORT` interface by an axiom. `fa` denotes universal quantification, `->` implication and `&` conjunction.

```
signature SORT = sig
  val sort : ('a * 'a -> bool) -> 'a list -> 'a list
  axiom fa f : 'a * 'a -> bool, xs : 'a list =>
    linOrder f -> sorted f (sort f xs) & bagEq xs (sort f xs)
end
```

We say that a module *implements* an interface if it contains all the specified functions with the specified types (*type correctness*), and, in addition, the functions satisfy all the specification axioms that are contained in the interface (*semantic correctness*). AxML verifies type correctness of a module, but does not verify semantic correctness. It is the programmer's responsibility to ensure the semantic correctness of implementations. It is important to understand that although the traditional SML type information appearing in the interface of a module will be verified, our system *deliberately* does not attempt to verify that a module satisfies the semantic specification axioms that appear in its interface. Our adaptation tool preserves semantic correctness *if it already holds*. This differs from recent work in Nuprl [3] where the implementation of an interface must contain a proof for each axiom.

AxML provides mechanisms to introduce atomic predicate symbols, like, for example, `linOrder`, `sorted` and `bagEq`. The extensive use of such predicate symbols in specification formulas, reflects, we think, the natural granularity in which programmers would write informal program documentation: When describing the behavior of a sorting function, a programmer would use the word "sorted" as basic vocabulary, without further explanation. (The detailed explanation of this word may have been given elsewhere, in order to ensure that programmers agree on its meaning.) We hope that a good use of AxML results in specification axioms that closely resemble informal documentation. From this angle, AxML can be viewed as a formal language for program documentation. Writing documentation in a formal language enforces a certain discipline on the documentation. Moreover, formal documentation can be type-checked, and some flaws in the documentation itself can be discovered that way.

Synthesis Requests. It is now time to describe AxML's user interface: The user will supply AxML files. Every SML file is an AxML file, but, in addition,

AxML files can contain semantic specifications in interfaces. Moreover, AxML files can contain so-called *synthesis requests* that request the adaptation of a set of modules to implement a certain interface. An AxML “compiler” will translate AxML files to SML files by simply dropping all semantic specifications and replacing synthesis requests by implementations that have been generated by a synthesis tool. The synthesized code is of a very simple kind. Typically, it will specialize by supplying arguments, curry, uncurry or compose existing functions. Synthesized code is not recursive. Its purpose is to transform modules into similar modules.

Here is an example of a synthesis request. Assume first that `Sort` is a module that implements the second version of the `SORT` interface. Assume also that `Int` is a module that contains a function `leq` of type `int * int -> bool`, and that `Int`’s interface reports that this function is a linear order. Here is the AxML interface `MYSORT` for an integer sort function:

```
signature MYSORT = sig
  val sort : int list -> int list
  axiom fa xs : int list => sorted Int.leq (sort xs) & bagEq xs (sort xs)
end
```

The following AxML declaration contains a synthesis request for an implementation of `MYSORT`. The synthesis request is the statement that is enclosed by curly braces.

```
structure MySort : MYSORT = { fromAxioms structure Sort structure Int
                             create MYSORT }
```

The AxML compiler will replace the synthesis request by an implementation of the interface `MYSORT`:

```
structure MySort : MYSORT = struct val sort = Sort.sort Int.leq end
```

Here is how AxML can be used to protect a software system against certain interface changes: If a module’s interface is still unstable and subject to change, the programmer may use it indirectly by requesting synthesis of the functions that they need from the existing module. Initially, it may even be the case that these functions are exactly contained in the module. (In this case, the “synthesized” functions are merely selected from the existing module.) If the interface of the existing module changes in a way that permits the recovery of the old module by simple transformations, the functions will automatically be resynthesized. In this manner, the AxML compiler overcomes what would be a type incompatibility in currently deployed systems.

The *simple transformations* that AxML will be able to generate consist of expressions that are built from variables by means of function abstraction, function application, record formation and record selection. AxML will deliberately not attempt to generate more complex transformations, because that seems too hard to achieve fully automatically.

Code synthesis of this kind is incomplete. Our synthesis algorithm can be viewed as a well-directed search procedure that interrupts its search after a certain time, possibly missing solutions that way.

```

signature BINREL = sig
  type 'a t = 'a * 'a -> bool
  pred 'a linOrder : 'a t -> prop
  pred 'a linPreorder : 'a t -> prop
  pred 'a preorder : 'a t -> prop
  pred 'a equivalence : 'a t -> prop
  val symmetrize : 'a t -> 'a t
  axiom fa r : 'a t => linOrder r -> linPreorder r
  axiom fa r : 'a t => linPreorder r -> preorder r
  axiom fa r : 'a t => preorder r -> equivalence (symmetrize r)
end

structure BinRel : BINREL = struct
  type 'a t = 'a * 'a -> bool
  fun symmetrize f (x,y) = f(x,y) andalso f(y,x)
end

```

Fig. 1. A structure of binary relations

A Further Benefit: Recognition of Semantic Errors. Because component adaptation is based on behavioral specifications rather than just traditional types, AxML will have another desirable effect: Sometimes a re-implementation of a function changes its behavior but not its traditional type. When this happens, other parts of the system may behave incorrectly because they depended on the old semantics. In traditional languages, such errors will not be reported by the compiler. AxML, on the other hand, fails to synthesize code unless correctness is guaranteed. Therefore, if behavioral changes due to re-implementations are recorded in the interfaces, then AxML will fail to resynthesize requested code if the changes result in incorrect program behavior. In this manner, the AxML compiler reports incompatibilities of behavioral specifications.

2 An Overview of AxML

User-Introduced Predicate Symbols. AxML provides the programmer with the facility to introduce new predicate symbols for use in specification axioms. For example, the structure in Figure 1 introduces predicate symbols and functions concerning binary relations. Predicates may have type parameters. Thus, a predicate symbol does not denote just a single truth-valued function but an entire type-indexed family. For example, `linPreorder` has one type parameter `'a`. In predicate specifications, the list of type parameters must be mentioned explicitly. The general form for *predicate specifications* is

$$\text{pred } (a_1, \dots, a_n) \text{ id} : \text{predty}$$

where a_1, \dots, a_n is a sequence of type variables, *id* is an identifier and *predty* is a predicate type, i.e., a type whose final result type is `prop`. The parentheses that enclose the type variable sequence may be omitted if the sequence consists of a single type variable and must be omitted if the sequence is empty.

When a structure *S* is constrained by a signature that contains a specification of a predicate symbol *id*, a new predicate symbol is introduced into the environ-

ment and can be referred to by the long identifier *S.id*. For instance, the structure declaration in the example introduces the predicate symbols `BinRel.linOrder`, `BinRel.linPreorder`, `BinRel.preorder` and `BinRel.equivalence`.

AxML permits the explicit initialization of type parameters for predicate symbols. The syntax for initializing the type parameters of a predicate symbol *longid* by types ty_1, \dots, ty_n is

$$\textit{longid} \text{ at } (ty_1, \dots, ty_n)$$

The parentheses that enclose the type sequence may be omitted if the sequence consist of a single type. For example, `(BinRel.linPreorder at int)` initializes the type parameter of `BinRel.linPreorder` by the type `int`.

From AxML’s point of view, predicate specifications introduce uninterpreted atomic predicate symbols. AxML synthesis is correct for *any* valid assignment of meanings to predicate symbols. On the other hand, programmers that introduce such symbols should have a concrete interpretation in mind. It is, of course, important that different programmers agree on the interpretation. The interpretations of the predicate symbols from the example are as follows:

`((BinRel.linPreorder at ty) r)` holds if and only if the relation r is a linear preorder at type ty , i.e., it is reflexive, transitive and for every pair of values (x, y) of type ty it is the case that either $r(x, y) = \text{true}$ or $r(y, x) = \text{true}$. Similarly, the interpretation of `BinRel.linOrder` is “is a linear order”, the interpretation of `BinRel.preorder` is “is a preorder”, and the interpretation of `BinRel.equivalence` is “is an equivalence relation”.

Formally, specification predicates are interpreted by sets of closed SML expressions that are closed under observational equality. For instance, it is not allowed to introduce a predicate of type `(int list -> int list) -> prop` that distinguishes between different sorting algorithms. All sorting algorithms are observationally equal, and, therefore, a predicate that is true for merge-sort but false for insertion-sort, call it `isInNlogN`, is not a legal specification predicate. The reader who is interested in a detailed treatment of the semantics of specification formulas is referred to [7].

There is a built-in predicate symbol for observational equality. It has the following type specification:

$$\text{pred 'a == : 'a -> 'a -> prop}$$

In contrast to user-introduced predicate symbols, `==` is an infix operator. The AxML synthesizer currently treats `==` like any other predicate symbol and does not take advantage of particular properties of observational equality.

Specification Axioms. Predicate symbols are turned into *atomic specification formulas* by first initializing their type parameters and then applying them to a sequence of terms. Here, the set of *terms* is a small subset of the set of SML expressions. It consists of all those expressions that can be built from variables and constants, using only function application, function abstraction, record formation and record selection. The set of terms coincides with the set of those

SML expressions that may occur in synthesized transformations. *Specification formulas* are built from atomic specification formulas, using the propositional connectives and universal quantification. For example, if `Int.even` is a predicate symbol of type `int -> prop`, then the following is a well-typed specification formula:

```
fa x : int => Int.even ((fn y => y + y) x)
```

On the other hand, the following is *not* a specification formula, because the argument of `Int.even` is not a term:

```
fa x : int => Int.even (let fun f x = f x in f 0 end)
```

A universally quantified formula is of one of the following three forms:

```
fa id : ty => F           fa id : ([a1, ..., an] . ty) => F           fa a => F
```

where *id* is a value identifier, *ty* is a type, a, a_1, \dots, a_n is a sequence of type or equality type variables and *F* is a formula. The first two forms denote *value quantification* and the third one *type quantification*. The expression $([a_1, \dots, a_n] . ty)$ denotes an SML type scheme, namely, the type scheme of all values that are of type *ty* and polymorphic in the type variables a_1, \dots, a_n . Quantification over polymorphic values is useful, because it allows to express certain theorems — so-called “free” theorems — that hold exactly because of the polymorphism [22]. Value quantification ranges over all *values* of the specified type or type scheme, as opposed to all, possibly non-terminating, closed expressions.

Type parameters of atomic predicate symbols may be explicitly instantiated, but they do not have to be. A formula where type instantiations are left implicit is regarded as a shorthand for the formula that results from inserting the most general type arguments that make the formula well-typed. For example,

```
fa f : 'a * 'a -> bool, xs : 'a list =>
  BinRel.linOrder f ->
  List.sorted f (sort f xs) & List.bagEq xs (sort f xs)
```

is a shorthand for

```
fa f : 'a * 'a -> bool, xs : 'a list =>
  (BinRel.linOrder at 'a) f ->
  (List.sorted at 'a) f (sort f xs) & (List.bagEq at 'a) xs (sort f xs)
```

Free type variables in formulas are implicitly all-quantified at the beginning of the formula. Therefore, the previous formula expands to

```
fa 'a => fa f : 'a * 'a -> bool, xs : 'a list =>
  (BinRel.linOrder at 'a) f ->
  (List.sorted at 'a) f (sort f xs) & (List.bagEq at 'a) xs (sort f xs)
```

```

signature TABLE = sig
  type ('k,'v) t
  val empty : ('k,'v) t
  val update : ('k,'v) t -> 'k -> 'v -> ('k,'v) t
  val lookup : ('k,'v) t -> 'k -> 'v option
  axiom fa x : 'k => lookup empty x == NONE
  axiom fa x : 'k, t : ('k,'v) t, y : 'v =>
    lookup (update t x y) x == SOME y
  axiom fa x1,x2 : 'k, t : ('k,'v) t, y : 'v =>
    Not (x1 == x2) -> lookup (update t x1 y) x2 == lookup t x2
end

```

Fig. 2. An interface of lookup tables

Example 1. The signature in Figure 2 specifies an abstract type t of lookup tables. The type t has two type parameters, namely $'k$, the type of keys, and $'v$, the type of values. Type variables that begin with a double prime range over equality types only. Equality types are types that permit equality tests. For example, the type of integers and products of equality types are equality types, but function types are not. Elements of the datatype $'b$ `option` are either of the form `SOME x` , where x is an element of type $'b$, or they are equal to `NONE`. The keyword `Not` denotes propositional negation.

Synthesis Requests. Synthesis requests are always surrounded by curly braces. They come in three flavors: As requests for structure expressions, functor bindings and structure level declarations. Thus, we have extended the syntax classes of structure expressions, functor bindings and structure-level declarations from the SML grammar [15] by the following three phrases:

$$\begin{aligned}
 \text{strexpr} & ::= \dots \mid \{ \text{createstrexpr} \} \\
 \text{funbind} & ::= \dots \mid \{ \text{createfunbind} \} \\
 \text{strdec} & ::= \dots \mid \{ \text{createstrdec} \}
 \end{aligned}$$

The synthesis requests are of the following forms:

$$\begin{aligned}
 \text{createstrexpr} & ::= \langle \text{fromAxioms axiomset} \rangle \text{ create sigexpr} \\
 \text{createfunbind} & ::= \langle \text{fromAxioms axiomset} \rangle \text{ create funspec} \\
 \text{createstrdec} & ::= \langle \text{fromAxioms axiomset} \rangle \text{ create spec}
 \end{aligned}$$

The parts between the angle brackets $\langle \rangle$ are optional. The syntax domains of signature expressions *sigexpr* and specifications *spec* are the ones from SML, but enriched with axioms and predicate specifications. The domain *funspec* of functor specifications is not present in SML. Functor specifications are of the form

$$\text{funspec} ::= \text{funid} (\text{spec}) : \text{sigexpr}$$

where *funid* is a functor identifier. *Axiom sets* are lists of specification formulas, preceded by the keyword `axiom`, and structure identifiers, preceded by the

```

signature TABLE' = sig
  type ('v,'k) t
  val empty : ('v,'k) t
  val update : ('v,'k) t -> 'k * 'v -> ('v,'k) t
  val lookup : 'k -> ('v,'k) t -> 'v option
  axiom fa x : 'k => lookup x empty == NONE
  axiom fa x : 'k, t : ('v,'k) t, y : 'v =>
    lookup x (update t (x,y)) == SOME y
  axiom fa x1,x2 : 'k, t : ('v,'k) t, y : 'v =>
    Not (x1 == x2) -> lookup x2 (update t (x1,y)) == lookup x2 t
end

```

Fig. 3. Another interface of lookup tables

keyword **structure**. A structure identifier introduces into the axiom set all the specification formulas that are contained in the structure that the identifier refers to. (An axiom is contained in a structure if it is contained in its explicit signature.)

When encountering a synthesis request, the AxML compiler creates a structure expression (respectively functor binding, structure level declaration) that implements the given signature (respectively functor specification, specification). The created implementation of the signature is guaranteed to be semantically correct, *provided* that the environment correctly implements the axioms in *axiomset*. A synthesis may fail for several reasons: Firstly, there may not exist a structure expression that satisfies the given specification. Secondly, a structure expression that satisfies the given specification may exist, but its correctness may not be provable from the assumptions in *axiomset*. Thirdly, a structure expression that satisfies the given specification may exist and its correctness may be provable from the given axiom set, but the synthesizer may not find it due to its incompleteness.

Example 2. Figure 3 shows an interface of lookup tables that differs from the one in Figure 2. The functions in the two signatures differ by type isomorphisms. Moreover, the type parameters of the type t appear in different orders. Here is a request for creation of a functor that transforms TABLE' structures into TABLE structures:

```
functor { create F ( structure T : TABLE' ) : TABLE }
```

Here is the synthesized functor:

```

functor F ( structure T : TABLE' ) : TABLE = struct
  type ('k,'v) t = ('v,'k) T.t
  val empty = T.empty
  val update = fn t => fn x => fn y => T.update t (x,y)
  val lookup = fn t => fn x => T.lookup x t
end

```

Note that this example demonstrates that AxML is capable of synthesizing type definitions.

3 Synthesis Methods

This section gives an overview of the methods. Details can be found in [7]. Our methods and the style of their presentation has been inspired by the description of a λ Prolog interpreter in [14].

3.1 Sequent Problems

Synthesis requests get translated to sequent problems. A *sequent problem* is a triple of the form

$$\mathcal{C}; (\Delta \vdash \Theta)$$

where \mathcal{C} is a problem context, and both Δ and Θ are finite sets of formulas. The *problem context* \mathcal{C} is a list of declarations. A *declaration* assigns types schemes to all value variables, and kinds to all type function variables that occur freely in Δ or Θ . *Kinds* specify the arity of type constructors. Typical kinds are the kind **type** of types, the kind **eqtype** of equality types and type function kinds like **type** \rightarrow **type** or **eqtype** \rightarrow **type** \rightarrow **eqtype**. For example, the kind of type constructor **t** in Figure 2 is **eqtype** \rightarrow **type** \rightarrow **type** (which happens to have the same inhabitants as **type** \rightarrow **type** \rightarrow **type**).

A declaration also tags each variable with either a \forall -symbol or an \exists -symbol. These are just tags and are not to be confused with quantifiers in specification formulas. Variables that are tagged by an \exists are called *existential variables* and variables that are tagged by a \forall are called *universal variables*. Universal variables may be viewed as fixed parameters or constants. Existential variables, on the other hand, are auxiliary variables that ought to be substituted. It is the goal of our methods to replace the existential variables by terms that only contain universal variables. These substitution terms constitute the implementations of the existential variables. The universal variables correspond to preexisting resources that can be used in the synthesis and the existential variables correspond to the values the user has requested to be synthesized. The order of declarations in the variable context \mathcal{C} is important because it encodes scoping constraints: Substitution terms for an existential variable x may only contain universal variables that occur before x in \mathcal{C} .

Example 3. The sequent problem that results from translating the synthesis request from Example 2 has the shape $(\mathcal{C}; (\Delta \vdash \{F\}))$, where Δ is the set of all specification axioms that are contained in signature `TABLE'`, F is the conjunction of all specification axioms that are contained in signature `TABLE`, and \mathcal{C} is the following problem context:

```

 $\forall$ T.t : type  $\rightarrow$  eqtype  $\rightarrow$  type,
 $\forall$ T.empty : ('v, 'k) T.t,
 $\forall$ T.update : ('v, 'k) T.t  $\rightarrow$  'k * 'v  $\rightarrow$  ('v, 'k) T.t,
 $\forall$ T.lookup : 'k  $\rightarrow$  ('v, 'k) T.t  $\rightarrow$  'v option,
 $\exists$ t : eqtype  $\rightarrow$  type  $\rightarrow$  type,
 $\exists$ empty : ('k, 'v) t,
 $\exists$ update : ('k, 'v) t  $\rightarrow$  'k  $\rightarrow$  'v  $\rightarrow$  ('k, 'v) t,
 $\exists$ lookup : ('k, 'v) t  $\rightarrow$  'k  $\rightarrow$  'v option

```

A *sequent* is a sequent problem whose problem context does not declare any existential variables. A sequent $(\mathcal{C}; (\Delta \vdash \Theta))$ is called *valid* iff for all interpretations of its universal variables by SML values, all interpretations of its universal type function variables by SML type definitions and all interpretations of atomic predicate symbols by specification properties, the following statement holds:

If *all* formulas in Δ are true, then *some* formula in Θ is true.

A solution to a sequent problem \mathcal{P} is a substitution s for the existential variables, such that the sequent that results from applying s to \mathcal{P} and removing the existential variables from \mathcal{P} 's problem context is valid. Here is the solution to the sequent problem from Example 3. The reader is invited to compare it to the synthesized functor from Example 2.

$$\begin{array}{ll} \mathbf{t} \mapsto (\lambda k. \lambda v. \mathbf{T.t} \ v \ k) & \mathbf{lookup} \mapsto (\mathbf{fn} \ \mathbf{t} \Rightarrow \mathbf{fn} \ \mathbf{x} \Rightarrow \mathbf{T.lookup} \ \mathbf{x} \ \mathbf{t}) \\ \mathbf{empty} \mapsto \mathbf{T.empty} & \mathbf{update} \mapsto (\mathbf{fn} \ \mathbf{t} \Rightarrow \mathbf{fn} \ \mathbf{x} \Rightarrow \mathbf{fn} \ \mathbf{y} \Rightarrow \mathbf{T.update} \ \mathbf{t} \ (\mathbf{x}, \mathbf{y})) \end{array}$$

It is not hard to recover the SML expressions whose synthesis has been requested, from a solution to the corresponding sequent problem. Therefore, the task that our methods tackle is the search for solutions to a given sequent problem.

3.2 Search

Conceptually, the search can be split into four phases. However, in our implementation these phases interleave.

Phase 1: Goal-Directed Proof Search. The first phase consists of a goal-directed search in a classical logic sequent calculus. Such a search operates on a list of sequent problems — the list of *proof goals* — that share a common problem context. It attempts to find a substitution that simultaneously solves all of the goals. The rules of the sequent calculus decompose a proof goal into a list of new proof goals. At some point of the search process, proof goals are reduced to unification problems. This happens when both sides of a sequent problem contain an atomic formula under the same atomic predicate symbol. In this case, we reduce the problem to a unification problem by equating the types and terms left of \vdash with the corresponding types and terms right of \vdash .

$$\mathcal{C}; \left(\begin{array}{l} \Delta \cup \{(P \text{ at } (ty_1, \dots, ty_k)) \ M_1 \dots M_n\} \\ \vdash \Theta \cup \{(P \text{ at } (ty'_1, \dots, ty'_k)) \ M'_1 \dots M'_n\} \end{array} \right) \rightarrow \mathcal{C}; \left\{ \begin{array}{l} ty_1 = ty'_1 \\ \dots \\ ty_k = ty'_k \\ M_1 = M'_1 \\ \dots \\ M_n = M'_n \end{array} \right\}$$

The system of equations has two parts — a system of *type equations*, and a system of *term equations*. We are looking for a substitution that solves the type equations up to $\beta\eta$ -equality, and the term equations up to observational equality in SML.

Phase 2: Unification up to $\beta\pi$ -Equality. It seems hopeless to come up with a good unification procedure for SML observational equality, even for the very restricted sublanguage of pure terms.³ Instead, we enumerate solutions up to $\beta\pi$ -equality, where π stands for the equational law for record selection. $\beta\pi$ -equality is not sound in a call-by-value language. Therefore, the substitutions that get returned by $\beta\pi$ -unification will later be applied to the original unification problems, and it will be checked whether they are solutions up to observational equality (see Phase 4). For $\beta\pi$ -unification, we use a version of Huet’s pre-unification procedure for simply typed λ -calculus [11]. A number of extensions were necessary to adapt $\beta\pi$ -unification to our language:

First-Order Type Functions. Because our language has first-order type function variables, we need to use higher-order unification at the level of types. Type unification problems that come up in practice are simple and fall into the class of so-called (*higher-order*) *pattern problems* [14]. For these problems, unifiability is decidable. Moreover, these problems have most general unifiers and there is a terminating algorithm that finds them. Our implementation only attempts to solve type unification problems that are pattern problems.⁴ If a type unification problem is not a pattern problem, it is postponed. If it can’t be postponed further, the synthesizer gives up.

The synthesizer never attempts to solve a term disagreement pair if there are still unsolved type disagreement pairs in the system. This way, it avoids an additional source of non-termination that results from the fact that terms in a system with unsolved type disagreement pairs may be non-well-typed and, hence, non-terminating.

Polymorphism. Higher-order unification in the presence of polymorphic universal variables has been described in [16]. In addition to polymorphic universal variables, we also allow polymorphic existential variables. In order to handle them in a good way, we use an explicitly typed intermediate language in the style of Core-XML [8]. For higher-order unification in an explicitly typed language, the presence of type function variables is important. Type function variables facilitate the “raising” of fresh type variables that get introduced into the scope of bound type variables.

Record Types. We treat record types in a similar way as products types are treated in [2].

No Extensionality Rules. Most extensions of higher-order unification to more expressive languages than simply typed λ -calculus assume extensionality rules. Extensionality rules simplify unification both conceptually and computationally. Our unification procedure, however, does not use extensionality rules for functions or records, because only very limited forms are observationally valid in a call-by-value language like SML.

Phase 3: Enumerating Terms of Given Types. After unification, terms may still have free existential variables. These variables need to be replaced by

³ Pure terms are those SML expressions that we permit as arguments for predicates.

⁴ In term unification, though, we also solve non-patterns.

terms that contain universal variables only. This is non-trivial, because the terms must have correct types. We need a procedure that enumerates terms of a given type in a given parameter context. By the Curry-Howard isomorphism, this is achieved by a proof search in a fragment of intuitionistic logic. Because of the presence of polymorphic types, this fragment exceeds propositional logic. Our current experimental implementation uses a very simple, incomplete procedure. Eventually, this procedure will be replaced by an enumeration procedure based on a sequent calculus in the style of [10, 5].

Phase 4: Soundness Check for Observational Equality. After Phase 3, we have found substitutions that solve the original sequent problems up to $\beta\pi$ -equality. However, $\beta\pi$ -equality is not sound in a call-by-value language that has non-termination, let alone side-effects. For example, the following β -equality is not observationally valid, because the left hand side may not terminate for certain interpretations of f by SML expressions:

$$(\lambda x. 0) (f 0) = 0 \tag{1}$$

The following β -equality is valid in functional SML but not in full SML including side effects, because it does not preserve the execution order:

$$(\lambda x. \lambda y. f y x) (g 0) 0 = f 0 (g 0) \tag{2}$$

Because $\beta\pi$ -equality is unsound in SML, the synthesizer applies the substitutions that have been discovered in Phases 1 through 3 to the original unification problems, and checks whether they solve them up to observational equality. For this check, it uses two different procedures depending on which option it has been invoked with. An option for functional SML guarantees correctness of the synthesized code in a purely functional subset of SML. An option for full SML guarantees correctness in full SML including side effects. Both checks are approximative and may reject $\beta\pi$ -solutions unnecessarily. For full SML, the synthesizer uses $\beta_v\pi_v$ -equality (“beta-pi-value-equality”) [17]. $\beta_v\pi_v$ -equality only allows β -reductions if the argument term is a syntactic value. Applications are not syntactic values. Therefore, Equation 1 is not a $\beta_v\pi_v$ -equality. For functional SML, it uses a strengthening of $\beta_v\pi_v$ -equality. Under this stronger equality, certain β -reductions that are not β_v -reductions are allowed in equality proofs. Roughly, β -reductions are allowed if the function parameter gets used in the function body. Equation 2 holds with respect to this stronger equality.

4 Related Work

Language Design. The AxML specification language is inspired by and closely related to EML (**E**xtended **M**L) [12]. In style, AxML specification axioms resemble EML specification axioms. However, EML axioms have some features that complicate the task of automatic component adaptation. For this reason, AxML puts certain restrictions on specification formulas, that are not present in

EML. Most notably, whereas EML identifies specification formulas with expressions of the SML type `bool`, AxML has an additional type `prop` for specification formulas. As a result, AxML specification axioms come from a small, particularly well-behaved language, whereas EML axioms include all SML expressions of type `bool`. In particular, EML axioms may be non-terminating SML expressions, whereas subexpressions of AxML axioms always terminate.

Automatic Retrieval, Adaptation and Composition. A closely related field of research is the use of specifications as search keys for software libraries. In library retrieval, it is generally desirable to search for components that match queries up to a suitable notion of similarity, like for example type isomorphism. Adaptation is necessary to overcome remaining differences between retrieved and expected components. Because human assistance is assumed, semantic correctness of the retrieved components is not considered critical in library retrieval. As a result, much of the research in this field has focused on traditional type specifications [18, 20, 4, 1]. Work on library retrieval that is based on finer semantic specifications than traditional types includes [19, 6, 23]. All of these rely on existing theorem provers to do a large part of the work. Hemer and Lindsay present a general framework for lifting specification matching strategies from the unit level to the module level [9]. The Amphion system [21, 13] is interesting in that it does not only retrieve functions from libraries, but it also composes them. There is a major difference between Amphion and our system: Whereas in Amphion a pre-post-condition specification for a function of type $(\tau \rightarrow \tau')$ is expressed as $(\forall x : \tau. \exists y : \tau'. \text{pre}(x) \Rightarrow \text{post}(x, y))$, it is expressed in our system as $(\exists f : \tau \rightarrow \tau'. \forall x : \tau. \text{pre}(x) \Rightarrow \text{post}(x, fx))$.⁵ As a result, the most important component of our synthesizer is a higher-order unification engine. Amphion, on the other hand, uses a resolution theorem prover for (constructive) first-order logic with equality.

5 Conclusion and Future Directions

We have enriched SML module interfaces by semantic specification axioms. We have assembled methods for automatically recognizing similarities between enriched interfaces and for synthesizing adapters that bridge incompatibilities. The use of uninterpreted predicate symbols prevents specification axioms from getting too detailed, so that our incomplete methods terminate quickly and successfully in many practical cases. The synthesized adapters preserve semantic correctness, as opposed to just type correctness. They may contain higher-order and polymorphic functions as well as parameterized type definitions.

In this paper, we have proposed to apply our methods for protecting evolving software systems against interface changes. We hope that, in the future, our methods can be extended and used in other ambitious applications:

⁵ This is how Amphion represents specifications internally. Amphion has an appealing graphical user interface.

Automatic Composition of Generic Library Functions. Standard libraries of modern programming languages often contain few, but very general, functions that can be composed and specialized in a lot of useful ways. Examples are I/O-primitives in standard libraries for Java or SML. The generality of such functions keeps the libraries both small and powerful. However, it can also make the functions hard to understand and difficult to use correctly. Programmers have to grasp the library functions in their full generality in order to be able to compose the special instances that they need. Even if they know in which modules of a library to look, naive programmers often find it hard to compose the library functions in the right way. Our hope is that, in the future, our methods can be extended to automatically compose library functions, based on simple semantic specifications given by the programmer. This kind of use puts a higher burden on the synthesis algorithms, because it assumes that the naive programmer only understands a limited vocabulary. He possibly uses simpler and less general atomic predicate symbols than the ones that are used in specifications of library functions. As a result, additional axioms are needed so that the synthesis tool can recognize the connection.

Automatic Library Retrieval. An interesting future direction is to extend and use our methods in automatic library retrieval based on semantic specifications. The difficulty is that libraries are large, and, thus, contain a large number of specification axioms. To deal with this difficulty, one could incorporate the AxML synthesizer as a confirmation filter into a filter pipeline, as proposed in [6].

References

- [1] M. V. Aponte, R. D. Cosmo. Type isomorphisms for module signatures. In *8th Int'l Symp. Prog. Lang.: Implem., Logics & Programs, PLILP '96*, vol. 1140 of *LNCS*. Springer-Verlag, 1996. 96
- [2] I. Cervesato, F. Pfenning. Linear higher-order pre-unification. In *Proc. 12th Ann. IEEE Symp. Logic in Comput. Sci.*, 1997. 94
- [3] R. L. Constable, J. Hickey. Nuprl's class theory and its applications. Unpublished, 2000. 85
- [4] R. Di Cosmo. *Isomorphisms of Types: From Lambda-Calculus to Information Retrieval and Language Design*. Birkhäuser, 1995. 96
- [5] R. Dyckhoff, L. Pinto. A permutation-free sequent calculus for intuitionistic logic. Technical Report CS/96/9, University of St Andrews, 1996. 95
- [6] B. Fischer, J. Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, 1997. 96, 97
- [7] C. Haack. *Foundations for a tool for the automatic adaptation of software components based on semantic specifications*. PhD thesis, Kansas State University, 2001. 88, 92
- [8] R. Harper, J. Mitchell. On the type structure of Standard ML. *ACM Trans. on Prog. Lang. and Sys.*, 15(2), 1993. 94
- [9] D. Hemer, P. Lindsay. Specification-based retrieval strategies for module reuse. In *Australian Software Engineering Conference*. IEEE Computer Society Press, 2001. 96

- [10] H. Herbelin. A λ -calculus structure isomorphic to Gentzen-style sequent calculus structure. In *Proc. Conf. Computer Science Logic*, vol. 933 of *LNCS*. Springer-Verlag, 1994. 95
- [11] G. Huet. A unification algorithm for typed λ -calculus. *Theoret. Comput. Sci.*, 1(1), 1975. 94
- [12] S. Kahrs, D. Sannella, A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173, 1997. 95
- [13] M. Lowry, A. Philpot, T. Pressburger, I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, vol. 869 of *LNCS*, Charlotte, 1994. 96
- [14] D. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1(4), 1991. 92, 94
- [15] R. Milner, M. Tofte, R. Harper, D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 90
- [16] T. Nipkow. Higher-order unification, polymorphism and subsorts. In *Proc. 2nd Int. Workshop Conditional & Typed Rewriting System*, LNCS. Springer-Verlag, 1990. 94
- [17] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoret. Comput. Sci.*, 1, 1975. 95
- [18] M. Rittri. Using types as search keys in function libraries. *J. Funct. Programming*, 1(1), 1991. 96
- [19] E. R. Rollins, J. M. Wing. Specifications as search keys for software libraries. In K. Furukawa, ed., *Eighth International Conference on Logic Programming*. MIT Press, 1991. 96
- [20] C. Runciman, I. Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2), 1991. 96
- [21] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *The 12th International Conference on Automated Deduction*, Nancy, France, 1994. 96
- [22] P. Wadler. Theorems for free! In *4th Int. Conf. on Functional Programming Languages and Computer Architecture*, 1989. 89
- [23] A. M. Zaremski, J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Technology*, 6(4), 1997. 96

HASCASL: Towards Integrated Specification and Development of Functional Programs

Lutz Schröder and Till Mossakowski

BISS, Department of Computer Science, Bremen University

Abstract. The development of programs in modern functional languages such as Haskell calls for a wide-spectrum specification formalism that supports the type system of such languages, in particular higher order types, type constructors, and parametric polymorphism, and that contains a functional language as an executable subset in order to facilitate rapid prototyping. We lay out the design of HASCASL, a higher order extension of the algebraic specification language CASL that is geared towards precisely this purpose. Its semantics is tuned to allow program development by specification refinement, while at the same time staying close to the set-theoretic semantics of first order CASL. The number of primitive concepts in the logic has been kept as small as possible; we demonstrate how various extensions to the logic, in particular general recursion, can be formulated within the language itself.

Introduction

In the application of formal methods to the specification and development of correct software, a critical problem is the transition from the level of specifications to the level of programs. In the HASCASL development paradigm introduced here, this problem is tackled as follows: we choose Haskell, a pure functional language, as the primary (but certainly not exclusive) target language, thereby avoiding the need to deal with side effects and other impure features (the fact that ML has such features was one of the primary obstacles in the design of Extended ML [17]). The specification language HASCASL is designed as an extension of the recently completed first-order based algebraic specification language CASL [3,8] with higher-order concepts such as (partial) function types, polymorphism, and type constructors (going beyond and, at times, deviating from [23]). This extended language is a *wide-spectrum language*; in particular, it is powerful enough to admit an executable sublanguage that is in close correspondence to Haskell, thus avoiding the need for a specialized interface language as employed in Larch [14]. At the same time, it is *simple* enough to keep the semantics natural and manageable. In this way, it becomes possible to perform the entire development process, from requirement specifications down to executable prototypes, by successive refinements within a single language.

In more detail, the basic logic of HASCASL consists essentially of the partial λ -calculus [21,22], extended with a form of type class polymorphism that combines features of Haskell and Isabelle [31], and provided with an intensional

Henkin style semantics. The main semantic problem that ensues is the treatment of recursive types and recursive functions, which are not available in classical higher-order logic. The traditional way of dealing with these features is Scott-Strachey denotational semantics [25,32], which needs to talk about notions like complete partial orders (cpo) and least fixed points.

It is indeed possible to add these features to the language without affecting the basic language definition (and hence complicating the semantics), by means of a boot-strap method that consists in writing the relevant extensions as specifications in basic HASCASL. Much of the force of this approach stems from the fact that there is a secondary ‘internal logic’ that lives within the scope of λ -abstractions. This internal logic is by no means another primitive concept, but rather grows naturally out of the *intensional* semantics of partial higher order logic. Through the interaction with the two-valued ‘external’ logic in specifications, the internal logic (which is, in its ‘basic version’, extremely weak) may be observed as well as extended. By means of a suitably extended internal logic, one can then define a type class of cpo in the style of HOLCF [27], so that recursive function definitions become available; similarly, the internal logic allows coding the relevant axioms for recursive datatypes. Thus we arrive at a methodology that allows a smooth transition from abstract requirements to design specifications: the former are specified in *partial* higher order logic (which is not available in HOLCF), while the latter employ cpo and recursion, with undefinedness represented as \perp . In this way, we can, in the requirement specifications, avoid extra looseness arising from the cpo: these are introduced only at the design level, using a fixed set of constructions for generating cpo from recursive datatypes.

The basic logic and design of HASCASL are laid out in Section 1. This is followed by a presentation of the ‘boot-strap’ features, namely, the internal logic, datatypes, and recursion, in Sections 2–4. The corresponding specifications are presented in detail. Section 5 concludes with a brief example that illustrates the HASCASL development methodology. For unexplained categorical terminology, the reader is referred to [1,19].

1 The Basic HASCASL Logic

We shall begin by defining the underlying logic of HASCASL. We will be as economical as possible with primitive concepts; along the way, we will demonstrate how several important features may be added either as built-in syntactic sugar or, non-conservatively, as HASCASL specifications. In order to avoid overburdening the presentation, we shall not be overly fussy about details of the syntax; a full description can be found in [29].

HASCASL is designed as a higher order extension of the first-order algebraic specification language CASL (*Common Algebraic Specification Language*) developed by COFI, the international *Common Framework Initiative for Algebraic Specification and Development* [8]. The features of CASL include first-order logic, partial functions, subsorts, sort generation constraints, and structured and architectural specifications. For the definition of the language cf. [9,3]; a full formal

semantics is laid out in [10]. The semantics of structured and architectural specifications is independent of the logic employed for basic specifications. In order to define our language, it is therefore sufficient to define its logic, i.e. essentially to fix notions of signature, model, sentence, and satisfaction as done below.

Roughly stated, the logic of HASCASL is Moggi's partial λ -calculus [21], equipped with a few semantically harmless extensions such as type constructors and type class polymorphism, and embedded into a full-featured external logic much as the one provided in first-order CASL.

1.1 Signatures and Sentences

The *types* associated to a HASCASL signature are built in the usual way from type constructors of a declared arity (possibly involving type classes; cf. Section 1.3). In particular, there are built-in type constructors $_ * _$, $_ * _ * _$ etc. for product types, $_ \rightarrow ? _$ and $_ \rightarrow _$ for partial and total function types, respectively, and a unit type `unit`. Types may be aliased; see [29] for details. Operators are constants of an assigned type. As in first order CASL, the user may specify a subsort relation, denoted $<$, on the declared sorts; in particular, $s \rightarrow t$ is a subtype of $s \rightarrow ? t$. Since subsorting will not really play a great role in the considerations below, we will mostly omit this feature from the rest of the presentation.

These data determine a notion of term according to the rules given in Figure 1. Notations like $\bar{x} : \bar{s}$ abbreviate sequences, here: $x_1 : s_1, \dots, x_n : s_n$. Explicit projection symbols are absent, since they can be coded by terms such as $\lambda x : s, y : t \bullet x$; here, we will use fixed abbreviations `fst`, `snd`, and `prn` with the obvious meaning. Application is regarded as a built-in operator with the obvious profile, so that no extra typing rule for application (of arbitrary terms) is required. Further built-in operators come with the subtype relation: whenever $s < t$, one has a partial downcast operator `as s` and an elementhood predicate $_ \in s$ on t with the same meaning as in first order CASL. Aside from the partial λ -abstraction of Figure 1, there is a total λ -abstraction $\lambda \bar{x} : \bar{s} \bullet \alpha$, which abbreviates a downcast to the type of total functions. Additional typing rules for total functions will be used in static analysis.

$\frac{x : s \text{ in } \Gamma}{\Gamma \triangleright x : s}$	$\frac{f : s \text{ in } \Sigma}{\Gamma \triangleright f : s}$	$\frac{\Gamma \triangleright \alpha_i : s_i, i = 1, \dots, n}{\Gamma \triangleright (\alpha_1, \dots, \alpha_n) : s_1 * \dots * s_n}$
$\frac{}{\Gamma \triangleright () : \text{unit}}$	$\frac{f : s \rightarrow ? t \text{ in } \Sigma}{\Gamma \triangleright f(\beta) : t}$	$\frac{\Gamma, \bar{x} : \bar{s} \triangleright \alpha : t}{\Gamma \triangleright \lambda \bar{x} : \bar{s} \bullet \alpha : s_1 * \dots * s_n \rightarrow ? t}$

Fig. 1. Term formation and typing rules

Such terms may then be used in formulas of an external logic that offers essentially the same ample features as first order CASL, e.g. strong and existential equality, a definedness predicate, and the usual connectives and quantifiers including disjunction, negation, and existence. This external logic is set apart distinctly from a weaker internal logic to be used within λ -terms; see below.

1.2 Models and Satisfaction

Since HASCASL is meant to extend first order CASL as faithfully as possible, its semantics will necessarily be set-theoretic. There is a broad range of choices available for a set-theoretic notion of model of higher order signatures; the principal options are as follows:

- In a *standard* model, all partial function types $s \rightarrow? t$ are interpreted by the full set of partial functions from the interpretation of s to that of t .
- In an *extensional Henkin* model [15], function types are interpreted by subsets of the full set of functions in such a way that all λ -terms can be interpreted (the latter property is called *comprehension*).
- In an *intensional* Henkin model, function types are interpreted by arbitrary sets equipped with an application operation of the appropriate type. Comprehension is still required; however, the way λ -terms are interpreted is now part of the structure of the model rather than just an existence axiom. Intensional Henkin models are discussed, e.g., in [20,21].

The notion chosen for HASCASL, for reasons of semantics as well as methodology, is that of intensional Henkin models. To begin, moving away from standard models avoids the well-known incompleteness problems. Extensionality carries the disadvantage of destroying the existence of initial models of signatures in a setting with partial functions (see [4] for a simple example); further arguments in favour of intensional models even in the total setting are brought forward in [26]. If, on the other hand, extensionality is for some reason required by the user, it may easily be enforced:

spec EXTENSIONALITY =
forall $a, b : Type; f, g : a \rightarrow? b \bullet (\forall x : a \bullet f(x) = g(x)) \Rightarrow f = g$

From a categorical point of view (see [12], in particular 1.8, 2.3, 3.1, 4.3, and 5.5, on why we think this point of view is relevant), intensional models are attractive because they are in direct correspondence with the natural class of categorical models; cf. Remark 2 below.

Methodologically speaking, intensional models offer the advantage of allowing a more fine-grained analysis of function spaces. In [6], for instance, a model of the language PCF is defined in a category of sequential algorithms, where functions are considered together with an evaluation strategy. This also allows a distinction between various ways in which a function may fail to yield a value while, in an extensional setting, a function may merely be either defined or undefined.

A peculiarity of the intensional approach is that the intended equality of terms has to be explicitly imposed on the models. We therefore give a deduction system for *existential* equality $- \stackrel{e}{=} -$, to be read ‘both sides are defined and equal’, in Figure 2. Deduction takes place in a given context Γ ; the ξ -rule uses reasoning with assumptions, marked by square brackets, in a locally enlarged context. $\text{def } \alpha$ abbreviates $\alpha \stackrel{e}{=} \alpha$. Notations like α/β refer to substitution of (the components of) β for the variables in the context of α . Rule (pr) is meant to work both ways. The rule (ax) serves the purpose of accommodating axioms that take the form of implications $\phi \Rightarrow \psi$ (in an indicated context); such axioms are needed in order to ensure correct behaviour w.r.t. subsorting and overloading, including the determination of the total function type (see [29] for details).

The deduction system given in [21] is quite similar, but takes strong equality (‘if any one side is defined, then so is the other, and both are equal’) as the primitive notion; the two systems are easily seen to have the same deductive strength. For the envisaged models, deduction is sound and complete [21,28]; as shown in [21], this completeness result fails for extensional models (!).

$$\begin{array}{c}
 \text{(var)} \frac{x : s \text{ in } \Gamma}{x \stackrel{e}{=} x} \quad \text{(op)} \frac{f : s \text{ in } \Sigma}{f \stackrel{e}{=} f} \quad \text{(sym)} \frac{\alpha \stackrel{e}{=} \beta}{\beta \stackrel{e}{=} \alpha} \quad \text{(tr)} \frac{\alpha \stackrel{e}{=} \beta \quad \beta \stackrel{e}{=} \gamma}{\alpha \stackrel{e}{=} \gamma} \\
 \\
 \text{(cong)} \frac{\alpha \stackrel{e}{=} \beta : t \quad f : t \rightarrow ?u \text{ in } \Sigma \quad \text{def } f(\alpha)}{f(\alpha) \stackrel{e}{=} f(\beta)} \quad \text{(ax)} \frac{\phi \Rightarrow_{y:t} \psi \text{ axiom} \quad \Gamma \triangleright \alpha : t \quad \phi\alpha; \text{ def } \alpha}{\psi\alpha} \quad \text{(str)} \frac{\text{def } f(\alpha)}{\text{def } \alpha} \\
 \\
 \text{(unit)} \frac{\text{def } \alpha : \text{unit}}{\alpha \stackrel{e}{=} ()} \quad \text{(pr)} \frac{\text{pr}_i(\alpha) \stackrel{e}{=} \text{pr}_i(\beta), i = 1, \dots, n}{\alpha \stackrel{e}{=} \beta : s_1 \times \dots \times s_n} \\
 \\
 \text{(\lambda-def)} \frac{}{\text{def } \lambda \bar{y} : \bar{t} \bullet \alpha} \quad \text{(\beta}_1\text{)} \frac{\text{def}(\alpha\gamma, \gamma)}{(\lambda \bar{y} : \bar{t} \bullet \alpha)(\gamma) \stackrel{e}{=} \alpha\gamma} \quad \text{(\beta}_2\text{)} \frac{\text{def}(\lambda \bar{y} : \bar{t} \bullet \alpha)(\gamma)}{\text{def } \alpha\gamma} \\
 \\
 \text{(\eta)} \frac{\text{def } \alpha : \bar{t} \rightarrow ?u}{\lambda \bar{y} : \bar{t} \bullet \alpha(\bar{y}) \stackrel{e}{=} \alpha} \quad \text{(\xi)} \frac{\begin{array}{c} [\bar{y} : \bar{t}; \text{def } \alpha] \quad [\bar{y} : \bar{t}; \text{def } \beta] \\ \vdots \\ \alpha \stackrel{e}{=} \beta \quad \alpha \stackrel{e}{=} \beta \end{array}}{\lambda \bar{y} : \bar{t} \bullet \alpha \stackrel{e}{=} \lambda \bar{y} : \bar{t} \bullet \beta}
 \end{array}$$

Fig. 2. Deduction rules for existential equality in context Γ

Definition 1. A *model* of a given HASCASL signature is an assignment of a set M_s to each type s , in such a way that unit is interpreted as a singleton set and product types are interpreted as cartesian products, together with an assignment of a partial interpretation function

$$M_{s_1} \times \cdots \times M_{s_n} \rightarrow? M_t$$

to each term of type t in context $(x_1 : s_1, \dots, x_n : s_n)$. These interpretation functions are required to respect deducible equality of terms according to Figure 2 (extended by the above-mentioned subsorting axioms). Moreover, substitution must be modeled as composition of partial functions, and terms of the form $x_1 : s_1, \dots, x_n : s_n \triangleright x_i : s_i$ must be interpreted by the appropriate product projections.

A *model morphism* between two such models is a family of functions h_s , where s ranges over all types, that satisfies the homomorphism condition w.r.t. all interpretation functions for terms. Finally, *satisfaction* of formulae (of the external logic, cf. Section 1.1) in such models is defined in the obvious (classical) way.

Remark 2. Intensional Henkin models as defined above are in direct correspondence to categorical interpretations in so-called partial cartesian closed categories (pccc), i.e. essentially categories that have ‘partial function space objects’ representing partial morphisms in much the same way as function spaces in a cartesian closed category represent (total) morphisms (recall that a partial morphism is a span of the form $\bullet \xleftarrow{f} \bullet \xrightarrow{m} \bullet$, where m belongs to a given distinguished class of monomorphisms that are admissible as ‘domains’); see [21] for a detailed definition. Equivalence results for partial λ -calculi on the one hand and pcccs on the other hand are discussed in [28]. An interpretation in a pccc \mathbf{C} gives rise to an intensional Henkin-model by composing the interpretation with the representable functor $hom_{\mathbf{C}}(1, _)$, where 1 denotes the terminal object, and this process can be reversed; this fact is mentioned for the total case in [7].

1.3 Type Classes and Polymorphism

On top of the syntax given so far, we can now add a type class oriented form of shallow polymorphism without really affecting the semantics in an essential way. Besides improving the usability of the language as such, this also takes us closer to actually incorporating most of Haskell as a sublanguage.

As in Isabelle [31], we regard *type classes* as subsets of the set of all types. Such a type class is declared by writing

class Cl or **class** $Cl1 < Cl2$ or **class** $Cl = CT$,

where $Cl2$ is an existing class of which the class $Cl1$ is declared to be a subclass, and CT is a *class term*. Class terms can be either class names, intersections of classes denoted as comma-separated lists, or built-in classes. Built-in classes are the class *Type* of all types and the downsets $\{a \bullet a < t\}$ of given types t under the subtyping relation. Constants can be declared as polymorphic over

type classes, by giving a type scheme with variables universally quantified over classes:

$$\mathbf{op} \ f : \forall a : Cl \bullet \tau,$$

where τ is a type that may contain the type variable a (the type variable may also be declared globally or locally by means of the keywords **var** and **forall**). This declaration has the effect of adding, for each type t of the class Cl , an instance $f[t] : \tau[a/t]$ to the signature (at this point, the mechanism is different from Isabelle, where operators are instantiated for *all* types — this is undesirable in situations where the model theory matters). For the sake of readability, we will mostly omit the explicit type information.

Similarly, axioms may be enclosed in a universal quantification over type variables with assigned classes:

$$\bullet \forall a : Cl \bullet \phi.$$

Such axioms are taken to mean an infinite collection of instances, one for each type. Quantification over types is only allowed at the outermost level of axioms.

Instances for a type class are produced by declaring arities for a type constructor in much the same sense as in Isabelle, stating that the result type belongs to a certain class if the argument types belong to certain other classes:

$$\mathbf{type} \ F : Cl_1 \rightarrow \dots Cl_n \rightarrow Cl.$$

In principle, one could leave it at this, except that one will often want to express the fact that axioms associated to the class are implied by the ones for the instance. To this end, one may write a list of axioms and constant declarations in grouping brackets directly after the declaration of the class, and then mark subclass declarations and type constructor declarations with the keyword **instance**. For example, part of a specification of a type class with a fixed idempotent endoprojection might look as follows:

```

class Proj
  {var a : Proj
   op pr : a → ?a
   • pr = λ x : a • pr(pr(x))}
type instance _ × _ : Proj → Proj → Proj
forall a, b : Proj
  • pr[a × b] = λ x : a, y : b • (pr(x), pr(y))
    
```

This declaration implicitly produces a proof obligation, similar to the **%implies**-annotation in CASL, which states that all axioms explicitly attached to the class *Proj* (only one in the example), instantiated for $a \times b$, are, similarly as in Isabelle/HOL, meant to follow from the axioms of the specification surrounding the instance, including the instantiated axioms for the arguments a and b .

Remark 3. The polymorphism introduced above is essentially ML-polymorphism (except that a **let**-construct is missing, which can, however, be regarded as syntactical sugar). The discourse in [11] may create the impression that the combination of ML-polymorphism and higher order logic is inconsistent. However, this is not the case: as demonstrated above, shallow polymorphism can be ‘coded away’ by just replacing polymorphic operators and axioms by all their

instances. The derivation of Girard’s paradox in [11], Section 5, is based on the assumption that terms of the language are identified up to untyped β -equality in the absence of type annotations; such an equality is obviously unsound w.r.t. the usual notions of model, and the paradox shows that a language with such an equality is inconsistent. When, as in the usual versions of ML-polymorphism, instantiations of polymorphic constants are internally annotated with their types, the contradiction disappears.

1.4 Predicates and Non-strict Functions

Two features appear to be still missing in the type system: on the one hand, predicates, which are provided in CASL, and on the other hand, non-strict functions, which are a feature of Haskell. However, these features can be regarded as syntactical sugar in the setting built up so far:

HASCASL, like CASL and ML, is *strict*, i.e. undefined arguments always yield undefined values, while Haskell functions are allowed to leave arguments unevaluated and thus yield results even on ‘undefined’ arguments. It is well-known that such *non-strict* functions may be emulated by means of function types $\text{unit} \rightarrow ?t$ as argument types (‘proceduring’); we support this concept as follows:

- The type $\text{unit} \rightarrow ?t$ is abbreviated as $?t$.
- There are two extra typing rules: a function that expects an argument of type t may be applied to a term α of type $?t$, which is then automatically replaced by $\alpha()$; conversely, a function that expects an argument of type $?t$ accepts arguments β of type t , which are automatically replaced by $\lambda x : \text{unit} \bullet \beta$.

(N.B.: for algebraic reasons, the type $s \rightarrow ?t$ is not the same as the type $s \rightarrow (?t)$, the use of which is expressly discouraged.)

Predicates are represented as partial functions into unit . The idea here is that definedness of such partial functions corresponds to satisfaction of predicates. For types of the form $t \rightarrow ?\text{unit}$, the abbreviation $\text{pred}(t)$ is provided. The definedness predicate def of first order CASL is interpreted, for each type s , as an abbreviation for $\lambda x : s \bullet ()$ (thanks to strictness, this has the expected behaviour). There is no direct way to use logical connectives in predicate λ -terms. However, universal conjunctive logic is available: the type $?\text{unit}$ can be regarded as a type of truth values, with truth, written tt , abbreviating the term $\lambda x : \text{unit} \bullet ()$ (of course, this is really an instance of def). The conjunction operator $_and_ : ?\text{unit} * ?\text{unit} \rightarrow ?\text{unit}$ is $\lambda x, y : ?\text{unit} \bullet ()$. Finally, the universal quantifier $\text{all} : \text{pred}(\text{pred}(t))$ is just the predicate $_ \in t \rightarrow \text{unit}$; falsity can then be coded as the element $ff = \lambda x : \text{unit} \bullet \text{all}(\lambda y : ?\text{unit} \bullet y)$ of $?\text{unit}$. The introduction of further logical operators is possible, but non-conservative; see Section 2.

2 Internal Equality and the Internal Logic

In basic HASCASL, it is specifically forbidden to use the equality symbol within λ -terms; however, equality can be sneaked back in by means of an *internal equality*. A predicate

$$eq : \forall a \bullet \text{pred}(a * a)$$

is called an internal equality (see also [21]) if $eq(x, y)$ is equivalent to $x \stackrel{e}{=} y$ in the deduction system of Figure 2 (due to intensionality, this is a stronger property than equivalence of the two formulas for each pair (x, y) of elements of a in a model).

In fact, internal equality can be specified in HASCASL. Interestingly, the introduction of internal equality turns out to be highly non-conservative, since it makes the logic available within λ -abstracted predicates substantially richer: besides the universal conjunctive logic of Section 1.4, one can define implication, disjunction, negation, and existential quantification as in [18]. The specification of internal equality and the new connectives is given in Figure 3. In order to improve readability, the equality symbol $\stackrel{e}{=}$ can, after all, be used within λ -terms, but is, then, implicitly replaced by eq . The CASL annotation **%def** indicates a definitional extension, i.e. an extension that induces a bijection of model classes. Similarly, **%implies** precedes axioms that are logical consequences of the previous axioms. It may come as a surprise that the last formula shown in Figure 3 expresses a form of extensionality; however, it is well-known that all categorical models are internally extensional [20].

The internal logic is intuitionistic: there may be more than two truth values, and $neg(neg(A))$ is in general different from A . The obvious deduction rules can be proved as lemmas; e.g., it is not hard to show that the rule

$$\frac{\phi \text{ impl } \psi; \quad \phi}{\psi}$$

is derivable from the rules in Figure 2 and the definitions in Figure 3. The *external* logic (cf. Section 1.1) remains classical: as soon as a predicate appears as an atomic formula, all internal truth values except tt are collapsed into the external *false*. Under (external) extensionality (cf. Section 1.2), the internal logic becomes *almost* classical in the sense that there are *at most* two truth values — one may still have $tt = ff$, which implies that all sorts are singletons. Of course, one can explicitly require $tt \neq ff$.

3 Recursive Datatypes

In order to actually represent functional programs in HASCASL, recursive datatypes are an indispensable feature. As in first order CASL, recursive data types are defined by means of the keyword **type** (cf. [9]), which may be qualified by a preceding **free** or **generated**; a type declaration defines constructors and — optionally — selectors for the declared type. The **generated** constraint introduces an induction axiom; intuitively, this means that the type is term-generated

```

spec INTERNALLOGIC =
  forall  $a : Type$ 
  op  $eq : pred(a \times a)$ 
    •  $\lambda x : a \bullet eq(x, x) = \lambda x : a \bullet tt$ 
    •  $\lambda x, y : a \bullet fst(x, eq(x, y)) = \lambda x, y : a \bullet fst(y, eq(x, y))$ 
  then %def
    forall  $a : Type$ 
    ops  $\_impl\_, \_or\_ : pred(?unit \times ?unit)$ 
       $neg : pred(?unit)$ 
       $ex : pred(pred(a))$ 
    •  $\_impl\_ = \lambda x, y : ?unit \bullet eq(x, x \text{ and } y)$ 
    •  $\_or\_ = \lambda x, y : ?unit \bullet all(\lambda r : ?unit \bullet ((x \text{ impl } r) \text{ and } (y \text{ impl } r)) \text{ impl } r)$ 
    •  $neg = \lambda x : ?unit \bullet x \text{ impl } ff()$ 
    •  $ex(a) = \lambda p : pred(a) \bullet all(\lambda r : ?unit \bullet all(\lambda x : a \bullet p(x) \text{ impl } r)) \text{ impl } r$ 
  then %implies
    forall  $a, b : Type$ 
    •  $all(\lambda f, g : a \rightarrow ? b \bullet all(\lambda x : a \bullet eq[?b](f(x), g(x))) \text{ impl } eq(f, g))$ 

```

Fig. 3. Specification of the internal logic

(‘no junk’). The **free** constraint additionally produces a case operator, which means that the terms are kept distinct (‘no confusion’).

The automatically generated operators and axioms for a free datatype are shown in Figure 4; the notation assumes that the type declaration was of the form

$$\mathbf{free\ type\ } t ::= C_1(t_{11}; \dots; t_{1k_1}) \mid \dots \mid C_n(t_{n1}; \dots; t_{nk_n})$$

where the t_{ij} are either non-recursive or equal to t (the generalization to nested or mutually recursive types does not present great additional difficulties). The axioms make use of the internal logic as introduced in Figure 3. Case operators take one argument x of the defined type and one function argument for each constructor, to be chosen depending on which constructor produced x .

Technically, we do not impose any restrictions on the types that appear in the recursion. However, in the case of recursion on the left hand side of the function arrow, the usual Russell-type paradoxes appear. Consider, e.g., the type

$$\mathbf{type\ } L ::= \mathit{abs}(\mathit{rep} : L \rightarrow ? L)$$

axiomatizing the untyped partial λ -calculus. In the presence of internal equality, negation is available in λ -terms, so that we have the term

$$\lambda x : L \bullet \mathit{fst}(x, \mathit{not\ def\ rep}(x)(x))$$

that produces Russell’s contradiction. A contradiction, in this case, means that all predicates are true, so that all types have at most one element. On the

```

• all(λ p : pred(t) •
  (all(λ x11 : t11, ..., x1k1 : tnkn •
    (p(x1i1) and ... and p(x1il)) impl p(C1(x11, ..., x1k1)))
    %% where i1, ..., il are the recursive occurrences of t in C1
    and ...)
  impl all(p)                                %(t_induction)%
var a : Type
op case : (t11 × ... × t1k1 →? a) × ... → (t →? a)
• all(λ f1 : t11 × ... × t1k1 →? a; ...; fn : tn1 × ... × tnkn →? a, x : t •
  (λ x11 : t11; ...; x1k1 : t1k1 • case(f1, ..., fn, C1(x11, ..., x1k1)))  $\stackrel{e}{=} f_1$ 
  and ... %% one equation for each constructor
  )                                            %(t_case_def)%
    
```

Fig. 4. Automatically generated axioms for free datatypes

other hand, the specification has non-trivial models in plain HASCASL *without* internal equality since the corresponding domain equation can be solved, e.g., in the partial cartesian closed category of pointed cpos [25].

Remark 4. Interestingly, the type

$$\mathbf{type} \ U ::= \mathit{abs}(\mathit{rep} : U \rightarrow U)$$

required for the *total* untyped λ -calculus does have non-trivial models even in the presence of internal equality: take a cartesian closed category \mathbf{C} with reflexive object U . The functor category $\mathbf{Set}^{\mathbf{C}^{op}}$ is a topos and as such gives rise to a Henkin model with internal equality. Moreover, the Yoneda embedding $Y : \mathbf{C} \hookrightarrow \mathbf{Set}^{\mathbf{C}^{op}}$ preserves the cartesian closed structure [30], so that $Y(U)$ is a reflexive object and hence provides us with a (non-extensional) model of the above type.

4 Recursive Functions

One problem of a Henkin-semantics for function types (intensional or extensional) is that even primitive recursive function definitions will not in general constitute definitional extensions, since the relevant functions may, in some models, not be present in the interpretation of the function type. It is possible, however, to impose a cpo structure on the relevant types, thus ensuring that all recursive functions actually live in the function type. Just as in the case of the internal logic, we can avoid an actual extension of the language; instead, we introduce the cpo structure by means of suitable specifications, building on the specification of internal equality (cf. Section 2). The overall concept is closely related to that of HOLCF [27]; the crucial difference is that the surrounding logic is (unlike HOL) partial.

The specification of the cpo structure and the fixed point operator is given in Figure 5. NAT is a specification of the natural numbers with a sort nat , operations $0 : nat$ and $Suc : nat \rightarrow nat$, and the usual axioms including induction and *primitive* recursion (which does not require the cpo structure for its definition). We introduce type classes Cpo and $Cppo$ of cpos and cpos with bottom, respectively, with generic instantiations that extend the ordering to products and partial and total continuous function types $a \xrightarrow{c} ? b$ and $a \xrightarrow{c} b$; the subclass $Flatcpo$ restricts the order to be equality. The continuous function types are subtypes of the built-in function types (due to intensionality, the given definitions of the elementhood predicates as λ -terms are necessary in order to determine the subsort uniquely); partial continuous functions are required to have Scott open domains. Elements of function types are compared pointwise, and elements of product types are compared componentwise. Now, we can introduce a least fixed point operator Y ; using this operator, we define a polymorphic undefined constant.

Of course, we cannot expect any of this to be conservative even over the internal logic. However, there are obviously no problems concerning consistency: in standard models, orderings of partial function spaces are cpos with bottom.

A further instance of the class Cpo is a free datatype t that has constructor arguments of class Cpo . The instance can be automatically generated; the generation process is invoked by means of the keyword **deriving** borrowed from Haskell. If t has constructors C_i as in Section 3, we obtain

type instance $t : Cpo$

- $\leq [t] = \lambda x, y : t \bullet$

$$\begin{aligned} & (case\ x\ of\ C_1(x_1, \dots, x_{k_1}) \rightarrow \\ & \quad (case\ y\ of\ C_1(y_1, \dots, y_{k_1}) \rightarrow x_1 \leq y_1\ and\ \dots\ and\ x_{k_1} \leq y_{k_1}; \\ & \quad \quad C_2(y_1, \dots, y_{k_2}) \rightarrow ff; \\ & \quad \quad \dots) \\ & \dots) \end{aligned}$$

(where we use built-in syntactical sugar for the case operation), i.e. applications of different constructors are incomparable, while applications of the same constructor are compared argument-wise. There is no circularity here: the definition of the ordering is recursive, but does not use the fixed point operator. Rather, it imposes a particular equation on the ordering, and this equation determines the ordering uniquely thanks to the induction axiom of Figure 4. It is easy to see that the case operation, restricted to continuous arguments, is continuous w.r.t. this ordering, and hence can be used in definitions of recursive functions.

Actual recursive definitions will be expressions that involve Y and a partial downcast to the total continuous function type. As long as operators are given the right types, such expressions actually denote functions: call a term α in context $(\bar{x} : \bar{s})$ that has a type of class Cpo *continuous* if $\lambda \bar{x} : \bar{s} \bullet \alpha$ is continuous (cf. Figure 5). Moreover, call a type a *cpo-type* if it is built from basic sorts and type variables of class Cpo by means of the instance declarations for type constructors given in Figure 5 (in particular, cpo-types are of class Cpo). Then we have

```

spec RECURSION = INTERNALLOGIC then NAT then
  class Cpo
    { var a : Cpo
      ops  $\_ \leq \_ : \text{pred}(a \times a)$ 
          isChain :  $\text{pred}(\text{nat} \rightarrow a)$ 
          isBound :  $\text{pred}(a \times (\text{nat} \rightarrow a))$ 
          sup :  $(\text{nat} \rightarrow a) \rightarrow? a$ 
          •  $\text{all}(\lambda x : a \bullet x \leq x)$ 
          •  $\text{all}(\lambda x, y, z : a \bullet (x \leq y \text{ and } y \leq z) \text{ impl } (x \leq z))$ 
          •  $\text{all}(\lambda x, y : a \bullet (x \leq y \text{ and } y \leq x) \text{ impl } x \stackrel{e}{=} y)$ 
          •  $\text{isChain}(a) = \lambda s : \text{nat} \rightarrow a \bullet \text{all}(\lambda n : \text{nat} \bullet s(n) \leq s(\text{Suc}(n)))$ 
          •  $\text{isBound}(a) = \lambda x : a, s : \text{nat} \rightarrow a \bullet \text{all}(\lambda n : \text{nat} \bullet s(n) \leq x)$ 
          •  $\text{all}(\lambda s : \text{nat} \rightarrow a \bullet \text{def } \text{sup}(s) \text{ impl } (\text{isBound}(\text{sup}(s), s) \text{ and } \text{all}(\lambda x : a \bullet \text{isBound}(x, s) \text{ impl } \text{sup}(s) \leq x)))$ 
          •  $\text{all}(\lambda s : \text{nat} \rightarrow a \bullet \text{isChain}(s) \text{ impl } \text{def } \text{sup}(s))$  }
  class Cppo < Cpo
    { var a : Cppo
      op bottom : a
          •  $\text{all}(\lambda x : a \bullet \text{bottom} \leq x)$  }
  class instance Flatcpo < Cpo
    •  $\forall c : \text{Flatcpo} \bullet \_ \leq \_ [a] = \text{eq}[a]$ 
  vars a, b : Cpo; c : Cppo
  type instance a × b : Cpo
    •  $\_ \leq \_ [a \times b] = \lambda x, y : a \times b \bullet \text{fst}(x) \leq \text{fst}(y) \text{ and } \text{snd}(x) \leq \text{snd}(y)$ 
  type instance  $\_ \times \_ : \text{Cppo} \rightarrow \text{Cppo} \rightarrow \text{Cppo}$ 
  type instance unit : Cppo
    •  $() \leq ()$ 
  type  $a \xrightarrow{c} b < a \rightarrow? b$ 
    •  $\_ \in (a \xrightarrow{c} b = \lambda f : a \rightarrow? b \bullet$ 
         $\text{all}(\lambda x, y : a \bullet (\text{def } f(x) \text{ and } x \leq y) \text{ impl } \text{def } f(y)) \text{ and}$ 
         $\text{all}(\lambda s : \text{nat} \rightarrow a \bullet (\text{isChain}(s) \text{ and } \text{def } f(\text{sup}(s))) \text{ impl } \text{ex}(\lambda m : \text{nat} \bullet$ 
         $\text{def } f(s(m)) \text{ and } \text{sup}(\lambda n : \text{nat} \bullet !f(s(n + m)))) \stackrel{e}{=} f(\text{sup}(s))))$ 
  type  $a \xrightarrow{c} b < a \xrightarrow{c} b$ 
    •  $\_ \in (a \xrightarrow{c} b) = \lambda f : a \xrightarrow{c} b \bullet f \in a \rightarrow b$ 
  type instance  $a \xrightarrow{c} b : \text{Cppo}$ 
    •  $\_ \leq \_ [a \xrightarrow{c} b] = \lambda f, g : a \xrightarrow{c} b \bullet \text{all}(\lambda x : a \bullet \text{def } f(x) \text{ impl } f(x) \leq g(x))$ 
  type instance  $a \xrightarrow{c} b : \text{Cpo}$ 
    •  $\_ \leq \_ [a \xrightarrow{c} b] = \lambda f, g : a \xrightarrow{c} b \bullet f \leq [a \xrightarrow{c} b] g$ 
  type instance  $a \xrightarrow{c} c : \text{Cppo}$ 
  then %def
    op Y :  $(c \xrightarrow{c} c) \xrightarrow{c} c$ 
        •  $\text{all}(\lambda f : c \xrightarrow{c} c \bullet f(Y(f)) \stackrel{e}{=} Y(f) \text{ and}$ 
             $\text{all}(\lambda x : c \bullet \text{eq}(f(x), x) \text{ impl } Y(f) \leq x))$ 
    op undefined :  $\text{unit} \xrightarrow{c} c = Y(\lambda x : \text{unit} \xrightarrow{c} c \bullet !x)$ 
    
```

Fig. 5. Specification of the cpo structure and the fixed point operator

Proposition 5. *If, for $\Gamma \triangleright \alpha : t$, all operator constants (besides application) that occur in α , as well as the variables in Γ , have cpo-types, and t is a cpo-type, then α is continuous.*

As a consequence, λ -abstractions of terms α as in the proposition are continuous and hence possess a least fixed point. Note that the fixed point operator itself is of a cpo-type, provided that its parameter a is instantiated with a cpo-type.

Remark 6. A rather different approach to the recursive function problem is pursued in synthetic domain theory (SDT) [16]: the type of propositions (given as the truth value object in a topos) and the ‘domain classifier’ $?unit$ are distinct, so that one can impose the requirement that *all* functions are continuous, w.r.t. a somewhat differently defined ordering, which in general fails to be antisymmetric (e.g., the order relation on the type of propositions is indiscrete). In the more simplistic view presented above, the internal equality necessarily fails to be even monotone. The SDT approach can be put on top of HASCASL instead of internal equality as defined above; the methodological effects of this need examination.

In the specification language Spectrum [13], the continuity problem is dealt with in a different fashion: elements of function types are required to be continuous; non-continuous operators (e.g., predicates) are admitted, but cannot be abstracted. In particular, there is no real logic available within λ -terms: one can use a type of booleans, but for this type, negation has a fixed point. A further, similar approach is found in [2], where there is a two-layer type system with two λ -abstractions used in programs and specifications, respectively.

5 From HASCASL Specifications to Haskell Programs — An Example

Consider the following requirement specification of finite maps from keys to elements, following a module of the library of the Glasgow Haskell compiler:

```

spec FINITEMAP = BOOL then
type FinMap : type  $\rightarrow$  type  $\rightarrow$  type
forall key : Eq; elt, elt1, elt2 : Type
ops empty : FinMap key elt;
      add : FinMap key elt  $\rightarrow$  key  $\rightarrow$  elt  $\rightarrow$  FinMap key elt;
      lookup : FinMap key elt  $\rightarrow$  key  $\rightarrow$ ? elt;
      map : (key  $\rightarrow$  elt1  $\rightarrow$  elt2)  $\rightarrow$  FinMap key elt1  $\rightarrow$  FinMap key elt2
forall m : FinMap key elt; e : elt; k : key
  •  $\neg$ def lookup empty k
  • k1 == k2 = True  $\Rightarrow$  lookup (add m k1 e) k2 = e
  • k1 == k2 = False  $\Rightarrow$  lookup (add m k1 e) k2 = lookup m k2
forall f : key  $\rightarrow$  elt1  $\rightarrow$  elt2; m1 : FinMap key elt1; k : key
  • lookup (map f m1) k = f k (lookup m1 k)

```

Here, `BOOL` is a specification of a datatype `Bool` of booleans with constants `True` and `False`, as well as a type class `Eq`, corresponding to the Haskell type class of the same name, with a boolean-valued equality function `==`. After several refinement steps, one arrives, e.g., at the following design specification that implements maps as association lists (not necessarily in the most effective way):

```

spec ASSOCLIST = BOOL then
vars a, elt, elt1, elt2 : Cpo; key : Cpo, Eq
free type List a ::= [] | _ :: _(a; List a) deriving Cpo
type AList key elt := List (key × elt)
program
  empty : AList key elt = []
  add (m : AList key elt) (k : key) (e : elt) :? AList key elt = (k, e) :: m
  lookup ([] : AList key elt) (k : key) :? elt = undefined()
  lookup ((k1, e1) :: m1) k = if k == k1 then e1 else lookup m1 k
  map (f : key → elt1 → elt2) ([] : AList key elt1) :? AList key elt2 = []
  map f ((k1, e1) :: m1) = (k1, f k1 e1) :: map f m1
    
```

The **program** block declares operations and simultaneously provides recursive definitions for them using the pattern matching notation known from most functional programming languages. These definitions are internally coded by the *case* operator of Section 3 and the fixed point operator *Y* of Section 4 in the obvious way. The specification is executable, but *not* monomorphic in the sense that it has, up to isomorphism, a unique model: with a Henkin-style semantics, intensional or extensional, it is generally infeasible (though perhaps not impossible) to eliminate the looseness inherent in the interpretation of function types.

Note in particular that types that are of class *Type* in `FINITEMAP` are refined by types of class *Cpo* in `ASSOCLIST`. The refinement can be expressed by means of the standard CASL structuring features:

```

view IMPLEMENTFM: FINITEMAP to ASSOCLIST =
  type FinMap ↦ AList
    
```

The transition from *cpo*-types to standard types is achieved by simply forgetting the *cpo*-structure.

`ASSOCLIST` has, apart from syntax issues, the form of a Haskell program. In fact, a large subset of Haskell can be directly translated to `HASCASL` in the style of this example and is thereby provided with a denotational semantics. We expect that this denotational semantics will turn out to be compatible with the de facto operational semantics given by the existing Haskell compilers.

6 Conclusions and Future Work

We propose to use `HASCASL` for the specification and development of functional programs, in particular Haskell programs. `HASCASL` faithfully extends `CASL` to

intensional partial higher-order logic with type class oriented shallow polymorphism. Its semantics is straightforward as well as flexible. In particular, we have shown that it is possible to extend the internal logic of λ -terms by means of specifications written in HASCASL in such a way that, in turn, recursive datatypes and a HOLCF-style fixed point theory become specifiable; non-continuous function types are retained for purposes of requirement specifications. We have ended up with a logic that allows one to write functional programs within the specification language itself. Thus, there is no need for mediating logic morphisms or interface logics between specifications and programs.

Future work will partly concern the inclusion of further programming language features in HASCASL. This concerns in particular features of the Glasgow extensions for Haskell, foremost among them existential types, provided that it can be established that this does not lead to inconsistencies (such as the ones arising from full System F polymorphism; cf. [11]). Further, possibly easier, extensions concern multiparameter type classes and constructor classes. Moreover, the semantic properties of the obvious embedding of first order CASL into HASCASL need elaboration, in particular w.r.t. the CASL structuring operations.

As in the case of first order CASL [24], proof support for HASCASL will be supplied by means of an encoding in Isabelle/HOL, with special attention paid to the coding of the intensional function types. In a subsequent step, we aim at an integration of the corresponding tools for both languages into the MAYA environment [5], which provides a management of proof obligations for structured specifications and a management of change (for both specifications and programs) in an evolutionary software development paradigm. Moreover, there will be a tool for translating executable HASCASL specifications to Haskell, which can also be used in order to animate specifications.

Acknowledgements

This work forms part of the DFG-funded project HasCASL (KR 1191/7-1). Partial support by the CoFI Working Group (ESPRIT WG 29432) is gratefully acknowledged, as well as the work of the CoFI Language and Semantics Task Groups. We wish to thank Bernd Krieg-Brückner for his salomonic solution of the non-strict function syntax problem, and Christoph Lüth for useful comments.

References

1. J. Adámek, H. Herrlich, and G. E. Strecker, *Abstract and concrete categories*, Wiley Interscience, 1990. 100
2. D. Aspinall, *Type systems for modular programming and specification*, Ph.D. thesis, Edinburgh, 1997. 112
3. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki, *CASL: the Common Algebraic Specification Language*, Theoret. Comput. Sci. (2003), to appear. 99, 100
4. E. Astesiano and M. Cerioli, *Free objects and equational deduction for partial conditional specifications*, Theoret. Comput. Sci. **152** (1995), 91–138. 102

5. S. Autexier and T. Mossakowski, *Integrating HOL-CASL into the development graph manager MAYA*, Frontiers of Combining Systems, LNCS, vol. 2309, Springer, 2002, pp. 2–17. [114](#)
6. G. Berry and P.-L. Curien, *Sequential algorithms on concrete data structures*, Theoret. Comput. Sci. **20** (1982), 265–321. [102](#)
7. V. Breazu-Tannen and A. R. Meyer, *Lambda calculus with constrained types*, Logic of Programs, LNCS, vol. 193, Springer, 1985, pp. 23–40. [104](#)
8. CoFI, *The Common Framework Initiative for algebraic specification and development, electronic archives*, <http://www.brics.dk/Projects/CoFI>. [99](#), [100](#), [115](#)
9. CoFI Language Design Task Group, *CASL – The CoFI Algebraic Specification Language – Summary, version 1.0*, Document/CASL/Summary, in [8], July 1999. [100](#), [107](#)
10. CoFI Semantics Task Group, *CASL – The CoFI Algebraic Specification Language – Semantics*, Note S-9 (version 0.96), in [8], July 1999. [101](#)
11. T. Coquand, *An analysis of Girard’s paradox*, Logic in Computer Science, IEEE, 1986, pp. 227–236. [105](#), [106](#), [114](#)
12. J. Goguen, *A categorical manifesto*, Math. Struct. Comput. Sci. **1** (1991), 49–67. [102](#)
13. R. Grosu and F. Regensburger, *The semantics of spectrum*, LNCS **816** (1994), 124–145. [112](#)
14. J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, *Larch: Languages and tools for formal specification*, Springer, 1993. [99](#)
15. L. Henkin, *The completeness of the first-order functional calculus*, J. Symbolic Logic **14** (1949), 159–166. [102](#)
16. J. M. E. Hyland, *First steps in synthetic domain theory*, Category Theory, LNM, vol. 1144, Springer, 1992, pp. 131–156. [112](#)
17. S. Kahrs, D. Sannella, and A. Tarlecki, *The definition of extended ML: A gentle introduction*, Theoret. Comput. Sci. **173** (1997), 445–484. [99](#)
18. J. Lambek and P. J. Scott, *Introduction to higher-order categorical logic*, Cambridge, 1986. [107](#)
19. S. Mac Lane, *Categories for the working mathematician*, Springer, 1997. [100](#)
20. J. C. Mitchell and P. J. Scott, *Typed lambda models and cartesian closed categories*, Categories in Computer Science and Logic, Contemp. Math., vol. 92, Amer. Math. Soc., 1989, pp. 301–316. [102](#), [107](#)
21. E. Moggi, *Categories of partial morphisms and the λ_p -calculus*, Category Theory and Computer Programming, LNCS, vol. 240, Springer, 1986, pp. 242–251. [99](#), [101](#), [102](#), [103](#), [104](#), [107](#)
22. E. Moggi, *The partial lambda calculus*, Ph.D. thesis, University of Edinburgh, 1988. [99](#)
23. T. Mossakowski, A. Haxthausen, and B. Krieg-Brückner, *Subsorted partial higher-order logic as an extension of CASL*, Workshop on Abstract Datatypes, LNCS, vol. 1827, Springer, 2000, pp. 126–145. [99](#)
24. Till Mossakowski, *CASL: From semantics to tools*, Tools and Algorithms for Construction and Analysis of Systems, LNCS, vol. 1785, Springer, 2000, pp. 93–108. [114](#)
25. G. Plotkin, *Domains (the ‘Pisa Notes’)*, <http://www.dcs.ed.ac.uk/home/gdp/> [100](#), [109](#)
26. A. Poigné, *On specifications, theories, and models with higher types*, Inform. and Control **68** (1986), no. 1–3, 1–46. [102](#)
27. F. Regensburger, *HOLCF: Higher order logic of computable functions*, Theorem Proving in Higher Order Logics, LNCS, vol. 971, 1995, pp. 293–307. [100](#), [109](#)

28. L. Schröder, *Classifying categories for partial equational logic*, Category Theory and Computer Science, ENTCS, 2002, to appear. 103, 104
29. L. Schröder and T. Mossakowski, *The definition of HASCASL*, in preparation. 100, 101, 103
30. D. S. Scott, *Relating theories of the λ -calculus*, To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalisms, Academic Press, 1980, pp. 403–450. 109
31. M. Wenzel, *Type classes and overloading in higher-order logic*, Theorem Proving in Higher Order Logics, LNCS, vol. 1275, Springer, 1997, pp. 307–322. 99, 104
32. Glynn Winskel, *The formal semantics of programming languages*, MIT, 1993. 100

Removing Redundant Arguments of Functions^{*}

María Alpuente, Santiago Escobar, and Salvador Lucas

DSIC, UPV
Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,sescobar,slucas}@dsic.upv.es

Abstract. The application of automatic transformation processes during the formal development and optimization of programs can introduce encumbrances in the generated code that programmers usually (or presumably) do not write. An example is the introduction of redundant arguments in the functions defined in the program. Redundancy of a parameter means that replacing it by any expression does not change the result. In this work, we provide a method for the analysis and elimination of redundant arguments in term rewriting systems as a model for the programs that can be written in more sophisticated languages. On the basis of the uselessness of redundant arguments, we also propose an erasure procedure which may avoid wasteful computations while still preserving the semantics (under ascertained conditions). A prototype implementation of these methods has been undertaken, which demonstrates the practicality of our approach.

1 Introduction

A number of researchers have noticed that certain processes of optimization, transformation, specialization and reuse of code often introduce anomalies in the generated code that programmers usually (or ideally) do not write [6,16,25,26]. Examples are redundant arguments in the functions defined by the program, as well as useless program rules.

Example 1. Consider the following program, which calculates the last element of a list and the concatenation of two lists of natural numbers, respectively:

```
data Nat = 0 | S Nat
append :: [Nat] -> [Nat] -> [Nat]      last :: [Nat] -> Nat
append nil y = y                       last (x:nil) = x
append (x:xs) y = x:(append xs y)      last (x:y:ys) = last (y:ys)
```

Assume that we specialize this program for the call (`applast ys z`) \equiv (`last (append ys (z:nil))`), which appends an element `z` at the end of a given list `ys` and then returns the last element, `z`, of the resulting list (the example is borrowed from DPPD library of benchmarks [24] and was also considered in [23,33] for logic program specialization). Commonly, the optimized program

^{*} Work partially supported by CICYT TIC2001-2705-C03-01, Acciones Integradas HI2000-0161, HA2001-0059, HU2001-0019, and Generalitat Valenciana GV01-424.

which can be obtained by using an automatic specializer of functional programs such as [3,4,5] is:

```

applast::[Nat] -> Nat -> Nat      lastnew::Nat -> [Nat] -> Nat -> Nat
applast nil    z = z              lastnew x nil    z = z
applast (x:xs) z = lastnew x xs z lastnew x (y:ys) z = lastnew y ys z

```

This program is too far from $\{\mathbf{applast}'\ ys\ z = \mathbf{lastnew}'\ z, \mathbf{lastnew}'\ z = z\}$, a more feasible program with the same evaluation semantics, or even the “optimal” program –without redundant parameters– $\{\mathbf{applast}''\ z = z\}$ which one would ideally expect (here the rule for the “local” function $\mathbf{lastnew}'$ is disregarded, since it is not useful when the definition of $\mathbf{applast}'$ is optimized). Indeed, note that the first argument of the function $\mathbf{applast}$ is redundant (as well as the first and second parameters of the auxiliary function $\mathbf{lastnew}$) and would not typically be written by a programmer who writes this program by hand. Also note that standard (post-specialization) renaming/compression procedures cannot perform this optimization as they only improve programs where program calls contain dead functors or multiple occurrences of the same variable, or the functions are defined by rules whose rhs’s are normalizable [3,11,12]. Known procedures for removing dead code such as [7,19,26] do not apply to this example either.

It seems interesting to formalize program analysis techniques for detecting these kinds of redundancies as well as to formalize transformations for eliminating dead code which appears in the form of redundant function arguments or useless rules and which, in some cases, can be safely erased without jeopardizing correctness. In this work, we investigate the problem of redundant arguments in term rewriting systems (TRSs), as a model for the programs that can be written in more sophisticated languages.

At first sight, one could naïvely think that redundant arguments are a straight counterpart of “needed redex” positions, which are well studied in [15]. Unfortunately, this is not true as illustrated by the following example.

Example 2. Consider the optimized program of Example 1 extended with:

```

take:: Nat -> [Nat] -> [Nat]
take 0    xs    = []
take (S n) (x:xs) = x:take n xs

```

The contraction of redex $(\mathbf{take}\ 1\ (1:2:[]))$ at position 1 in the term $t = \mathbf{applast}\ (\mathbf{take}\ 1\ (1:2:[]))\ 0$ (we use 1, 2 instead of $\mathbf{S}\ 0$, $\mathbf{S}\ (\mathbf{S}\ 0)$) is *needed* to normalize t (in the sense of [15]). However, the first argument of $\mathbf{applast}$ is redundant for normalization, as we showed in Example 1, and the program could be improved by dropping this useless parameter.

In this paper, we provide a semantic characterization of redundancy which is parametric w.r.t. the observed semantics \mathbf{S} . After some preliminaries in Section 2, in Section 3 we consider different (reduction) semantics, including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). In Section 4 we introduce the notion of redundancy of an argument w.r.t. a semantics \mathbf{S} , and derive a decidability result

for the redundancy problem w.r.t. \mathcal{S} . Inefficiencies caused by the redundancy of arguments cannot be avoided by using standard rewriting strategies. Therefore, in Section 5 we formalize an elimination procedure which gets rid of the useless arguments and provide sufficient conditions for the preservation of the semantics. Preliminary experiments indicate that our approach is both practical and useful. We conclude with some comparisons with the related literature and future work. Proofs of all technical results are given in [2].

2 Preliminaries

Term rewriting systems provide an adequate computational model for functional languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda) [8,17,34]. In the remainder of the paper we follow the standard framework of term rewriting for developing our results (see [8] for missing definitions).

Definitions are given in the one-sorted case. The extension to many-sorted signatures is not difficult [30]. Throughout the paper, \mathcal{X} denotes a countable set of variables and Σ denotes a set of function symbols $\{f, g, \dots\}$, each one having a fixed arity given by a function $ar : \Sigma \rightarrow \mathbb{N}$. If $ar(f) = 0$, we say that f is a constant symbol. By $\mathcal{T}(\Sigma, \mathcal{X})$ we denote the set of terms; $\mathcal{T}(\Sigma)$ is the set of ground terms, i.e., terms without variable occurrences. A term is said to be linear if it has no multiple occurrences of a single variable. A k -tuple t_1, \dots, t_k of terms is written \bar{t} . The number k of elements of the tuple \bar{t} will be clarified by the context. Let $Subst(\Sigma, \mathcal{X})$ denote the set of substitutions and $Subst(\Sigma)$ be the set of ground substitutions, i.e., substitutions on $\mathcal{T}(\Sigma)$. We denote by id the “identity” substitution: $id(x) = x$ for all $x \in \mathcal{X}$. By $Pos(t)$ we denote the set of positions of a term t . A rewrite rule is an ordered pair (l, r) , written $l \rightarrow r$, with $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$, $l \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. The left-hand side (*lhs*) of the rule is l and r is the right-hand side (*rhs*). A TRS is a pair $\mathcal{R} = (\Sigma, R)$ where R is a set of rewrite rules. $L(\mathcal{R})$ denotes the set of *lhs*’s of \mathcal{R} . By $NF_{\mathcal{R}}$ we denote the set of finite normal forms of \mathcal{R} . Given $\mathcal{R} = (\Sigma, R)$, we consider Σ as the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{F}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{F}$, called *defined functions*, where $\mathcal{F} = \{f \mid f(\bar{l}) \rightarrow r \in R\}$ and $\mathcal{C} = \Sigma - \mathcal{F}$. Then, $\mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of constructor terms. A pattern is a term $f(l_1, \dots, l_n)$ such that $f \in \mathcal{F}$ and $l_1, \dots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. The set of patterns is $\mathcal{Patt}(\Sigma, \mathcal{X})$. A constructor system (*CS*) is a TRS whose *lhs*’s are patterns. A term t is a head-normal form (or root-stable) if it cannot be rewritten to a redex. The set of head-normal forms of \mathcal{R} is denoted by $HNF_{\mathcal{R}}$.

3 Semantics

The redundancy of an argument of a function f in a TRS \mathcal{R} depends on the semantic properties of \mathcal{R} that we are interested in observing. Our notion of semantics is aimed to couch operational as well as denotational aspects.

A *term semantics* for a signature Σ is a mapping $S : \mathcal{T}(\Sigma) \rightarrow \mathcal{P}(\mathcal{T}(\Sigma))$ [28]. A *rewriting semantics* for a TRS $\mathcal{R} = (\Sigma, R)$ is a term semantics S for Σ such that, for all $t \in \mathcal{T}(\Sigma)$ and $s \in S(t)$, $t \rightarrow_{\mathcal{R}}^* s$.

The semantics which is most commonly considered in functional programming is the set of values (ground constructor terms) that \mathcal{R} is able to produce in a finite number of rewriting steps ($\text{eval}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\mathcal{C}) \mid t \rightarrow_{\mathcal{R}}^* s\}$). Other kinds of semantics often considered for \mathcal{R} are, e.g., the set of all possible reducts of a term which are reached in a finite number of steps ($\text{red}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\Sigma) \mid t \rightarrow_{\mathcal{R}}^* s\}$), the set of such reducts that are ground head-normal forms ($\text{hnf}_{\mathcal{R}}(t) = \text{red}_{\mathcal{R}}(t) \cap \text{HNF}_{\mathcal{R}}$), or ground normal forms ($\text{nf}_{\mathcal{R}}(t) = \text{hnf}_{\mathcal{R}}(t) \cap \text{NF}_{\mathcal{R}}$). We also consider the (trivial) semantics **empty** which assigns an empty set to every term. We often omit \mathcal{R} in the notations for rewriting semantics when it is clear from the context.

The ordering \preceq between semantics [28] provides some interesting properties regarding the redundancy of arguments. Given term semantics S, S' for a signature Σ , we write $S \preceq S'$ if there exists $T \subseteq \mathcal{T}(\Sigma)$ such that, for all $t \in \mathcal{T}(\Sigma)$, $S(t) = S'(t) \cap T$. We have $\text{empty} \preceq \text{eval}_{\mathcal{R}} \preceq \text{nf}_{\mathcal{R}} \preceq \text{hnf}_{\mathcal{R}} \preceq \text{red}_{\mathcal{R}}$.

Given a rewriting semantics S , it is interesting to determine whether S provides non-trivial information for every input expression. Let \mathcal{R} be a TRS and S be a rewriting semantics for \mathcal{R} , we say that \mathcal{R} is *S-defined* if for all $t \in \mathcal{T}(\Sigma)$, $S(t) \neq \emptyset$ [28]. S-definedness is monotone w.r.t. \preceq : if $S \preceq S'$ and \mathcal{R} is S-defined, \mathcal{R} is also S' -defined.

S-definedness has already been studied in the literature for different semantics [28]. In concrete, the eval-definedness is related to the standard notion of *completely defined* (CD) TRSs (see [18,20]). A defined function symbol is completely defined if it does not occur in any ground term in normal form, that is to say functions are reducible on all ground terms (of appropriate sort). A TRS \mathcal{R} is completely defined if each defined symbol of the signature is completely defined. In one-sorted theories, completely defined programs occur only rarely. However, they are common when using types, and each function is defined for all constructors of its argument types.

Let \mathcal{R} be a normalizing (i.e., every term has a normal form) and completely defined TRS; then, \mathcal{R} is $\text{eval}_{\mathcal{R}}$ -defined. Being completely defined is sensitive to extra constant symbols in the signature, and so is redundancy; we are not concerned with modularity in this paper.

4 Redundant Arguments

Roughly speaking, a redundant argument of a function f is an argument t_i which we do not need to consider in order to compute the semantics of any call containing a subterm $f(t_1, \dots, t_k)$.

Definition 1 (Redundancy of an argument). *Let S be a term semantics for a signature Σ , $f \in \Sigma$, and $i \in \{1, \dots, \text{ar}(f)\}$. The i -th argument of f is redundant w.r.t. S if, for all contexts $C[\]$ and for all $t, s \in \mathcal{T}(\Sigma)$ such that $\text{root}(t) = f$, $S(C[t]) = S(C[t[s]_i])$.*

We denote by $\text{rarg}_{\mathbb{S}}(f)$ the set of redundant arguments of a symbol $f \in \Sigma$ w.r.t. a semantics \mathbb{S} for Σ . Note that every argument of every symbol is redundant w.r.t. empty. Redundancy is antimonotone with regard to the ordering \preceq on semantics.

Theorem 1. *Let \mathbb{S}, \mathbb{S}' be term semantics for a signature Σ . If $\mathbb{S} \preceq \mathbb{S}'$, then, for all $f \in \Sigma$, $\text{rarg}_{\mathbb{S}'}(f) \subseteq \text{rarg}_{\mathbb{S}}(f)$.*

The following result guarantees that constructor symbols have no redundant arguments, which agrees with the common understanding of constructor terms as completely meaningful pieces of information.

Proposition 1. *Let \mathcal{R} be a TRS such that $\mathcal{T}(\mathcal{C}) \neq \emptyset$, and consider \mathbb{S} such that $\text{eval}_{\mathcal{R}} \preceq \mathbb{S}$. Then, for all $c \in \mathcal{C}$, $\text{rarg}_{\mathbb{S}}(c) = \emptyset$.*

In general, the redundancy of an argument is undecidable. In the following, for a signature Σ , term semantics \mathbb{S} for Σ , $f \in \Sigma$, and $i \in \{1, \dots, \text{ar}(f)\}$, by “the redundancy problem w.r.t. \mathbb{S} ”, we mean the redundancy of the i -th argument of f w.r.t. \mathbb{S} . The following theorem provides a decidability result w.r.t. all the semantics considered in this paper. This result recalls the decidability of other related properties of TRSs, such as the confluence and joinability, and reachability problems (for left-linear, right-ground TRSs) [10,29].

Theorem 2. *For a left-linear, right-ground TRS $\mathcal{R} = (\Sigma, R)$ over a finite signature Σ , the redundancy w.r.t. semantics $\text{red}_{\mathcal{R}}$, $\text{hnf}_{\mathcal{R}}$, $\text{nf}_{\mathcal{R}}$, and $\text{eval}_{\mathcal{R}}$ is decidable.*

In the following sections, we address the redundancy analysis from a complementary perspective. Rather than going more deeply in the decidability issues, we are interested in ascertaining conditions which (sufficiently) ensure that an argument is redundant in a given TRS. In order to address this problem, we investigate redundancy of positions.

4.1 Redundancy of Positions

When considering a particular (possibly non-ground) function call, we can observe a more general notion of redundancy which allows us to consider arbitrary (deeper) positions within the call. We say that two terms $t, s \in \mathcal{T}(\Sigma, \mathcal{X})$ are p -equal, with $p \in \text{Pos}(t) \cap \text{Pos}(s)$ if, for all occurrences w with $w < p$, $t|_w$ and $s|_w$ have the same root.

Definition 2 (Redundant position). *Let \mathbb{S} be a term semantics for a signature Σ and $t \in \mathcal{T}(\Sigma, \mathcal{X})$. The position $p \in \text{Pos}(t)$ is redundant in t w.r.t. \mathbb{S} if, for all $t', s \in \mathcal{T}(\Sigma)$ such that t and t' are p -equal, $\mathbb{S}(t') = \mathbb{S}(t'[s]_p)$.*

We denote by $\text{rpos}_{\mathbb{S}}(t)$ the set of redundant positions of a term t w.r.t. a semantics \mathbb{S} . Note that the previous definition cannot be simplified by getting rid of t' and simply requiring that for all $s \in \mathcal{T}(\Sigma)$, $\mathbb{S}(t) = \mathbb{S}(t[s]_p)$ since redundant positions cannot be analyzed independently if the final goal is to remove the useless arguments, i.e. our notion of redundancy becomes not compositional.

Example 3. Let us consider the TRS \mathcal{R} :

$$f(a,a) \rightarrow a \quad f(a,b) \rightarrow a \quad f(b,a) \rightarrow a \quad f(b,b) \rightarrow b$$

Given the term $f(a,a)$, for all term $s \in \mathcal{T}(\Sigma)$, $\text{eval}_{\mathcal{R}}(t[s]_1) = \text{eval}_{\mathcal{R}}(t)$ and $\text{eval}_{\mathcal{R}}(t[s]_2) = \text{eval}_{\mathcal{R}}(t)$. However, $\text{eval}_{\mathcal{R}}(t[b]_1[b]_2) \neq \text{eval}_{\mathcal{R}}(t)$.

The following result states that the positions of a term which are below the indices addressing the redundant arguments of any function symbol occurring in t are redundant.

Proposition 2. *Let S be a term semantics for a signature $\Sigma = \mathcal{F} \uplus \mathcal{C}$, $t \in \mathcal{T}(\Sigma, \mathcal{X})$, $p \in \text{Pos}(t)$, $f \in \mathcal{F}$. For all positions q, p' and $i \in \text{rarg}_{\Sigma}(f)$ such that $p = q.i.p'$ and $\text{root}(t|_q) = f$, $p \in \text{rpos}_{\Sigma}(t)$ holds.*

In the following section, we provide some general criteria for ensuring redundancy of arguments on the basis of the (redundancy of some) positions in the rhs's of program rules, specifically the positions of the rhs's where the arguments of the functions defined in the lhs's 'propagate' to.

4.2 Using Redundant Positions for Characterizing Redundancy

Theorem 1 says that the more restrictive a semantics is, the more redundancies there are for the arguments of function symbols. According to our hierarchy of semantics (by \preceq), eval seems to be the most fruitful semantics for analyzing redundant arguments. In the following, we focus on the problem of characterizing the redundant arguments w.r.t. eval by studying the redundancy w.r.t. eval of some positions in the rhs's of program rules.

We say that $p \in \text{Pos}(t)$ is a sub-constructor position of t if for all $q < p$, $\text{root}(t|_q) \in \mathcal{C}$. In particular, Λ is a sub-constructor position.

Definition 3 ((f, i)-redundant variable). *Let S be a term semantics for a signature Σ , $f \in \mathcal{F}$, $i \in \{1, \dots, \text{ar}(f)\}$, and $t \in \mathcal{T}(\Sigma, \mathcal{X})$. The variable $x \in \mathcal{X}$ is (f, i)-redundant in t if it occurs only at positions $p \in \text{Pos}_x(t)$ which are redundant in t , in symbols $p \in \text{rpos}_{\Sigma}(t)$, or it appears in sub-constructor positions of the i -th parameter of f -rooted subterms of t , in symbols $\exists q, q'$ such that $p = q.i.q'$, $\text{root}(t|_q) = f$ and q' is a sub-constructor position of subterm $t|_{q.i}$.*

Note that variables which do not occur in a term t are trivially (f, i)-redundant in t for any $f \in \Sigma$ and $i \in \{1, \dots, \text{ar}(f)\}$. Given a TRS $\mathcal{R} = (\Sigma, R)$, we write \mathcal{R}_f to denote the TRS $\mathcal{R}_f = (\Sigma, \{l \rightarrow r \in R \mid \text{root}(l) = f\})$ which contains the set of rules defining $f \in \mathcal{F}$.

Theorem 3. *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be a left-linear CS. Let $f \in \mathcal{F}$ and $i \in \{1, \dots, \text{ar}(f)\}$. If, for all $l \rightarrow r \in \mathcal{R}_f$, $l|_i$ is a variable which is (f, i)-redundant in r , then $i \in \text{rarg}_{\text{eval}_{\mathcal{R}}}(f)$.*

Example 4. A standard example in the literature on *useless variable elimination* (UVE) -a popular technique for removing dead variables, see [36,19]- is the following program¹:

```
loop(a,bogus,0) → loop(f(a,0),s(bogus),s(0))
loop(a,bogus,s(j)) → a
```

Here it is clear that the second argument does not contribute to the value of the computation. By Theorem 3, the second argument of `loop` is redundant w.r.t. $\text{eval}_{\mathcal{R}}$.

The following example demonstrates that the restriction to constructor systems in Theorem 3 above is necessary.

Example 5. Consider the following TRS $\mathcal{R} \{f(x) \rightarrow g(f(x)), g(f(x)) \rightarrow x\}$. Then, the argument 1 of $f(x)$ in the lhs of the first rule is a variable which, in the corresponding rhs of the rule, occurs within the argument 1 of a subterm rooted by f , namely $f(x)$. Hence, by Theorem 3 we would have that $1 \in \text{rarg}_{\text{eval}_{\mathcal{R}}}(f)$. However, $\text{eval}_{\mathcal{R}}(f(a)) = \{a\} \neq \{b\} = \text{eval}_{\mathcal{R}}(f(b))$ (considering $a, b \in \mathcal{C}$), which contradicts $1 \in \text{rarg}_{\text{eval}_{\mathcal{R}}}(f)$.

Using Theorem 3, we are also able to conclude that the first argument of function `lastnew` in Example 1 is redundant w.r.t. $\text{eval}_{\mathcal{R}}$. Unfortunately, Theorem 3 does not suffice to prove that the *second* argument of `lastnew` is redundant w.r.t. $\text{eval}_{\mathcal{R}}$.

In the following, we provide a different sufficient criterion for redundancy which is less demanding regarding the shape of the left hand sides, although it requires orthogonality and eval -definedness, in return. The following definitions are auxiliary.

Definition 4. Let Σ be a signature, $t = f(t_1, \dots, t_k)$, $s = f(s_1, \dots, s_k)$ be terms and $i \in \{1, \dots, k\}$. We say that t and s unify up to i -th argument with mgu σ if $\langle t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_k \rangle$ and $\langle s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_k \rangle$ unify with mgu σ .

Definition 5 ((f, i)-triple). Let $\mathcal{R} = (\Sigma, R)$ be a CS, $f \in \Sigma$, and $i \in \{1, \dots, \text{ar}(f)\}$. Given two different (possibly renamed) rules $l \rightarrow r$, $l' \rightarrow r'$ in \mathcal{R}_f such that $\text{Var}(l) \cap \text{Var}(l') = \emptyset$, we say that $\langle l \rightarrow r, l' \rightarrow r', \sigma \rangle$ is an (f, i)-triple of \mathcal{R} if l and l' unify up to i -th argument with mgu σ .

Example 6. Consider the following CS \mathcal{R} from Example 1:

```
applast(nil,z) → z                lastnew(x,nil,z) → z
applast(x:xs,z) → lastnew(x,xs,z) lastnew(x,y:ys,z) → lastnew(y,ys,z)
```

This program has a single (`lastnew`, 2)-triple:

```
\langle lastnew(x,nil,z) → z, lastnew(x,y:ys,z) → lastnew(y,ys,z), id \rangle
```

Definition 6 (Joinable (f, i)-triple). Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be a CS, $f \in \mathcal{F}$, and $i \in \{1, \dots, \text{ar}(f)\}$. An (f, i)-triple $\langle l \rightarrow r, l' \rightarrow r', \sigma \rangle$ of \mathcal{R} is joinable if $\sigma_{\mathcal{C}}(r)$

¹ The original example uses natural 100 as stopping criteria for the third argument, while we simplify here to natural 1 in order to code it as 0/s terms.

and $\sigma_{\mathcal{C}}(r')$ are joinable (i.e., they have a common reduct). Here, substitution $\sigma_{\mathcal{C}}$ is given by:

$$\sigma_{\mathcal{C}}(x) = \begin{cases} \sigma(x) & \text{if } x \notin \text{Var}(l|_i) \cup \text{Var}(l'|_i) \\ a & \text{otherwise, where } a \in \mathcal{C} \text{ is an arbitrary constant of appropriate sort} \end{cases}$$

Example 7. Consider again the CS \mathcal{R} and the single $(\text{lastnew}, 2)$ -triple given in Example 6. With ϑ given by $\vartheta = \{\mathbf{y} \mapsto 0, \mathbf{ys} \mapsto \text{nil}\}$, the corresponding rhs's instantiated by ϑ , namely \mathbf{z} and $\text{lastnew}(0, \text{nil}, \mathbf{z})$, are joinable (\mathbf{z} is the common reduct). Hence, the considered $(\text{lastnew}, 2)$ -triple is joinable.

Roughly speaking, the result below formalizes a method to determine redundancy w.r.t. eval which is based on finding a common reduct of (some particular instances of) the right-hand sides of rules.

Definition 7 ((f, i)-joinable TRS). Let $\mathcal{R} = (\Sigma, R)$ be a TRS, S be a rewriting semantics for \mathcal{R} , $f \in \Sigma$, and $i \in \{1, \dots, \text{ar}(f)\}$. \mathcal{R} is (f, i) -joinable if, for all $l \rightarrow r \in \mathcal{R}_f$ and $x \in \text{Var}(l|_i)$, x is (f, i) -redundant in r and all (f, i) -triples of \mathcal{R} are joinable.

Theorem 4. Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be an orthogonal and $\text{eval}_{\mathcal{R}}$ -defined CS. Let $f \in \mathcal{F}$ and $i \in \{1, \dots, \text{ar}(f)\}$. If \mathcal{R} is (f, i) -joinable then $i \in \text{rarg}_{\text{eval}_{\mathcal{R}}}(f)$.

Joinability is decidable for terminating, confluent TRSs as well as for other classes of TRSs such as right-ground TRSs (see e.g., [29]). Hence, Theorem 4 gives us an effective method to recognize redundancy in CD, left-linear, and (semi-)complete TRSs, as illustrated in the following.

Example 8. Consider again the CS \mathcal{R} of Example 6. This program is orthogonal, terminating and CD (considering sorts), hence is eval -defined. Now, we have the following. The first argument of lastnew is redundant w.r.t. $\text{eval}_{\mathcal{R}}$ (Theorem 3). The second argument of lastnew is redundant w.r.t. $\text{eval}_{\mathcal{R}}$ (Theorem 4). As a consequence, the positions of variables \mathbf{x} and \mathbf{xs} in the rhs of the first rule of applast have been proven redundant. Then, since both $\text{lastnew}(0, \text{nil}, \mathbf{z})$ and \mathbf{z} rewrite to \mathbf{z} , $\mathcal{R}_{\text{applast}}$ is $(\text{applast}, 1)$ -joinable. By Theorem 4, we conclude that the first argument of applast is redundant.

5 Erasing Redundant Arguments

The presence of redundant arguments within input expressions wastes memory space and can lead to time consuming explorations and transformations (by replacement) of their structure. Redundant arguments are not necessary to determine the result of a function call. At first sight, one could expect that a suitable rewriting strategy which is able to avoid the exploration of redundant arguments of symbols could be defined. In Example 2, we showed that needed reduction is not able to avoid redundant arguments. Context-sensitive rewriting (*csr*) [27], which can be used to forbid reductions on selected arguments of symbols, could also seem adequate for avoiding fruitless reductions at redundant arguments. In

csr, a replacement map μ indicates the arguments $\mu(f) \subseteq \{1, \dots, ar(f)\}$ of function symbol f on which reductions are allowed. Let \mathcal{R} be the program `applast` of Example 1 extended with the rules for function `take` of Example 2. If we fix $\mu(\text{applast}) = \{2\}$ to (try to) avoid wasteful computations on the first argument of `applast`, using *csr* we are not able to reduce `applast (take 1 (1:2: [])) 0` to 0.

In this section, we formalize a procedure for *removing* redundant arguments from a TRS. The basic idea is simple: if an argument of f is redundant, it does not contribute to obtaining the value of any call to f and can be dropped from program \mathcal{R} . Hence, we remove redundant formal parameters and corresponding actual parameters for each function symbol in \mathcal{R} . We begin with the notion of syntactic erasure which is intended to pick up redundant arguments of function symbols.

Definition 8 (Syntactic erasure). *A syntactic erasure is a mapping $\rho : \Sigma \rightarrow \mathcal{P}(\mathbb{N})$ such that for all $f \in \Sigma$, $\rho(f) \subseteq \{1, \dots, ar(f)\}$. We say that a syntactic erasure ρ is sound for a semantics S if, for all $f \in \Sigma$, $\rho(f) \subseteq \text{rarg}_S(f)$.*

Example 9. Given the signature $\Sigma = \{0, \text{nil}, \text{s}, :, \text{applast}, \text{lastnew}\}$ of the TRS \mathcal{R} in Example 6, with $ar(0) = ar(\text{nil}) = 0$, $ar(\text{s}) = 1$, $ar(:) = ar(\text{applast}) = 2$, and $ar(\text{lastnew}) = 3$, the following mapping ρ is a sound syntactic erasure for the semantics $\text{eval}_{\mathcal{R}}$: $\rho(0) = \rho(\text{nil}) = \rho(\text{s}) = \rho(:) = \emptyset$, $\rho(\text{applast}) = \{1\}$, and $\rho(\text{lastnew}) = \{1, 2\}$.

Since we are interested in *removing* redundant arguments from function symbols, we transform the functions by reducing their arity according to the information provided by the redundancy analysis, thus building a new, *erased* signature.

Definition 9 (Erasure of a signature). *Given a signature Σ and a syntactic erasure $\rho : \Sigma \rightarrow \mathcal{P}(\mathbb{N})$, the erasure of Σ is the signature Σ_ρ whose symbols $f_\rho \in \Sigma_\rho$ are one to one with symbols $f \in \Sigma$ and whose arities are related by $ar(f_\rho) = ar(f) - |\rho(f)|$.*

Example 10. The erasure of the signature in Example 9 is $\Sigma_\rho = \{0, \text{nil}, \text{s}, :, \text{applast}, \text{lastnew}\}$, with $ar(0) = ar(\text{nil}) = 0$, $ar(\text{s}) = ar(\text{applast}) = ar(\text{lastnew}) = 1$, and $ar(:) = 2$. Note that, by abuse, we use the same symbols for the functions of the erased signature.

Now we extend the procedure to terms in the obvious way.

Definition 10 (Erasure of a term). *Given a syntactic erasure $\rho : \Sigma \rightarrow \mathcal{P}(\mathbb{N})$, the function $\tau_\rho : \mathcal{T}(\Sigma, \mathcal{X}) \rightarrow \mathcal{T}(\Sigma_\rho, \mathcal{X})$ on terms is: $\tau_\rho(x) = x$ if $x \in \mathcal{X}$ and $\tau_\rho(f(t_1, \dots, t_n)) = f_\rho(\tau_\rho(t_{i_1}), \dots, \tau_\rho(t_{i_k}))$ where $\{1, \dots, n\} - \rho(f) = \{i_1, \dots, i_k\}$ and $i_m < i_{m+1}$ for $1 \leq m < k$.*

The erasure procedure is extended to TRSs: we erase the lhs's and rhs's of each rule according to τ_ρ . In order to avoid extra variables in rhs's of rules (that arise from the elimination of redundant arguments of symbols in the corresponding lhs), we replace them by an arbitrary constant of Σ (which automatically belongs to Σ_ρ).

Definition 11 (Erasure of a TRS). Let $\mathcal{R} = (\Sigma, R)$ be a TRS, such that Σ has a constant symbol a , and ρ be a syntactic erasure for Σ . The erasure \mathcal{R}_ρ of \mathcal{R} is $\mathcal{R}_\rho = (\Sigma_\rho, \{\tau_\rho(l) \rightarrow \sigma_l(\tau_\rho(r)) \mid l \rightarrow r \in R\})$ where the substitution σ_l for a lhs l is given by $\sigma_l(x) = a$ for all $x \in \text{Var}(l) - \text{Var}(\tau_\rho(l))$ and $\sigma_l(y) = y$ whenever $y \in \text{Var}(\tau_\rho(l))$.

Example 11. Let \mathcal{R} be the TRS of Example 6 and ρ be the sound syntactic erasure of Example 9. The erasure \mathcal{R}_ρ of \mathcal{R} consists of the erased signature of Example 10 together with the following rules:

$$\begin{array}{ll} \text{applast}(z) \rightarrow z & \text{lastnew}(z) \rightarrow z \\ \text{applast}(z) \rightarrow \text{lastnew}(z) & \text{lastnew}(z) \rightarrow \text{lastnew}(z) \end{array}$$

Below, we introduce a further improvement aimed to provide the final, “optimal” program.

The following theorem establishes the correctness and completeness of the erasure procedure for the semantics eval.

Theorem 5 (Correctness and completeness). Let $\mathcal{R} = (\Sigma, \mathcal{R})$ be a left-linear TRS, ρ be a sound syntactic erasure for $\text{eval}_{\mathcal{R}}$, $t \in \mathcal{T}(\Sigma)$, and $\delta \in \mathcal{T}(\mathcal{C})$. Then, $\tau_\rho(t) \rightarrow_{\mathcal{R}_\rho}^* \delta$ iff $\delta \in \text{eval}_{\mathcal{R}}(t)$.

In the following we ascertain the conditions for the preservation of some computational properties of TRSs under erasure.

Theorem 6. Let \mathcal{R} be a left-linear TRS. Let ρ be a sound syntactic erasure for $\text{eval}_{\mathcal{R}}$. If \mathcal{R} is $\text{eval}_{\mathcal{R}}$ -defined and confluent, then the erasure \mathcal{R}_ρ of \mathcal{R} is confluent.

Theorem 7. Let \mathcal{R} be a left-linear and CD TRS, and ρ be a sound syntactic erasure for $\text{eval}_{\mathcal{R}}$. If \mathcal{R} is normalizing, then the erasure \mathcal{R}_ρ of \mathcal{R} is normalizing.

In the theorem above, we cannot strengthen normalization to termination. A simple counterexample showing that termination may get lost is the following.

Example 12. Consider the left-linear, (confluent, CD, and) terminating TRS $\mathcal{R} \{\mathbf{f}(\mathbf{a}, \mathbf{y}) \rightarrow \mathbf{a}, \mathbf{f}(\mathbf{c}(\mathbf{x}), \mathbf{y}) \rightarrow \mathbf{f}(\mathbf{x}, \mathbf{c}(\mathbf{y}))\}$. The first argument of \mathbf{f} is redundant w.r.t. $\text{eval}_{\mathcal{R}}$. However, after erasing the argument, we get the TRS $\{\mathbf{f}(\mathbf{y}) \rightarrow \mathbf{a}, \mathbf{f}(\mathbf{y}) \rightarrow \mathbf{f}(\mathbf{c}(\mathbf{y}))\}$, which is not terminating.

In the example above, note that the resulting TRS is not orthogonal, whereas the original program is. Hence, this example also shows that orthogonality is not preserved under erasure.

The following post-processing transformation can improve the optimization achieved.

Definition 12 (Reduced erasure of a TRS). Let $\mathcal{R} = (\Sigma, R)$ be a TRS and ρ be a syntactic erasure for Σ . The reduced erasure \mathcal{R}'_ρ of \mathcal{R} is obtained from the erasure \mathcal{R}_ρ of \mathcal{R} by a compression transformation defined as removing any trivial rule $t \rightarrow t$ of \mathcal{R}_ρ and then normalizing the rhs’s of the rules w.r.t. the non-trivial rules of \mathcal{R}_ρ .

Reduced erasures are well-defined whenever \mathcal{R}_ρ is confluent and normalizing since, for such systems, every term has a unique normal form.

Example 13. Let \mathcal{R}_ρ be the erasure of Example 13. The reduced erasure consists of the rules $\{\text{applast}(\mathbf{z}) \rightarrow \mathbf{z}, \text{lastnew}(\mathbf{z}) \rightarrow \mathbf{z}\}$.

Since right-normalization preserves confluence, termination and the equational theory (as well as confluence, normalization and the equational theory, in almost orthogonal and normalizing TRSs) [13], and the removal of trivial rules does not change the evaluation semantics of the TRS \mathcal{R} either, we have the following.

Corollary 1. *Let \mathcal{R} be a left-linear TRS, ρ be a sound syntactic erasure for $\text{eval}_{\mathcal{R}}$, $t \in \mathcal{T}(\Sigma)$, and $\delta \in \mathcal{T}(\mathcal{C})$. If (the TRS which results from removing trivial rules from) \mathcal{R}_ρ is confluent and terminating (alternatively, if it is almost orthogonal and normalizing), then, $\tau_\rho(t) \rightarrow_{\mathcal{R}'_\rho}^* \delta$ if and only if $\delta \in \text{eval}_{\mathcal{R}}(t)$, where \mathcal{R}'_ρ is the reduced erasure of \mathcal{R} .*

Erasures and reduced erasures of a TRS preserve left-linearity. For a TRS \mathcal{R} satisfying the conditions in Corollary 1, by using [13], it is immediate that the reduced erasure \mathcal{R}'_ρ is confluent and normalizing. Also, \mathcal{R}'_ρ is CD if \mathcal{R} is.

Hence, let us note that these results allow us to perform the optimization of program `applast` while guaranteeing that the intended semantics is preserved.

6 Conclusion

This paper provides the first results concerning the detection and removal of useless arguments in program functions. We have given a semantic definition of redundancy which takes the semantics \mathcal{S} as a parameter, and then we have considered the evaluation semantics (closer to functional programming).

In order to provide practical methods to recognize redundancy, we have ascertained suitable conditions which allow us to simplify the general redundancy problem to the analysis of redundant positions within rhs's of the program rules. These conditions are quite demanding (requiring \mathcal{R} to be orthogonal and $\text{eval}_{\mathcal{R}}$ -defined) but also the optimization which they enable is strong, and powerful. Actually, inefficiencies caused by the redundancy of arguments cannot be avoided by using standard reduction strategies. Therefore, we have developed a transformation for eliminating dead code which appears in the form of useless function calls and we have proven that the transformation preserves the semantics (and the operational properties) of the original program under ascertained conditions. The optimized program that we produce cannot be created as the result of applying standard transformations of functional programming to the original program (such as partial evaluation, supercompilation, and deforestation, see e.g. [32]). We believe that the semantic grounds for redundancy analyses and elimination laid in this paper may foster further insights and developments in the functional programming community and neighbouring fields.

The practicality of our ideas is witnessed by the implementation of a prototype system which delivers encouragingly good results for the techniques deployed in the paper (Sections 4.2 and 5). The prototype has been implemented in PAKCS, the current distribution of the multi-paradigm declarative language Curry [14], and is publicly available at

<http://www.dsic.upv.es/users/elp/redargs>.

We have used the prototype to perform some preliminary experiments (available at <http://www.dsic.upv.es/users/elp/redargs/experiments>) which show that our methodology does detect and remove redundant arguments of many common transformation benchmarks, such as `bogus`, `lastappend`, `allzeros`, `doubleflip`, etc (see [24] and references therein). See [2] for details.

6.1 Related Work

Some notions have appeared in the literature of what it means for a term in a TRS \mathcal{R} to be “computationally irrelevant”. Our analysis is different from all the related methods in many respects and, in general, incomparable to them.

Contrarily to our notion of redundancy, the meaninglessness of [22,21] is a property of the terms themselves (they may have meaning in \mathcal{R} or may not), whereas our notion refers to arguments (positions) of function symbols. In [22], Section 7.1, a term t is called *meaningless* if, for each context $C[\]$ s.t. $C[t]$ has a normal form, we have that $C[t']$ has the same normal form for all terms t' . This can be seen as a kind of superfluity (w.r.t. normal forms) of a fixed expression in any context, whereas our notion of redundancy refers to the possibility of getting rid of some arguments of a given function symbol with regard to some observed semantics. The meaninglessness of [22] is not helpful for the purposes of optimizing programs by removing useless arguments of function symbols which we pursue. On the other hand, terms with a normal form are proven meaningful (i.e., not meaningless) in [22,21], whereas we might have redundant arguments which are normal forms.

Among the vast literature on analysis (and removal) of unnecessary data structures, the analyses of *unneededness* (or *absence*) of functional programming [9,16], and the *filtering* of useless arguments and unnecessary variables of logic programming [25,31] are the closest to our work. In [16], a notion of *needed/unneeded* parameter for list-manipulation programs is introduced which is closely related to the redundancy of ours in that it is capable of identifying whether the value of a subexpression is ignored. The method is formulated in terms of a fixed, finite set of projection functions which introduces some limitations on the class of neededness patterns that can be identified. Since our method gives the information that a parameter is definitely not necessary, our redundancy notion implies Hughes’s unneededness, but not vice versa. For instance, constructor symbols cannot have redundant arguments in our framework (Proposition 1), whereas Hughes’ notion of unneededness can be applied to the elements of a list: Hughes’ analysis is able to determine that in the `length` function (defined as usual), the spine of the argument list is needed but the elements of the list are not needed; this is used to perform some optimizations for the compiler. However, this information cannot be used for the purposes of our work, that is, to remove these elements when the entire list cannot be eliminated.

On the other hand, Hughes’s notion of *neededness/unneededness* should not be confused with the standard notion of needed (positions of) redexes of [15]:

Example 2 shows that Huet and Levy’s neededness does not imply the non-redundancy of the corresponding argument or position (nor vice versa).

The notion of redundancy of an argument in a term rewriting system can be seen as a kind of *compartment property* as defined in [9]. Cousot’s compartment analysis generalizes not only the unneededness analyses but also strictness, termination and other standard analyses of functional programming. In [9], compartment is mainly investigated within a denotational framework, whereas our approximation is independent from the semantic formalism.

Proietti and Pettorossi’s *elimination procedure* for the removal of unnecessary variables is a powerful unfold/fold-based transformation procedure for logic programs; therefore, it does not compare directly with our method, which would be seen as a post-processing phase for program transformers optimization. Regarding the kind of *unnecessary variables* that the elimination procedure can remove, only variables that occur more than once in the body of the program rule and which do not occur in the head of the rule can be dropped. This is not to say that the transformation is powerless; on the contrary, the effect can be very striking as these kinds of variables often determine multiple traversals of intermediate data structures which are then removed from the program. Our procedure for removing redundant arguments is also related to the Leuschel and Sørensen RAF and FAR algorithms [25], which apply to removing unnecessary arguments in the context of (conjunctive) partial evaluation of logic programs. However, a comparison is not easy either as we have not yet considered the semantics of computed answers for our programs.

People in the functional programming community have also studied the problem of useless variable elimination (UVE). Apparently, they were unaware of the works of the logic programming community, and they started studying the topic from scratch, mainly following a flow-based approach [36] or a type-based approach [7,19] (see [7] for a discussion of this line of research). All these works address the problem of safe elimination of dead *variables* but heavily handle data structures. A notable exception is [26], where Liu and Stoller discuss how to safely eliminate dead code in the presence of recursive data structures by applying a methodology based on regular tree grammars. Unfortunately, the method in [26] does not apply to achieve the optimization pursued in our running example `applast`.

Obviously, there exist examples (inspired) in the previously discussed works which cannot be directly handled with our results, consider the following TRS:

$$\text{length}([]) \rightarrow 0 \quad \text{length}(x:xs) \rightarrow s(\text{length}(xs)) \quad f(x) \rightarrow \text{length}(x:[])$$

Our methods do not capture the redundancy of the argument of `f`. In [26] it is shown that, in order to evaluate `length(xs)`, we do not need to evaluate the elements of the argument list `xs`. In Liu et al.’s methodology, this means that we could replace the rule for `f` above by `f(·) → length(·:[])` where `·` is a new constant. However, as discussed in Section 5, this could still lead to wasteful computations if, e.g., an eager strategy is used for evaluating the expressions: in that case, a term `t` within a call `f(t)` would be wastefully reduced. Nevertheless, Theorem 3 can be used now with the new rule to recognize the

first argument of f as redundant. That is, we are allowed to use the following rule: $f \rightarrow \text{length}(_ : [])$ which completely avoids wasteful computations on redundant arguments. Hence, the different methods are complementary and an enhanced test might be developed by properly combining them.

Recently, the problem of identifying redundant arguments of function symbols has been reduced to proving the validity of a particular class of inductive theorems in the equational theory of confluent, sufficiently complete TRSs. We refer to [1] for details, where a comparison with approximation methods based on abstract interpretation can also be found.

References

1. M. Alpuente, R. Echahed, S. Escobar, S. Lucas. Redundancy of Arguments Reduced to Induction. In *Proc. of WFLP'02*, ENTCS, to appear, 2002. **130**
2. M. Alpuente, S. Escobar, S. Lucas. Removing Redundants Arguments of Functions. Technical report DSIC-II/8/02, UPV, 2002. **119, 128**
3. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of PEPM'97, ACM Sigplan Notices*, volume 32(12):151–162. ACM Press, New York, 1997. **118**
4. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998. **118**
5. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of ICFP'99, ACM Sigplan Notices*, 34(9):273–283, ACM Press, New York, 1999. **118**
6. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison-Wesley, 1986. **117**
7. S. Berardi, M. Coppo, F. Damiani and P. Giannini. Type-Based Useless-Code Elimination for Functional Programs. In Walid Taha, editor, *Proc. of SAIG 2000*, LNCS 1924:172–189, Springer-Verlang, 2000. **118, 129**
8. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. **119**
9. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to compartment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. of ICCL'94*, pages 95–112. IEEE Computer Society Press, Los Alamitos, California, 1994. **128, 129**
10. M. Dauchet, T. Heuillard, P. Lescanne, and S. Tison. Decidability of the Confluence of Finite Ground Term Rewrite Systems and of Other Related Term Rewriting Systems. *Information and Computation*, 88:187–201, 1990. **121**
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of PEPM'93*, pages 88–98. ACM, New York, 1993. **118**
12. R. Glück and M. Sørensen. Partial deduction and driving are equivalent. In *Proc. of PLILP'94*, LNCS 844:165–181. Springer-Verlag, Berlin, 1994. **118**
13. B. Gramlich. On Interreduction of Semi-Complete Term Rewriting Systems. *Theoretical Computer Science*, 258(1-2):435–451, 2001. **127**
14. M. Hanus. Curry: An Integrated Functional Logic Language. Available at <http://www.informatik.uni-kiel.de/~curry>, 2001. **127**
15. G. Huet and J. J. Lévy. Computations in orthogonal term rewriting systems. In J. L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of J.*

- Alan Robinson*, pages 395-414 and 415-443. The MIT Press, Cambridge, MA, 1991. [118](#), [128](#)
16. J. Hughes. Backwards Analysis of Functional Programs. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, pages 187–208, 1988. [117](#), [128](#)
 17. J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay and T. S. E. Maibaum. *Handbook of Logic in Computer Science*, volume 3, pages 1-116. Oxford University Press, 1992. [119](#)
 18. D. Kapur, P. Narendran, and Z. Zhang. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica* 24:395-416, 1987. [120](#)
 19. N. Kobayashi. Type-based useless variable elimination. In *roc. of PEPM-00*, pages 84-93, ACM Press, 2000. [118](#), [123](#), [129](#)
 20. E. Kounalis. Completeness in data type specifications. In B. F. Caviness, editor, *Proc. of EUROCAL'85*, LNCS 204:348-362. Springer-Verlag, Berlin, 1985. [120](#)
 21. R. Kennaway, V. van Oostrom, F. J. de Vries. Meaningless Terms in Rewriting. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Proc. of ALP'96*, LNCS 1139:254–268. Springer-Verlag, Berlin, 1996. [128](#)
 22. J. Kuper. Partiality in Logic and Computation. Aspects of Undefinedness. PhD Thesis, Universiteit Twente, February 1994. [128](#)
 23. M. Leuschel and B. Martens. Partial Deduction of the Ground Representation and Its Application to Integrity Checking. Tech. Rep. CW 210, K. U. Leuven, 1995. [117](#)
 24. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Tech. Rep., Accessible via <http://www.ecs.soton.ac.uk/~mal/>. [117](#), [128](#)
 25. M. Leuschel and M. H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc of LOPSTR'96*, LNCS 1207:83–103. Springer-Verlag, Berlin, 1996. [117](#), [128](#), [129](#)
 26. Y. A. Liu and S. D. Stoller. Eliminating dead code on recursive data. Science of Computer Programming, 2002. To appear. Preliminary version in *Proc. of SAS'99*, LNCS 1694:211–231. Springer-Verlag, Berlin, 1999. [117](#), [118](#), [129](#)
 27. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1-61, January 1998. [124](#)
 28. S. Lucas. Transfinite Rewriting Semantics for Term Rewriting Systems *Proc. of RTA'01*, LNCS 2051:216–230. Springer-Verlag, Berlin, 2001. [120](#)
 29. M. Oyamaguchi. The reachability and joinability problems for right-ground term rewriting systems. *Journal of Information Processing*, 13(3), pp. 347–354, 1990. [121](#), [124](#)
 30. P. Padawitz. *Computing in Horn Clause Theories*. EATCS Monographs on Theoretical Computer Science, vol. 16. Springer-Verlag, Berlin, 1988. [119](#)
 31. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *J. Logic Program.* 19,20, 261–320. [128](#)
 32. A. Pettorossi and M. Proietti. A comparative revisit of some program transformation techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110: 355–385. Springer-Verlag, Berlin, 1996. [127](#)
 33. A. Pettorossi and M. Proietti. A Theory of Logic Program Specialization and Generalization for Dealing with Input Data Properties. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110: 386–408. Springer-Verlag, Berlin, 1996. [117](#)

34. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993. [119](#)
35. M. Schütz, M. Schmidt-Schauss and S. E. Panitz. Strictness analysis by abstract reduction using a tableau calculus. In A. Mycroft, editor, *Proc. of SAS'95*, LNCS 983:348–365. Springer-Verlag, 1995.
36. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proc. of POPL'99*, pages 291–302, ACM Press, 1999. [123](#), [129](#)

A Class of Decidable Parametric Hybrid Systems

Michaël Adélaïde and Olivier Roux

IRCCyN/CNRS UMR 6597

Institut de Recherche en Communication et Cybernétique de Nantes

1 rue de la Noë, BP 92101, 44321 Nantes cedex 03, France

{michael.adelaide,olivier.roux}@ircyn.ec-nantes.fr

Abstract. Parametric analysis of hybrid systems consists in computing the parameters that enable a correct behaviour of the specified systems. As most problems are not decidable in the theory of hybrid systems, finding decidable parametric systems is of great interest. The so-called class of *weakly controlled hybrid systems* is a new class of parametric systems for which a partition of the space of the parameters can be computed. Firstly, this partition is finite. Secondly, the non-parametric automata induced by fixing two values of a same cell of the partition for each parameter are bisimilar. A parametric hybrid system is weakly-controlled whenever all its elementary cycles have an edge that reset all the variables; the parameters could not be modified.

1 Introduction

Context. Hybrid Automata [Hen96] are a specification and verification model for hybrid systems, which are dynamical systems with both discrete and continuous components [Hen96]. The problem that underlies the safety verification for hybrid automata is reachability: can unsafe states be reached from an initial state by executing the system?

With this traditional analysis, the answers are boolean: “the system is safe (or unsafe)”. Parametric analysis consists in finding the conditions on a special set of variables called the parameters for which the system is safe. Obviously, the problem is more general and its difficulty higher.

Related work. While performing a parametric analysis, the main question is to verify if the decidable properties of the non parametric automata are kept decidable for the parametric automata. The first result is negative [AHV93]: the control-node reachability is not decidable for Parametric Timed Automata (PTA) whereas it is for Timed Automata [AHL00]. The same negative result holds for Slope Parametric Rectangular Automata (SPRA) [Won97] whereas the answer is positive for Initialized Automata¹ [Won97]: an edge must reset a variable whenever the flow specification of its source node and of its target node are different. For Singular Automata, the problem is undecidable [AHL00]. It

¹ SPRA are parametric Initialized Automata.

remains undecidable for the parametric class of Slope Parametric Linear Hybrid Automata (*SPLHA*) [BBRR97].

For Parametric Timed Automata that mix clocks and counters, [AAB00] has developed a semi-decision procedure that prevents from doing “nasty” loops. For each cycle of the automata, it guesses the values of variables and parameters after the cycle being iterated a given number of times. The first positive result about decidability of the Control-Node Reachability problem (*CNR*) in parametric cases is given for Lower Upper Timed Automata *L/UPTA* [HRSV00] which are a subclass of Parametric Timed Automata (*PTA*).

This paper follows [CMP00]. In the non-parametric case, the existence of an edge on every elementary cycle that resets all the variables is sufficient for the forward analysis to complete [CMP00]. They called this class : the class of initialized automata². It is a generalization of the O-minimal hybrid automata depicted in [LPY99] for which the μ -calculus [HM00] is decidable and for which a dedicated algorithm exists. Every edge of the automata [LPY99] deal with resets all the variables.

Objectives. In this paper, we prove that if each cycle has an edge resetting all variables then the forward analysis terminates also *in the parametric case*: we call this class the class of *weakly controlled hybrid automata* in order not to modify the original definition of initialization. The difficulty in the analysis is due to the fact that variables are reset while the parameters are not. Using the tree built by forward-analysis, we tackle the issue of transforming the parametric infinite state system into a finite number of non-parametric finite state systems.

outline. The paper is structured as follows. Firstly, the model is depicted. Parametric hybrid automata and Forward-analysis are defined. Secondly, “Weakly-controlled Hybrid Automata” are introduced. Thirdly, it is shown that Forward-Analysis ends for “Weakly-Controlled Hybrid Automata”. Lastly, a finite partition on the parameters space is built. On each cell of it, all the non-parametric hybrid automata achieved by fixing the parameters are bisimilar.

2 The Model

2.1 Preliminaries

Let $X = \{x_1, \dots, x_n\}$ be a set of indeterminates and $x = (x_1, \dots, x_n)$.

- $\mathbb{Q}[X]$ is the set of polynomials in X with rational coefficients
- Let $Pol(X)$ be the set of polynomial predicates with rational coefficients, i.e. $\Pi \in Pol(x)$ iff Π is a finite union of many applications of disjunction, conjunction and negation operations on elementary predicates of the form $p(x) \leq 0$ where $p \in \mathbb{Q}[X]$.
- let $p \in Pol(X)$ be a predicate whose free variables contain X , $a = (a_1, \dots, a_n)$ be a vector of n polynomials with rational coefficients. $p[x/a]$ is the predicate where all the occurrences of x_i are replaced by a_i .

² However, this class is different from the one depicted in [Won97].

2.2 Syntax

A Parametric Hybrid Automaton H consists of the following components:

Parameters and Variables. Two disjoint finite sets $K = \{k_1, \dots, k_p\}$ of **parameters** and $X = \{x_1, \dots, x_n\}$ of real-numbered **variables**. Let $t \notin K \cup X$ be an additional variable called the **time variable**. Let $\dot{X} = \{\dot{x}_1, \dots, \dot{x}_n\}$ be the set of the first derivatives of the variables (with respect to the time) and $X' = \{x'_1, \dots, x'_n\}$ the set of the post-values of discrete transitions. The vectors associated with the respective sets are $k = (k_1, \dots, k_p)$, $x = (x_1, \dots, x_n)$, $\dot{x} = (\dot{x}_1, \dots, \dot{x}_n)$ and $x' = (x'_1, \dots, x'_n)$.

Control Graph. (V, E) is a finite directed graph. V is its set of vertices (or control-nodes), E is its set of edges.

Initial, Invariant and Flow Predicates. Three predicates are associated on each control-node of the graph [Hen96]. They are defined by the following vertex labelling functions: $init : V \rightarrow Pol(K \cup X')$, $inv : V \rightarrow Pol(K \cup X)$ and $flow : V \rightarrow Pol(K \cup X \cup \dot{X})$.

We suppose that there is one and only one control-node v_0 such that $init(v_0)$ is consistent. The couple $(-\infty, v_0)$ is called the **initial edge associated with** v_0 .

Assume that for each control node v , $flow(v) \stackrel{def}{=} (\dot{x} = A(v, k)x)$ where $A(v, k)$ is a square matrix of size $n \times n$ with coefficients in $\mathbb{Q}[K]$ and assume $A(v, k)$ is nilpotent of index j (i.e. $A(v, k)^j = 0$ and for all $i < j$ $A(v, k)^i \neq 0$). Then $\Gamma(v)(k, x', t) = (\sum_{i=0}^{j-1} \frac{(tA(v, k))^i}{i!})x'$ is called the **continuous trajectory** in the control-node v .

Guard and Action Predicates. Two predicates are given on each edge [Hen96]. They are defined by the following edge labelling functions:

$$guard : E \rightarrow Pol(K \cup X) \text{ and } act : E \rightarrow Pol(K \cup X \cup X').$$

2.3 Semantics

States and Regions. A **state** $\nu = (v, (\kappa, \xi))$ is given by a control node $v \in V$ called the **location** of ν , a couple of vectors $(\kappa, \xi) \in \mathbb{R}^p \times \mathbb{R}^n$ called the **valuation** of ν . The coordinates of κ give the parameters values, the ones of ξ give the variables values. An **initial state** $\nu = (v, (\kappa, \xi))$ is a state such that (κ, ξ) verifies the initial predicate of v_0 i.e. $init(v_0)[k/\kappa, x/\xi]$. A **region** is a set of states associated with the same control-node. In other words, a region is a set $\{v\} \times Q$ where v is a control-node and Q is a subset of \mathbb{R}^{p+n} . Whenever it is possible, Q will be symbolically described by a polynomial predicate.

Continuous Evolutions. Let $\nu_i = (v_i, (\kappa_i, \xi_i))$ be two states of H for $i \in \{1, 2\}$. ν_2 is a **continuous successor** of ν_1 iff:

1. $\kappa_1 = \kappa_2 = \kappa$ and $v_1 = v_2 = v$
2. there exists two positive real numbers $0 \leq t_1 \leq t_2$ and $\xi_0 \in \mathbb{R}^n$ such that :
 $\xi_i = \Gamma(v)[k/\kappa, x'/\xi_0, t/t_i]$ for $i \in \{1, 2\}$ and $\forall t \in [0, t_2] : \text{inv}(v)[k/\kappa, x'/\xi_0]$.

Given a control-node v and a predicate $\Pi' \in \text{Pol}(K \cup X')$, the **extension** function $\text{ext} : V \times \text{Pol}(K \cup X') \rightarrow \text{Pol}(K \cup X)$ computes the predicate that gives the values of the continuous successors of the region $\{v\} \times \Pi'$:

$$\text{ext}(v, \Pi') = \exists \delta. \delta \geq 0. \exists x'_1 \dots \exists x'_n. \Pi' \wedge \forall t. ((0 \leq t \leq \delta) \Rightarrow \text{inv}(v)[x/\Gamma(v)(k, t)]).$$

ext can be fully depicted via quantifier elimination operations in real algebra [GVRRT95].

Discrete Transitions. Let $\nu_i = (v_i, (\kappa_i, \xi_i))$ be two states of H for $i \in \{1, 2\}$. ν_2 is a **discrete successor** of ν_1 iff:

1. $\kappa_1 = \kappa_2 = \kappa$
2. there exists an edge e of H of source v_1 and target v_2 such that:
 $(\text{inv}(v_1) \wedge \text{guard}(e))[k/\kappa, x/\xi_1]$, $\text{act}(e)[k/\kappa, x/\xi_1, x'/\xi_2]$ and
 $\text{inv}(v_2)[k/\kappa, x/\xi_2]$.

Given a node v_1 , a predicate $\Pi \in \text{Pol}(K \cup X)$ and an edge e of source v_1 and target v_2 , the **jump** function $\text{jump} : \text{Pol}(K \cup X) \times E \rightarrow \text{Pol}(K \cup X')$ computes the predicate that gives the values of the discrete successors of $\{v_1\} \times \Pi$ after having crossed e :

$$\text{jump}(\Pi, e) = (\exists x_1 \dots \exists x_n. (\Pi \wedge \text{guard}(e) \wedge \text{act}(e))) \wedge (\text{inv}(v')[x/x']).$$

jump can be fully depicted via quantifier elimination operations in real algebra [GVRRT95].

Successors. Given a control-node v_1 , a region $R_1 = \{v_1\} \times \Pi'_1$ (when the control enters v_1) and an edge e (of source v_1 and target v_2), the **successors** $R_2 = \{v_2\} \times \Pi'_2$ of R_1 after elapsing time in v_1 and crossing e are given by the function $\text{post} : \text{Pol}(K \cup X) \times E \rightarrow \text{Pol}(K \cup X)$; thus:

$$\Pi'_2 = \text{post}(\Pi'_1, e) = \text{jump}(\text{ext}(v_1, \Pi'_1), e).$$

Trajectories and Reachable States. A trajectory is a concatenation of continuous evolutions of durations δ_i and discrete jumps on e_i from an initial state ν_0 . A reachable state is a target of a trajectory. Let $\text{Reach}(H)$ be the set of the reachable states of H .

Iterating Post. While following a path, the function post of successors needs to be iterated. A **word** is a sequence $e_0 \dots e_d$ of edges that corresponds

1. either to the path $v_0 \xrightarrow{e_1} v_1 \dots v_{i-1} \xrightarrow{e_i} v_i \dots \xrightarrow{e_d} v_d$ of (V, E) when e_0 is the initial edge $(-\infty, v_0)$

2. or to the path $v_0 \xrightarrow{e_0} v_1 \dots v_{i-1} \xrightarrow{e_i} v_i \dots \xrightarrow{e_d} v_d$ otherwise.

Let $words(H)$ be the set of words of H . $post$ is extended to words as follows, $Post : Pol(K \cup X) \times words(H) \rightarrow Pol(K \cup X)$. For any word w and any word σ such that σw is a word using the following rules: (1) $Post(\Pi, \epsilon) = \Pi$ if ϵ is the word of null length, (2) $Post(\Pi, \sigma) = init(v_0)$ if $\sigma = (-\infty, v_0)$ is an initial edge, (3) $Post(\Pi, \sigma) = post(\Pi, \sigma)$ if $\sigma \in E$, (4) $Post(\Pi, \sigma w) = Post(post(\Pi, \sigma), w)$.

2.4 Forward Analysis

The first problem to answer is to compute $Reach(H)$, the **set of the reachable states** of a parametric hybrid automaton H . The method used here is called Forward-Analysis [AR00]. It consists in calculating the reachable states obtained after having crossed any number of edges. It builds a possibly infinite labelled tree.

Every edge L of the tree $Reach(H)$ is labelled with an edge $e \in E$ and a predicate $\Pi' \in Pol(K \cup X')$ which represents the values of the variables immediately after the control crossed e (equivalently, Π' represents the values of the variables when entering $target(e)$).

Every node N of target of $Reach(H)$ is labelled with a region $\{v\} \times ext(v, \Pi')$ where v is the target of an edge e ; e and Π' are the labels of the edge L that leads to N .

The computations can be done using $Post$. Let N be a node of $Reach(H)$, L_0 be the edge that leads to N (if N is the root of $Reach(H)$, L_0 is the “initial-edge” of the tree and it is labelled with the initial edge $(-\infty, v_0)$ and the predicate $init(v_0)$) and L_1 be an edge from N (we suppose here that N has a son). Let (e_i, Π'_i) be the labels of L_i . By construction, $\Pi'_1 = post(\Pi'_0, e_1)$.

3 Weakly-Controlled Hybrid Automata

In this section, a sufficient criterion on special edges, the weakly-controlled edges, is given for the forward analysis to terminate : the “Weakly-controlled” hypothesis.

3.1 Definition

A parametric hybrid automaton H is weakly-controlled iff all the cycles of its support-graph contain a weakly-controlled edge.

The predicate $act(e)$ links the values of the variables in $source(e)$ with the ones after e having been crossed. For instance, if the specifications on e are : “set x_1 to 3 and add 2 to x_2 ” then $act(e) \stackrel{def}{=} ((x'_1 = 3) \wedge (x'_2 = x_2 + 2))$.

An edge e of H is **weakly-controlled** iff $act(e) \in Pol(k \cup X')$ (i.e. e resets all the variables). $act(e) \wedge inv(target(e))[x/x']$ is called the **weak-control** value of e .

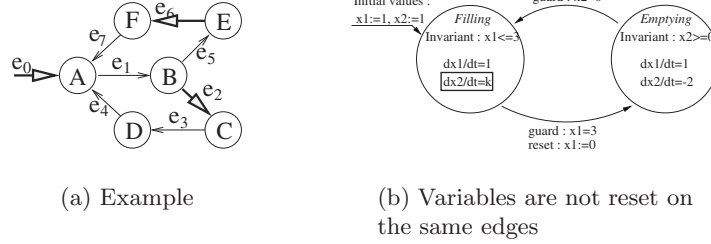


Fig. 1. Examples

3.2 Remarks

Reset on the Same Edge. In section 4, we prove that forward analysis completes for weakly-controlled automata. In Fig.1(a), assume that the edges e_2 and e_6 reset all the variables. This automaton is weakly-controlled. The example of Fig.1(b) [AR00] shows that forward-analysis is not guaranteed to complete if the variables are not reset on the same edge. x_1 and x_2 are reset on the cycle but not on the same edge. The values of k that allow incoming in ‘filling node’ are in an infinite sequence of intervals $[0, M_n]$ where $\lim_{n \rightarrow +\infty} M_n = 2$.

Synchronized Product. In [CMP00], it is shown that the synchronized product of two initialized automata is initialized whenever the synchronized edges reset all the variables. Consequently, the synchronized product of the automata of the Philips audio protocol of communication given in [DY95] is weakly-controlled and this makes our algorithm (section 5) convenient for determining the maximum clock divergence for the protocol. [Won97] finds the condition on the clocks using a semi-algorithm but no reason are given to explain the completion of the method.

3.3 Weakly-Controlled and Forward Analysis

Crossing Conditions. Let e be an edge of E of source v_1 and target v_2 , and Π'_1 be a predicate of $Pol(K \cup X')$. The crossing conditions of Π'_1 for e are the conditions on the parameters such that v_2 is reachable from $\{v_1\} \times \Pi'_1$, i.e. $\exists x'_1 \dots \exists x'_n. post(\Pi'_1, e)$. They are given by the function $p : Pol(K \cup X') \times E \rightarrow Pol(K)$ defined by:

$$p(\Pi'_1, e) = \exists x'_1 \dots \exists x'_n. post(\Pi'_1, e).$$

Crossing Weakly-Controlled Edges. The following theorem gives the post-values of the variables after crossing a weakly-controlled edge.

Theorem 1 *If $e \in E$ is a weakly-controlled edge, then for every $\Pi' \in Pol(K \cup X')$:*

$$post(\Pi', e) = p(\Pi', e) \wedge act(e).$$

Proof. We compute $post(\Pi', e)$ for $\Pi' \in Pol(K \cup X)$.

Let $\Pi_1 = ext(source(e), \Pi')$. Then, $post(\Pi', e) = jump(\Pi_1, e)$ with $jump(\Pi_1, e) = (\exists x'_1 \dots \exists x'_n. (\Pi_1 \wedge guard(e) \wedge act(e))) \wedge inv(v')[x/x']$.

We can write $act(e) = act(e) \wedge act(e)$ and get one predicate $act(e)$ out of the quantified predicate (as $act(e)$ does not depend on X'): $jump(\Pi_1, e) = (\exists x'_1 \dots \exists x'_n. (\Pi_1 \wedge guard(e) \wedge act(e))) \wedge \Phi(e)$ where $\phi(e) = act(e) \wedge inv(v')[x/x']$. We can write $inv(v')[x/x'] = inv(v')[x/x'] \wedge inv(v')[x/x']$. Thus, $post(\Pi', e) = p(\Pi', e) \wedge \Phi(e)$. Consequently, e is weakly controlled. Its weakly-control value is $\Phi(e)$. \square

4 Computing the Reachable States of a Weakly-Controlled Hybrid Automaton

Forward Analysis builds a tree the edges L of which are labelled with edges $e \in E$ and predicates $\Pi' \in Pol(K_{cup} X')$. The region $\{target(e)\} \times Pi[x'/x]$ is called a **region entering target(e)**. Note that many region can incoming a same control-node v : its depends on the the path taken to go to v . The following theorem sets that the number of different entering regions built by Forward Analysis is finite.

Theorem 2 *Let H be a weakly-controlled hybrid automaton. Then, there exists a bound M such that for all reachable entering regions R built by Forward-Analysis, there is a path γ leading to R with $length(\gamma) \leq M$.*

Remarks. If the problem is only Control-Node Reachability (given a control-node $v \in V$, the Control-Node Reachability seeks all the valuations $(\kappa, \xi) \in \mathbb{R}^p \times \mathbb{R}^n$ such that the state $(v, (\kappa, \xi))$ is reachable), this theorem is useless. Actually, when computing the tree of the reachable states, if a weakly-controlled edge e (which weak-control value is $\Phi(e)$), is found for the second time then the tree can be cut. Let $\Pi'_i = C_i \wedge \Phi(e)$ be the predicates that define the region after crossing e for the i^{th} time along a branch. then, $C_i \subset C_{i+1}$. Only $\Pi'_i \subset \Pi'_{i+1}$ holds; which is sufficient to achieve Control-Node Reachability.

Nevertheless, the property given by theorem 2 is useful. It will be used in the last section of the paper.

4.1 Proof of the Theorem

Let R be a reachable region and γ a path leading to R . The path of minimal length leading to a reachable region R is called the **minimal path** leading to R . It is denoted $\gamma_m(R)$.

If $\gamma_m(R)$ does not contain weakly-controlled edges, then $length(\gamma_m(R)) \leq |V|$ as all the cycles of H contain at least one weakly-controlled edge. Consequently, it is sufficient to prove that the number of weakly-controlled regions is finite. Actually, if it is the case, a bound M_{wc} exists for minimal paths leading to weakly-controlled regions, and $M_{wc} + |V|$ is a bound for all minimal paths.

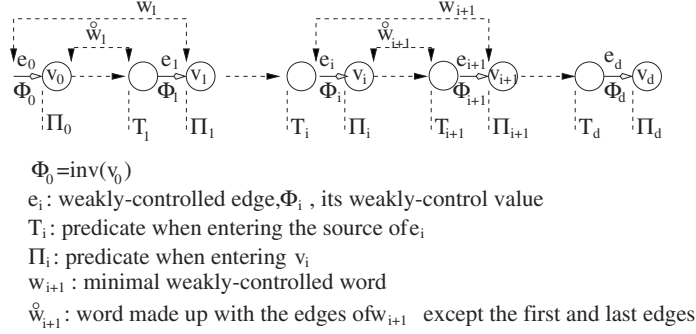


Fig. 2. Decomposition into words

Then, only weakly-controlled regions R are considered from now on. Any path γ leading to R contains at least one weakly-controlled edge. The idea developed in the proof is to look at the valuations of the weakly-controlled regions preceding R in γ and what happens between two successive weakly-controlled regions, studying the so-called minimal weakly-controlled words. A word $w = e_1 \dots e_p$ of H is **weakly-controlled** iff:

1. either e_1 is an initial edge and e_p is a weakly-controlled edge
2. or both e_1 and e_p are weakly-controlled edges

A weakly-controlled word $e_1 \dots e_d$ is **minimal** iff for all $i \in \{2, \dots, p-1\}$, e_i is not a weakly-controlled edge.

Preliminaries. In Fig.2, edges e_i are weakly-controlled, Φ_i are their weak-control values. Π'_i are the predicates verified by the regions incoming in the nodes v_i , T'_i are the predicates verified by the regions incoming in $source(e_i)$ (with the convention $T'_0 = \Pi'_0 = \Phi(e_0) = \text{init}(v_0)$); it means that $\Pi'_i = p(T'_i, e) \wedge \Phi_i$. The words w_i start from e_{i-1} to e_i . The words \hat{w}_i are the words made up from the words w_i except their first and last edges; $w_i = e_{i-1} \cdot \hat{w}_i \cdot e_i$.

Iterating Post. The main property of *Post* is the following lemma. It uses the expression of *ext* and *jump* with variables elimination.

Lemma 1 *Let H be a parametric hybrid automaton with polynomial flow or a PRA. Let $C \in \text{Pol}(K)$, $\Pi' \in \text{Pol}(K \cup X')$, $w \in \text{words}(H)$, then*

$$\text{Post}(C \wedge \Pi', w) = C \wedge \text{Post}(\Pi', w) \text{ and } p(C \wedge \Pi', w) = C \wedge p(\Pi', w).$$

The proof is an easy induction on the length of w . Actually, for any condition C on the parameters, any edge e and any predicate $\Pi' \in \text{Pol}(K \cup X')$ $\text{post}(C \wedge \Pi', e) = C \wedge \text{post}(\Pi', e)$ since parameters are never quantified. \square

Crossing a Minimal Weakly-Controlled Word. The following lemma computes the predicate Π'_{i+1} of the region after having crossed e_{i+1} .

Lemma 2 *With the notations defined in the preliminaries and according to the definition of the crossing conditions (section 4.3):*

$$\Pi'_{i+1} = p(\text{Post}(\Pi'_i, \overset{\circ}{w}_{i+1}), e_{i+1}) \wedge \Phi_{i+1}.$$

Lemma 2 is proven as follows. After having crossed $\overset{\circ}{w}_{i+1}$ (i.e. before crossing e_{i+1}), $T'_{i+1} = \text{Post}(\Pi'_i, \overset{\circ}{w}_{i+1})$. As e_{i+1} is weakly-controlled, $\Pi'_{i+1} = p(T'_{i+1}, e_{i+1}) \wedge \Phi_{i+1}$. Thus, $\Pi'_{i+1} = p(\text{Post}(\Pi'_i, \overset{\circ}{w}_{i+1}), e_{i+1}) \wedge \Phi_{i+1}$. \square

Crossing Successive Words. The following lemma computes the predicate Π'_d of the region after having crossed e_d with all the encountered Φ_i .

Lemma 3 *With the notations defined in the preliminaries*

$$\Pi'_d = \left(\bigwedge_{i=0}^{d-1} p(\text{Post}(\Phi_i, \overset{\circ}{w}_{i+1}), e_{i+1}) \right) \wedge \Phi_d.$$

This lemma is proved by induction on the number d of crossed minimal weakly-controlled words. When $d = 1$, it the preceding lemma holds since $\Pi'_0 = \Phi_0$ (for the initial transition, the weak-control value and the predicate are the same). Suppose now that for a given d , the lemma 3 holds: $\Pi'_d = \left(\bigwedge_{i=0}^{d-1} p(\text{Post}(\Phi_i, \overset{\circ}{w}_{i+1}), e_{i+1}) \right) \wedge \Phi_d$.

While crossing $\overset{\circ}{w}_{d+1}$, using lemma 2: $\Pi'_{d+1} = p(\text{Post}(\Pi'_d, \overset{\circ}{w}_{d+1}), e_{d+1}) \wedge \Phi_{d+1}$.

Using the value of Π'_d given by the induction hypothesis (and using lemma 1 as p ranges over $\text{Pol}(K)$): $p(\text{Post}(\Pi'_d, \overset{\circ}{w}_{d+1}), e_{d+1}) = C \wedge p(\text{Post}(\Phi_d, \overset{\circ}{w}_{d+1}), e_{d+1})$.

where $C = \left(\bigwedge_{i=0}^{d-1} p(\text{Post}(\Phi_{i+1}, \overset{\circ}{w}_i), e_{i+1}) \right)$.

It follows that: $\Pi'_d = \left(\bigwedge_{i=0}^{d-1} p(\text{Post}(\Phi_i, \overset{\circ}{w}_{i+1}), e_{i+1}) \right) \wedge \Phi_d$. \square

Finiteness. The following lemma gives a good criterion to know when a weakly-controlled region has been already found. It only uses the minimal weakly-controlled words, and it makes no inclusion test.

Lemma 4 *If there exists $i < d$ such that $w_i = w_d$, then the following holds:*

1. *if w_i is a cycle then $w_d = w_{d-1}$*
2. *if for all k such that $i < k < d$, there exists $j \leq i$ with $w_j = w_k$, then $\Pi'_i = \Pi'_d$.*

This lemma is a consequence of lemma 3, the commutativity of the predicates conjunction and the fact that $\pi \wedge \pi = \pi$ for any predicate π . \square

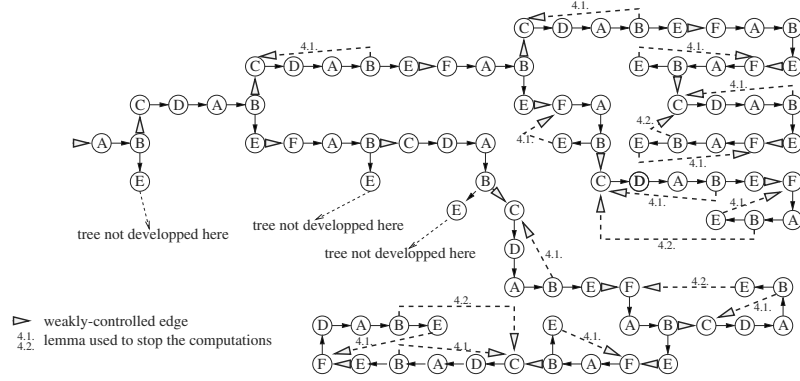


Fig. 3. Tree computed after applying theorem 3

End of the Proof. To end the proof, it is sufficient to find a bound for the lengths of the words $w = w_1 \dots w_d$ for which lemma 4.2. cannot be applied for any $w_1 \dots w_i$. let $w = w_1 \dots w_d$ with $d = (|mwcwords(H)| + 1)^2$. Let $D(w, w_i) = \{j \leq d | w_i = w_j\}$. There exists j such that $|D(w, w_j)| \geq |mwcwords(H)| + 1$. Then, one element of $D(w, w_j)$ verifies the hypothesis of lemma 4.2. If it was not the case, with $i_1 < i_2 < \dots < i_t$ the ordered elements of $D(w, w_j)$, for all $l \leq t - 1$ there exists k with $i_l < k < i_{l+1}$ and for all $m < k$, $w_k \neq w_m$. This means that $|mwcwords(H)| \geq |D(w, w_i)|$ which is impossible. \square

4.2 Remarks

Lemma 4 is applied to the example of Fig.1(a). The result is the tree of Fig.3. Lemma 4 prevents the algorithm from computing tests of inclusion of regions. Actually, it gives a criterion to know if two regions are equal.

5 Finding Bisimulation

5.1 Statements

Sate Equivalences. State equivalences [HM00] play an important role in verification. Given a transition system $S = (Q, \rightarrow)$ and a state equivalence \simeq , the **quotient transition system** [HM00] is built: its states are the equivalence classes of \simeq and an equivalence class T_2 is the successor of another T_1 denoted $T_1 \rightarrow /_{\simeq} T_2$ iff there exists $t_1 \in T_1$ and $t_2 \in T_2$ with $t_1 \rightarrow t_2$. When \simeq has finite index, the quotient construction transforms an infinite transition system into a finite one. The model-checking [HM00] is applied to the quotient system. The most interesting state equivalence is bisimilarity since the μ -calculus is decidable iff bisimilarity has finite index [HM00].

Bisimilarity. A binary relation \preceq on the states of an **non**-parametric hybrid automaton A is a simulation [HM00] if for any states s and t , $s \preceq t$ implies the two following conditions:

1. s and t are associated with the same control-node
2. for each successor s' of s there exists a successor t' of t such that $s' \preceq t'$

If \preceq is symmetric, it is called a **bisimilarity** and denoted \simeq_1 . The class of transition systems for which bisimilarity has finite index is of great interest.

Bisimilarity and Weakly-Controlled Hybrid Systems. If on every cycle of an hybrid system H , there exists an edge that reset all the variables, then \simeq_1 admits a finite index [KPSY99]. This condition is a relaxation of resetting all variables at every edge [LPY99]. It is more restrictive than the “weak-control” hypothesis because parameters cannot be reset. Furthermore, when a “weakly-controlled edge is crossed many times the conditions change. For instance, in the automaton of Fig:1 (a), the predicates that describe the states after crossing e_0e_2 and $e_0e_2e_6e_2$ are different as the minimal weakly-controlled words crossed are not the same. Consequently, the criterion cannot be applied to parametric weakly-controlled hybrid automata but only to every non-parametric hybrid automaton $H[k/\kappa]$ for $\kappa \in \mathbb{R}^p$. The idea for the analysis of the parametric hybrid systems is to compute a finite number of non-parametric hybrid automata $H[k/\kappa]/\simeq_1$ that characterize completely the parametric automata H .

Bisimulation. Let $P_i = (S_i, \Sigma_i, \rightarrow_i, q_i)$ for $i \in \{1, 2\}$ two labelled transition systems. The relation $\sim_C S_1 \times S_2$ is a **strong bisimulation** iff :

1. if $s_1 \sim s_2$ and $s_1 \xrightarrow{a}_1 t_1$ then there is $t_2 \in S_2$ such that $t_1 \sim t_2$ and $s_2 \xrightarrow{a}_2 t_2$
2. if $s_1 \sim s_2$ and $s_2 \xrightarrow{a}_2 t_2$ then there is $t_1 \in S_1$ such that $t_1 \sim t_2$ and $s_1 \xrightarrow{a}_1 t_1$

P_1 and P_2 are strongly bisimilar, denoted $P_1 =_{sb} P_2$ iff $\Sigma_1 = \Sigma_2$ and there is a strong bisimulation $\sim_C S_1 \times S_2$ such that $q_1 \sim q_2$.

The following theorem asserts the decidability of finding bisimulations for weakly-controlled parametric hybrid automata:

Theorem 3 *Let H be a weakly-controlled hybrid automaton. There exists a finite partition \mathcal{P} of \mathbb{R}^p such that if κ_1 and κ_2 belong to the same cell of \mathcal{P} then $H[k/\kappa_1]$ and $H[k/\kappa_2]$ are strongly bisimilar.*

5.2 Cylindrical Algebraic Decomposition

The proof is based upon the Cylindrical Algebraic Decomposition.

Definition. Let $Y = [y_1, \dots, y_n]$. A **Cylindrical Algebraic Decomposition** (CAD) [Jir95] $cad(\mathcal{P}, Y, n)$ of \mathbb{R}^n is a decomposition \mathcal{D} of \mathbb{R}^n into disjoint regions (a region of \mathbb{R}^n is a connected subset of \mathbb{R}^n) over which every polynomial (in $\mathbb{R}[Y]$) of a set \mathcal{P} has a constant sign. The decomposition is **algebraic** [Jir95] i.e. every region can be constructed by finitely many applications of union, intersection and complementation operations on sets of the form $\{y : f(y) \leq 0\}$ where $f \in \mathbb{R}[y]$. The decomposition is **cylindrical** i.e.:

- if $n = 1$ then \mathcal{D} is a partition of \mathbb{R} into a finite set of numbers and open intervals bounded by these numbers,
- if $n \geq 2$ then there exists a CAD \mathcal{D}' of \mathbb{R}^{n-1} such that: for every region \mathcal{R}' of \mathcal{D}' , there exists an integer j and $j + 1$ functions $q_i : \mathcal{R}' \rightarrow \overline{\mathbb{R}}$ such that $q_0 = -\infty$, $q_j = +\infty$ and $q_i < q_{i+1}$ on \mathcal{R}' for all $i \in \{0, \dots, j-1\}$. Then, the built regions of \mathcal{D} have the form $\mathcal{R}' \wedge \{y | (q_i(y_1, \dots, y_{j-1}) < y_j < q_{i+1}(y_1, \dots, y_{j-1})) \wedge (y_1, \dots, y_{j-1}) \in \mathcal{R}'\}$ for all $i \in \{0, \dots, j-1\}$ or $\mathcal{R}' \wedge \{(y_1, \dots, y_{j-1}), q_j(y_1, \dots, y_{j-1}) | (y_1, \dots, y_{j-1}) \in \mathcal{R}'\}$. \mathcal{R}' is said to be a **support** for the cells of \mathcal{R} .

Construction. Given a family P of polynomials in $\mathbb{R}[y_1, \dots, y_n]$, the algorithm processes runs as follows:

1. if $n = 1$, then the roots of the polynomial are sorted and the decomposition is given by these numbers and the intervals between them
2. if $n > 1$ then:
 - (a) a family P' of polynomials in $\mathbb{R}[y_1, \dots, y_{n-1}]$ is computed: it is the **projection phase**
 - (b) a CAD \mathcal{D}' of \mathbb{R}^{n-1} is found for P' : it is the **base phase**
 - (c) a CAD \mathcal{D} of \mathbb{R}^n is built from \mathcal{D}' : it is the **lift phase**

An iterative algorithm can be used: it first computes all the projection sets and it finally builds all the CAD. Let $cad(\mathcal{P}, Y, p)$ be the CAD of \mathbb{R}^p built after the p^{th} lift phase. Let Π be a predicate the polynomials of which belong to \mathcal{P} $cells(cad(\mathcal{P}, Y, p), \Pi)$ be the cells of $cad(\mathcal{P}, Y, p)$ on which the predicate Π holds.

Quantifier Elimination Using CAD. Let \mathcal{P} be a set of polynomials in $\mathbb{R}[Y]$ and let $Y = [y_1, \dots, y_n]$. Let Π be a predicate the elementar Then $\exists y_n. \Pi$ is the union of the cells \mathcal{R}' of $cad(\mathcal{P}, Y, n-1)$ which are the support of the cells of $cells(cad(\mathcal{P}, Y, p), \Pi)$.

5.3 Partition of \mathbb{R}^p

Every node N of the tree $Reach(H)$ is given a local time variable t_N and a set of variables $X'(N) = \{x'(N)_1, \dots, X'(N)_n\}$. The vector $x'(N) = (x'(N)_1, \dots, x'(N)_n)$ represents the values of the variables when “incoming” in N . New predicates are created as follows.

For every node N (labelled with v and Π) of $Reach(H)$:

$$\begin{aligned} inv(N) &\stackrel{def}{=} inv(v)[x'/x'(N), x/\Gamma_v[x'/x'(N), t/t(N)]], \\ init(N) &\stackrel{def}{=} init(v)[x'/x'(N)] \wedge (t(N) = 0), \Pi(N) \stackrel{def}{=} \Pi[x/\Gamma_v[x'/x'(N), t/t(N)]] \\ &\text{and } t(N) = 0. \end{aligned}$$

For every edge L (labelled with e and Π' , of source v_1 and target v_2) that links the node N_1 to the node N_2 : $guard(L) \stackrel{def}{=} guard(e)[x/\Gamma_v[x'/x'(N_1), t/t(N_1)]]$, $act(L) \stackrel{def}{=} act(e)[x'/\Gamma_v[x/x(N_1), t/t(N_1), x'/x'(N_2)]] \wedge (t(N_2) = 0)$ and $\Pi'(N) \stackrel{def}{=} \Pi[x'/x'(N_2)] \wedge (t(N_2) = 0)$.

Then, the polynomials of the new predicates are stored in a list \mathcal{P} .

We suppose now that the nodes of the tree are indexed and their index is the prefix order. Let $Y = K \cdot [x_1^{N_1}, \dots, x_n^{N_1}, t_{N_1}] \cdot \dots \cdot [x_1^{N_{|N|}}, \dots, x_n^{N_{|N|}}, t_{N_{|N|}}]$. Let $\mathcal{D}_0 = cad((P), Y, p)$ and $\mathcal{D}_i = cad((P), Y, p + 2i)$ for $i \in \{2, \dots, n\}$. The partition \mathcal{D}_0 of \mathbb{R}^p solves the problem.

5.4 State Equivalence \sim

Blocks. A block is a cell \mathcal{R} of $\mathcal{D}_i = cad((P), Y, p + 2i)$ on which $\Pi(N_i)$ holds. Let $\vec{y}_i = (x'(N_1)_1, \dots, x'(N_1)_n, t(N_1), \dots, x'(N_{|N|})_1, \dots, x'(N_{|N|})_n, t(N_{|N|}))$. The states represented in \mathcal{R} are the $(v_i, (\kappa, \xi))$ where $v_i \in V$ is the controlled-node that labels N_i and $\exists \vec{y}_i \exists x'(N_i) \exists t(N_i). (\mathcal{R} \wedge (\xi = \Gamma_{v_i}(\kappa, t(N_i))))$.

State Equivalence. For two states ν_1 and ν_2 , $\nu_1 \sim \nu_2$ iff:

1. either both of the states are not reachable
2. or there exists an integer i and a block \mathcal{R} such that $\nu_1 \in \mathcal{R}$ and $\nu_2 \in \mathcal{R}$.

Two remarks have to be done here. Firstly, every state of $Reach(H)$ belongs to a block since \mathcal{D}_i covers \mathbb{R}^{p+2i} in all. Secondly, the number of blocks is finite.

Continuous Successor of a Block. Let $\mathcal{R} \in cells(\mathcal{D}_i, \Pi(N_i))$. As \mathcal{R} is cylindrical (section 5.2), either $\mathcal{R} = c(k, \vec{y}_i, x'(N_i)) \wedge (t(N_i) = q(k, \vec{y}_i, x'(N_i)))$ or $\mathcal{R} = c(k, \vec{y}_i, x'(N_i)) \wedge (q(k, \vec{y}_i, x'(N_i)) < t(N_i) < q'(k, \vec{y}_i, x'(N_i)))$. In the first case, the if the cell $\mathcal{R}' = c(k, \vec{y}_i, x'(N_i)) \wedge (q(k, \vec{y}_i, x'(N_i)) < t(N_i) < q'(k, \vec{y}_i, x'(N_i)))$ verify $\Pi(N_i)$, it is a continuous successor of \mathcal{R} . In the second case, the cell to be studied is $\mathcal{R}' = c(k, \vec{y}_i, x'(N_i)) \wedge (t(N_i) = q'(k, \vec{y}_i, x'(N_i)))$.

Discrete Successors of a Block in the Tree . Let $\mathcal{R} \in cells(\mathcal{D}_i, \Pi(N_i))$. Let L be an edge $Reach(H)$ of source N_i and target N_j , the discrete successors of $Reach(H)$ are given by $jump$ (section 2.3):

$$\exists x(N_i). ((\exists \vec{y}_i \exists t(N_i) \exists x'(N_i). ((\mathcal{R} \wedge (x(N_i) - \Gamma_{v_i} = 0) \wedge \Pi(N_i))) \wedge guard(L) \wedge act(L)) \wedge inv(N_j)).$$

$\exists x(N_i)$ can be easily computed thanks to $\exists x(N_i) - \Gamma_{v_i} = 0$. Furthermore, the predicates $guard(L)$, $act(L)$ and $inv(N_j)$ do not depend on the quantified variables. Then, the successors are given by:

$$\exists \vec{y}_i \exists t(N_i) \exists x'(N_i). ((R) \wedge \Pi(N_i) \wedge guard(L) \wedge act(L) \wedge inv(N_j)).$$

Finally, the successors are given by: $\exists \vec{y}_j. ((R) \wedge \Pi(N_i) \wedge guard(L) \wedge act(L) \wedge inv(N_j))$.

Let $\mathcal{C} = cells(\mathcal{D}_j, (R) \wedge \Pi(N_i) \wedge guard(L) \wedge act(L) \wedge inv(N_j))$. The elements of \mathcal{C} are the successors of \mathcal{R} . Let $\mathcal{R}' \in \mathcal{C}$. The states in \mathcal{R}' are given by $(v_j, (\kappa, \xi))$ where $v_j \in V$ is the controlled-node that labels N_j , $\exists \vec{y}_j \exists x'(N_j) \exists t(N_j). (\mathcal{R}' \wedge (\xi = \Gamma_{v_j}(\kappa, t(N_j))))$.

Discrete Successor of a Block which Are Not in the Tree. The remaining case is the one in which a weakly-controlled edge e leads to a node N_j with $j < i$ (dotted edges in Fig.3). Suppose that the edge L of source N_h and target N_i is associated the weakly controlled edge e . New variables (but the least ones) $X'' = \{x''_1, \dots, x''_n\}$ and t'' are created.

The following predicates $inv''(N_j) \stackrel{def}{=} inv(N_j)[x'(N_j)/x'', t(N_j)/t'']$, $t'' = 0$, $guard'' \stackrel{def}{=} guard(L)[t(N_j)/t'']$ and $act'' \stackrel{def}{=} act(L)[x(N_h)/x(N_i), x'(N_j)/X(N_i)]$ are created.

Then, the discrete successors of \mathcal{R} via L are given by $\exists \vec{y}_{i+1}. ((R) \wedge \Pi(N_i) \wedge guard(L)) \wedge (act'' \wedge inv'')$.

If (R) verify $guard(L)$ then, after all the variables be eliminated, the successors are $c_p(k) \wedge (act'' \wedge inv'')$ where $\mathcal{R} = c_p(k) \wedge c_{p+2.i}(k, \vec{y}_i, x'(N_i), t(N_i))$. Thus, cells of $cells(\mathcal{D}_j, act(N_j) \wedge inv(N_j))$ are the discrete successors of (R) .

Conclusion. The cells of \mathcal{D}_0 are the support of all the computed *CAD*. It follows that, whenever κ_1 and κ_2 belong to the same cell of \mathcal{D}_0 , The automata $H[k/\kappa_i]$ are strongly bisimilar. \square

6 Conclusions and Perspectives

In this paper, weakly-controlled hybrid systems which are parametric hybrid systems have been depicted. A hybrid system is weakly-controlled whenever all its elementary cycles have an edge that reset all the variables. The parameters are never modified. The class of weakly-controlled hybrid system is closed under the synchronized product over weakly-controlled edges.

The main result is based upon the completion of the forward analysis for weakly-controlled automata. Actually, a finite partition \mathcal{P} of the parameters space can be built. On each cell of \mathcal{P} , the non-parametric automata induced are bisimilar.

To have an implementation, new algebraic tools are needed. If the model considered is ‘‘Slope Parametric Linear Hybrid Automata’’ [BBRR97], parametric linear tools [AR00] are needed. The complete algorithm is being implemented.

References

- [AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. 12th Intern. Conf. on Computer Aided Verification (CAV'00)*, LNCS 1855. Springer-Verlag, July 2000. 133
- [AHLP00] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE 88 (2)*, pages 971–984, 2000. 132
- [AHV93] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proc. of the 25th Annual ACM Symposium on Theory of Computing, STOC'93*, pages 592–601, 1993. 132
- [AR00] M. Adélaïde and O. Roux. Using cylindrical algebraic decomposition for the analysis of slope parametric hybrid automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems 2000*. Springer-Verlag, september 18-22 2000. 136, 137, 145
- [BBRR97] F. Boniol, A. Burgueño, O. Roux, and V. Rusu. Analysis of slope-parametric hybrid automata. In O. Maler, editor, *Proc. of the International Workshop on Real time and Hybrid Systems, HART 97*, pages 75–80. Springer-Verlag, March 26-28 1997. 133, 145
- [CMP00] W. Charatonik, S. Mukhopadhyay, and A. Podelski. Compositional termination analysis of symbolic forward analysis. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2000. 133, 137
- [DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proc. IEEE RTSS'95*, december 5–7 1995. 137
- [GVRRT95] Laureano Gonzales-Vega, Fabrice Rouillier, Marie-Françoise Roy, and Guadalupe Trujillo. *Some Tapas of Computer Algebra*. Eindhoven University of Technology, a.m. cohen and h. cuypers and h. sterk edition, 1995. 135
- [Hen96] T. Henzinger. The theory of hybrid automata. In *IEEE Symposium on Logic In Computer Science*, pages 278–282, 1996. 132, 134
- [HM00] T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *Proceedings of the 17th International Conference on Theoretical Aspects of Computer Science (STACS 2000)*, pages 13–34. Springer-Verlag, 2000. 133, 141, 142
- [HRSV00] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. Technical report, BRICS, 2000. 133
- [Jir95] M. Jirstrand. Cylindrical algebraic decomposition - an introduction. Technical report, Computer Algebra Information Network, Europe, Departement of Electrical Engineering, Linköping university, S-581 83 Linköping. Sweden, October 1995. 143
- [KPSY99] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Decidable integration graphs. In *Information and Computation*, Vol. 150, No. 2, pages 209–243. Springer-Verlag, may 1999. 142
- [LPY99] G. Lafferriere, G.J. Pappas, and S. Yovine. A new class of decidable hybrid systems. In F. W. Vaandrager and J. H. van Schuppen, editors, *Hybrid Systems, Computation and Control*, volume 1569 of LNCS, pages 137–151. Springer-Verlag, 1999. 133, 142
- [Won97] H. Wong-Toi. Analysis of slope-parametric rectangular automata. In *Hybrid Systems V*. Springer-Verlag, 1997. 132, 133, 137

Vacuity Checking in the Modal Mu-Calculus^{*}

Yifei Dong, Beata Sarna-Starosta, C.R. Ramakrishnan, and Scott A. Smolka

Department of Computer Science, State University of New York at Stony Brook
Stony Brook, NY, 11794-4400, USA
{ydong,bss,cram,sas}@cs.sunysb.edu

Abstract. Vacuity arises when a logical formula is trivially true in a given model due, for example, to antecedent failure. Beer et al. have recently introduced a logic-independent notion of vacuity and shown that certain logics, i.e., those with polarity, admit an efficient decision procedure for vacuity detection. We show that the modal mu-calculus, a very expressive temporal logic, is a logic with polarity and hence the results of Beer et al. are applicable. We also extend the definition of vacuity to achieve a new notion of *redundancy* in logical formulas. Redundancy captures several forms of antecedent failure that escape traditional vacuity analysis, including vacuous actions in temporal modalities and unnecessarily strong temporal operators. Furthermore, we have implemented an efficient redundancy checker for the modal mu-calculus in the context of the XMC model checker. Our checker generates *diagnostic information* in the form of all *maximal* subformulas that are redundant and exploits the fact that XMC can cache intermediate results in *memo tables* between model-checking runs. We have applied our redundancy checker to a number of previously published case studies, and found instances of redundancy that have gone unnoticed till now. These findings provide compelling evidence of the importance of redundancy detection in the design process.

1 Introduction

Model checking [8,20,9] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal-logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [10,16].

Most model checkers produce a *counter example* when the system under investigation does not satisfy a given temporal-logic formula. Such a counter example typically takes the form of an execution trace of the system leading to the violation of the formula. Until recently, however, standard practice was to move on to another formula when the model checker returned a result of true.

^{*} This work was supported in part by NSF grants EIA-9705998, CCR-9876242, CCR-9988155; ONR grant N000140110967; and ARO grants DAAD190110003, DAAD190110019.

Starting with [3] and continuing with [4,18,5], researchers have been working on the problem of “suspecting a positive answer.” In particular, this pioneering work shows how to detect situations of *vacuity*, where a model checker returns true but the formula is vacuously true in the model. Vacuity can indicate a serious underlying flaw in the formula itself or in the model. The most basic form of vacuity is *antecedent failure*, where a formula containing an implication proves true because the antecedent, or pre-condition, of the implication is never satisfied in the model. The work of [4,18,5] has shown that this notion extends in a natural way to various temporal logics. This work also considers the problem of generating “interesting witnesses” to valid formulas that are not vacuously true.

The point that we would like to reinforce in the present paper is that a vacuously true formula may represent a design problem as serious in nature as a model-checking result of false. In fact, we present compelling anecdotal evidence that *redundancy checking*—a more encompassing form of vacuity detection that we discuss below—is just as important as model checking, and a model-checking process that does not include redundancy checking is inherently incomplete and suspect.

Our investigation of redundancy checking has been conducted in the context of XMC [21,22], a model checker for systems described in XL, a highly expressive extension of value-passing CCS [19], and properties given as formulas of the modal mu-calculus. We have both implemented a redundancy checker for XMC that extends the approaches put forth [18,5] in several significant ways, and assessed its performance and utility on several substantive case studies. Our main contributions can be summarized as follows.

- We extend the vacuity-checking results of [18,5] to the modal mu-calculus [17], a very expressive temporal logic whose expressive power supersedes that of CTL* and related logics such as CTL and LTL (Section 2). In particular, [5] presents an effective procedure for vacuity checking for any logic with *polarity*. We prove that the modal mu-calculus is also a logic with polarity and therefore the results of [5] apply.
- We introduce a notion called *redundancy* that extends vacuity and identifies forms of antecedent failure that escape traditional vacuity analysis (Section 3). For example, consider the modal mu-calculus formula $[a, b]\phi$, meaning that in the given model, ϕ holds necessarily after every a - and b -transition. This formula is trivially true if the formula $[-]\phi$ also holds, meaning that regardless of the type of transition taken, necessarily ϕ .
- The algorithm of [5] pursues a bottom-up approach to vacuity checking in logics with polarity, replacing minimal subformulas by true or false. In contrast, our algorithm returns all *maximal* redundant subformulas. Such information can assist in isolating and correcting sources of redundancy in a model and its logical requirements. Our algorithm for redundancy checking is described in Section 4.
- XMC uses the XSB logic-programming system [24] as its underlying resolution engine, which, in turn implements tabled resolution. It is therefore

possible when using XMC to cache intermediate results in *memo tables* between model-checking runs. As pointed out in [5], this is likely to significantly improve the performance of a vacuity checker, as the process of vacuity detection for a logic with polarity requires a series of model-checking runs, all involving the same model and highly similar formulas. Our performance results bear out this conjecture (Section 5).

- We have applied our redundancy checker to a number of previously published case studies, and found instances of redundancy that have gone unnoticed till now (Section 5). These case studies include the Rether real-time ethernet protocol [14]; the Java meta-locking algorithm for ensuring mutually exclusive access by threads to object monitor queues [2]; and the safety-critical part of a railway signaling system [11]. These findings provide compelling evidence of the importance of redundancy detection in the design process.

2 Vacuity in the Modal Mu-Calculus

In [5], Beer et al. describe an efficient procedure for vacuity checking in logics with *polarity*. In this section, we first show that the modal mu-calculus is a logic with polarity, and hence the results of [5] are applicable. We then expand the notion of vacuity to one of *redundancy* to handle subtle forms of antecedent failure that are not considered vacuous. We begin by recalling the pertinent definitions and results from [5].

2.1 Vacuity and Logics with Polarity

We use φ , possibly with subscripts and primes, to denote formulas in a given logic, and ψ to denote subformulas of a formula. In the following, identical subformulas that occur at different positions in a formula are considered to be distinct. We use $\varphi[\psi \leftarrow \psi']$ to denote the formula obtained from φ by replacing subformula ψ with ψ' .

Consider a formula φ of the form $\psi_1 \Rightarrow \psi_2$ and a model M such that ψ_1 *does not hold* in M . Note that φ holds in M (due to antecedent failure) regardless of the whether ψ_2 holds or not. This form of “redundancy” is captured by the following definitions.

Definition 1 (Affect [5]) *A subformula ψ of formula φ affects φ in model M if there is a formula ψ' , such that the truth values of φ and $\varphi[\psi \leftarrow \psi']$ are different in M .*

Definition 2 (Vacuity [5]) *Formula φ is said to be ψ -vacuous in model M if there is a subformula ψ of φ such that ψ does not affect φ in M . Moreover, let S be a set of subformulas of φ . Formula φ is S -vacuous in model M if there exists $\psi \in S$ such that φ is ψ -vacuous in M .*

Although Definitions 1 and 2 capture a notion of vacuity independently of any particular logic, they cannot be directly used to *check* vacuity. Beer et

al. describe a procedure to check vacuity of formulas in logics with polarity, where the candidate ψ' chosen as a replacement in Definition 1 is drawn from $\{true, false\}$.

These ideas are formalized in [5] as follows. Let S be a set of subformulas of φ and $\min(S)$ be the subset of S of formulas that are minimal with respect to the (syntactic) subformula preorder. Also, let $\llbracket \varphi \rrbracket$ denote the set of all models in which formula φ is valid ($\llbracket \varphi \rrbracket = \{M \mid M \models \varphi\}$).

Definition 3 (Polarity of an operand; Logic with polarity [5]) *If σ is an n -ary operator in a logic, the i -th operand of σ has positive (negative) polarity if for every formula $\varphi_1, \dots, \varphi_{i-1}, \varphi_{i+1}, \dots, \varphi_n$, and two formulas ψ_1 and ψ_2 such that $\llbracket \psi_1 \rrbracket \subseteq \llbracket \psi_2 \rrbracket$ ($\llbracket \psi_2 \rrbracket \subseteq \llbracket \psi_1 \rrbracket$) we have that*

$$\llbracket \sigma(\varphi_1, \dots, \varphi_{i-1}, \psi_1, \varphi_{i+1}, \dots, \varphi_n) \rrbracket \subseteq \llbracket \sigma(\varphi_1, \dots, \varphi_{i-1}, \psi_2, \varphi_{i+1}, \dots, \varphi_n) \rrbracket$$

An operator has polarity if every one of its operands has some polarity (positive or negative). A logic with polarity is a logic in which every operator has polarity.

The notion of polarity is extended from operands to formulas as follows. The top-level formula φ is assumed to have positive polarity. Let ψ be some subformula of φ with \circ as its top-level operator. Let ψ_i be the i -th operand of \circ in ψ . Then ψ_i has positive polarity if the polarities of ψ and the i -th operand of \circ are identical (i.e. both positive or both negative); ψ_i has negative polarity otherwise. For instance, let $\varphi = \psi_1 \wedge \neg(\psi_2 \vee \neg\psi_3)$. Note that both operands of ‘ \wedge ’ and ‘ \vee ’ have positive polarity and the operand of ‘ \neg ’ has negative polarity. The subformulas of φ have the following polarities: φ, ψ_1 and ψ_3 have positive polarities; ψ_2 has negative polarity.

In this paper, a subformula’s polarity is often discussed together with the formula’s validity in a model. For the sake of convenience, we say that ψ is positive in $\langle M, \varphi \rangle$ if $M \models \varphi$ and ψ has positive polarity, or $M \not\models \varphi$ and ψ has negative polarity. Similarly, we say that ψ is negative in $\langle M, \varphi \rangle$ if $M \models \varphi$ and ψ has negative polarity, or $M \not\models \varphi$ and ψ has positive polarity.

Theorem 4 ([5]) *In a logic with polarity, for a formula φ and a set S of subformulas of φ , for every model M , the following are equivalent:*

1. φ is S -vacuous in M
2. There is a $\psi \in \min(S)$ such that:

$$M \models \varphi \iff M \models \varphi[\psi \leftarrow \perp_\psi]$$

where $\perp_\psi = false$ if ψ is positive in $\langle M, \varphi \rangle$; $\perp_\psi = true$ otherwise.

Theorem 4 directly yields a procedure to check vacuity of a formula with complexity $O(|\varphi| \cdot C_M(|\varphi|))$, where $C_M(n)$ denotes the complexity of model checking a formula of size n with respect to model M .

2.2 The Modal Mu-Calculus is a Logic with Polarity

In order to be able to apply the vacuity-detection procedure of [5] to mu-calculus formulas, we need to show that every operator in the mu-calculus has a polarity. Following [6], the syntax of modal mu-calculus formulas over a set of variable names Var , a set of atomic propositions $Prop$, and a set of labels \mathcal{L} is given by the following grammar, where $P \in Prop$, $Z \in Var$, and $a \in \mathcal{L}$:

$$\begin{aligned} \varphi \rightarrow & P \mid Z \mid \varphi \wedge \varphi \mid \neg\varphi \mid [a]\varphi \\ & \mid \nu Z.\varphi \text{ if every free occurrence of } Z \text{ in } \varphi \text{ is positive} \end{aligned}$$

Additional derived operators, such as $\langle \cdot \rangle$, \vee and μ (the duals of $[\cdot]$, \wedge and ν , respectively), are introduced for convenience. Moreover, the $\langle \cdot \rangle$ and $[\cdot]$ modalities may be specified with *sets* of action labels: $[S]\varphi$ stands for $\bigwedge_{a \in S} [a]\varphi$, and $[-S]\varphi$ stands for $[\mathcal{L} - S]\varphi$. The formula $[-\emptyset]\varphi$, i.e. $[\mathcal{L}]\varphi$, is shorthand as $[-]\varphi$.

A mu-calculus *structure* \mathcal{T} (over $Prop$, \mathcal{L}) is a labeled transition system with set \mathcal{S} of states and transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ (also written as $s \xrightarrow{a} t$), together with an interpretation $\mathcal{V}_{Prop} : Prop \rightarrow 2^{\mathcal{S}}$ for the atomic propositions.

Proposition 5 *The modal mu-calculus is a logic with polarity.*

Proof. The polarity of \wedge and \neg has been shown in [5]. It remains to show that operators $[a]$ and ν also have polarity.

Polarity of $[a]$. Given a mu-calculus structure \mathcal{T} and an interpretation $\mathcal{V} : Var \rightarrow 2^{\mathcal{S}}$ of the variables, the set of states satisfying $[a]\varphi$ is defined as:

$$\llbracket [a]\varphi \rrbracket_{\mathcal{V}}^{\mathcal{T}} = \{s \mid \forall t. s \xrightarrow{a} t \Rightarrow t \in \llbracket \varphi \rrbracket_{\mathcal{V}}^{\mathcal{T}}\}$$

For φ_1, φ_2 such that $\llbracket \varphi_1 \rrbracket \subseteq \llbracket \varphi_2 \rrbracket$, $\llbracket [a]\varphi_1 \rrbracket = \{s \mid \forall t. s \xrightarrow{a} t \Rightarrow t \in \llbracket \varphi_1 \rrbracket\}$. But this means that $t \in \llbracket \varphi_2 \rrbracket$ and, thus, $s \in \llbracket [a]\varphi_2 \rrbracket$. Therefore, $\llbracket [a]\varphi_1 \rrbracket \subseteq \llbracket [a]\varphi_2 \rrbracket$, and so, $[a]$ has positive polarity.

Polarity of ν . Similarly, the meaning of $\nu Z.\varphi$ is given as:

$$\llbracket \nu Z.\varphi \rrbracket_{\mathcal{V}}^{\mathcal{T}} = \bigcup \{S \subseteq \mathcal{S} \mid S \subseteq \llbracket \varphi \rrbracket_{\mathcal{V}[Z:=S]}^{\mathcal{T}}\}$$

where $\mathcal{V}[Z := S]$ maps Z to S and otherwise agrees with \mathcal{V} .

For φ_1, φ_2 such that $\llbracket \varphi_1 \rrbracket \subseteq \llbracket \varphi_2 \rrbracket$, $\llbracket \nu Z.\varphi_1 \rrbracket = \bigcup \{S \subseteq \mathcal{S} \mid S \subseteq \llbracket \varphi_1 \rrbracket_{\mathcal{V}[Z:=S]}\}$. This implies that $S \subseteq \llbracket \varphi_2 \rrbracket_{\mathcal{V}[Z:=S]}$ and therefore $\llbracket \nu Z.\varphi_1 \rrbracket \subseteq \llbracket \nu Z.\varphi_2 \rrbracket$, which demonstrates positive polarity of ν . \square

3 Redundancy

Proposition 5 permits us to directly apply the vacuity-detection procedure of [5] to mu-calculus formulas. However, in the case of the mu-calculus, the notion of vacuity does not capture subtle forms of antecedent failure. For example,

the action labels in modal operators can be a source of antecedent failure. In this section, we extend the notion of vacuity to take into account sources of antecedent failure that escape traditional vacuity analysis.

Consider the mu-calculus formula $\varphi_1 = \mu X.(p \vee \langle \{a, b\} \rangle X)$ which states that proposition p holds eventually along some path containing action labels a or b . Now consider the formula $\varphi_2 = \mu X.(p \vee \langle a \rangle X)$ which states that p holds eventually along some path containing only label a . The formula φ_2 is more specific than φ_1 : if φ_2 holds in a model M , so does φ_1 . Hence there are syntactic elements in φ_1 that are not *needed* for its validity in M .

The above example is actually a form of antecedent failure. To see this, notice that $\langle \{a, b\} \rangle X \equiv \langle a \rangle X \vee \langle b \rangle X$ which may be written as $\neg \langle a \rangle X \Rightarrow \langle b \rangle X$. Clearly, the antecedent in this implication fails whenever $\langle a \rangle X$ holds. Since syntactic elements such as action labels in the mu-calculus are not subformulas, they are not covered by the definition of vacuity given in [5]. Section 5 describes several previously published case studies that suffer from antecedent failure due to unnecessary actions in modal operators. Thus, this is a problem that occurs in practice and tends to obscure bugs in system models.

To capture these other forms of antecedent failure, we extend the notion of vacuity to one of *redundancy*. Vacuity detection can be seen as determining whether a formula can be strengthened or weakened without affecting its validity. For example, let $\varphi = \psi_1 \vee \psi_2$, and let M be a structure such that $M \models \varphi$. If $M \models \psi_1$, then φ can be replaced by ψ_1 . Since $\llbracket \psi_1 \rrbracket \subseteq \llbracket \varphi \rrbracket$, this replacement has in effect strengthened the formula. Vacuity checking, as defined in [5], does precisely this by checking for the validity of $\varphi[\psi_2 \leftarrow \text{false}]$, which is the same as ψ_1 . Similarly, the vacuity of a formula invalid in M is detected by testing the validity of a weaker formula. However, replacement of subformulas by *true* or *false* alone is insufficient in the case of mu-calculus. Let M be a structure such that $M \models \langle \{a, b\} \rangle \psi$, but $M \not\models \langle \{a, b\} \rangle \text{false}$. Following [5], $\langle \{a, b\} \rangle \psi$ is not ψ -vacuous in M . However, if $M \models \langle a \rangle \psi$, there is antecedent failure in the proof of $M \models \langle \{a, b\} \rangle \psi$ as explained in the previous paragraph. We can now show that the b in the formula $\langle \{a, b\} \rangle \psi$ is superfluous if we can strengthen it to $\langle a \rangle \psi$.

Checking for vacuity by strengthening (for valid formulas) or weakening (for invalid formulas) needs to be done judiciously. For instance, if φ holds in M , even if a stronger formula φ' holds in M , there may be no redundancy in φ . Consider a model M that satisfies the formula $\mu X. p \vee \langle - \rangle X$ in two steps, i.e., as $\langle - \rangle \langle - \rangle p$. Note that X can be written as $\neg p \Rightarrow \langle - \rangle (\neg p \Rightarrow \langle - \rangle (\neg p \Rightarrow \langle - \rangle X))$, and hence the antecedent of the innermost ' \Rightarrow ' fails in M . However, X is not trivially true in M , and hence should not be considered redundant. Hence we define a syntactic preorder over formulas, and restrict the choices of replacements during vacuity detection to only formulas related by this preorder.

The preorder \prec on which we base redundancy checking in the mu-calculus captures a notion of *syntactic simplification* and is defined as follows:

- $\text{false} \prec \varphi$, and $\text{true} \prec \varphi$ for any $\varphi \notin \{\text{true}, \text{false}\}$
- $[S]\varphi \prec [T]\varphi$ and $[-S]\varphi \prec [-T]\varphi$ for any φ if $S \subset T$.
- $\langle S \rangle \varphi \prec \langle T \rangle \varphi$ and $\langle -S \rangle \varphi \prec \langle -T \rangle \varphi$ for any φ if $S \subset T$.

Intuitively, $\varphi_1 \prec \varphi_2$ means that φ_1 is syntactically simpler than φ_2 . A formula φ is redundant with respect to a model M if it can be strengthened or weakened by replacing a subformula ψ with a simpler one without changing φ 's validity.

Definition 6 (Redundancy) *A formula φ is said to be redundant with respect to model M if there is a subformula ψ of φ such that*

$$M \models \varphi \iff \exists \psi' \prec \psi \text{ such that } M \models \varphi[\psi \leftarrow \psi']$$

where we also insist of ψ' that

- $\llbracket \psi' \rrbracket \subseteq \llbracket \psi \rrbracket$ if ψ is positive in $\langle M, \varphi \rangle$;
- $\llbracket \psi \rrbracket \subseteq \llbracket \psi' \rrbracket$ if ψ is negative in $\langle M, \varphi \rangle$.

We also say ψ is redundant in φ with respect to M .

Definition 6 specifies an effective procedure for finding redundancy. The test for $\llbracket \psi' \rrbracket \subseteq \llbracket \psi \rrbracket$ (or its dual) for formulas ψ, ψ' such that $\psi' \prec \psi$ is straightforward based on the following identities that hold for all φ :

- $\llbracket false \rrbracket \subseteq \llbracket \varphi \rrbracket$ and $\llbracket \varphi \rrbracket \subseteq \llbracket true \rrbracket$
- $\llbracket [T]\varphi \rrbracket \subseteq \llbracket [S]\varphi \rrbracket$ and $\llbracket [-S]\varphi \rrbracket \subseteq \llbracket [-T]\varphi \rrbracket$ if $S \subset T$
- $\llbracket \langle S \rangle \varphi \rrbracket \subseteq \llbracket \langle T \rangle \varphi \rrbracket$ and $\llbracket \langle -T \rangle \varphi \rrbracket \subseteq \llbracket \langle -S \rangle \varphi \rrbracket$ if $S \subset T$

Observe that the notion of redundancy extends vacuity, since the constants *true* and *false* are defined to be simpler than any formula, and hence are always candidate replacement formulas. Hence we have:

Proposition 7 *For all formulas φ in modal mu-calculus, if φ is vacuous then φ is redundant.*

The simplification preorder defined above can be further refined by introducing additional cases. For example, we can stipulate that:

- $[-]\varphi \prec [T]\varphi$ for any φ .
- $\langle - \rangle \varphi \prec \langle T \rangle \varphi$ for any φ .

For an illustration of the rationale behind these additions, consider the formulas $\varphi_1 = [a]\psi$ and $\varphi_2 = [-]\psi$. If $M \models \varphi_2$ then clearly also $M \models \varphi_1$, but the label a itself plays no role, and hence is redundant.

Redundancy of Temporal Operators For the mu-calculus, we do not need additional rules for simplifying fixed-point operators. For example, consider formula $\varphi = \mu X.p \vee \langle - \rangle X$ (*EFp* in CTL) and a model M where φ holds. Replacing $\langle - \rangle X$ by *false* we get $\mu X.p$, which is equivalent to p , as the strengthened formula. In terms of CTL, this is as if we check the vacuity of *EF* operator. The explicit nature of the fixed-point operators lets us check for their redundancy with no additional rules.

To check for redundancy in temporal logics without explicit fixed-point operators, such as CTL and CTL*, a simplification preorder over temporal operators must be specified. For example, in the case of CTL, the natural simplification preorder would stipulate that *false* \prec *AG* φ \prec *EG* φ \prec φ \prec *AF* φ \prec *EF* φ . Such a simplification preorder would therefore enable one to detect redundancies that arise from subformulas as well as temporal operators.

4 Algorithm for Redundancy Detection

Our algorithm for redundancy checking is based on Definition 6: a subformula ψ in φ is redundant if there is a “simpler” ψ' (i.e. ψ' is smaller than ψ in the simplification preorder \prec) such that $M \models \varphi \iff M \models \varphi[\psi \leftarrow \psi']$. Let a *modal formula* be a modal mu-calculus formula whose top-level operator is $[\cdot]$ or $\langle \cdot \rangle$. When ψ is not a modal formula, only *true* and *false* are simpler than ψ and hence the definition of redundancy collapses to the definition of vacuity in [5]. For a modal formula ψ , the number of simpler formulas ψ' is exponential in the number of action labels in the modal operator of ψ . However, not every simplification of ψ needs to be considered for checking the redundancy of ψ . Below we define a set of possible replacements for each modal formula that is sufficient for the redundancy check.

Definition 8 (Replacement Set) *Given a model M , formula φ , and a modal subformula ψ of φ , the replacement set of ψ with respect to φ and M , denoted by $\Sigma(\psi)$, is given by:*

- if ψ is positive in $\langle M, \varphi \rangle$

$$\begin{aligned}\Sigma(\langle T \rangle \phi) &= \{ \langle T - \{a\} \rangle \phi \mid a \in T \} \\ \Sigma(\langle -T \rangle \phi) &= \emptyset \\ \Sigma([T] \phi) &= \{ [-] \phi \} \\ \Sigma([-T] \phi) &= \{ [- (T - \{a\})] \phi \mid a \in T \}\end{aligned}$$

- if ψ is negative in $\langle M, \varphi \rangle$

$$\begin{aligned}\Sigma(\langle T \rangle \phi) &= \{ \langle - \rangle \phi \} \\ \Sigma(\langle -T \rangle \phi) &= \{ \langle - (T - \{a\}) \rangle \phi \mid a \in T \} \\ \Sigma([T] \phi) &= \{ [T - \{a\}] \phi \mid a \in T \} \\ \Sigma([-T] \phi) &= \emptyset\end{aligned}$$

Note that the replacement set for a modal formula ψ is linear in the number of action labels in ψ 's modal operators,

The rationale for the above definition of Σ is two-fold. First, it incorporates the semantic condition $\llbracket \psi' \rrbracket \subseteq \llbracket \psi \rrbracket$ (or its dual) into the syntactic check. Second, it picks only maximal candidates for replacement, ignoring those that are “covered” by others. For instance, let $\psi = \langle T \rangle \psi_1$ be a subformula of φ . Furthermore let ψ have positive polarity and $M \models \varphi$. Formulas of the form $\langle S \rangle \psi_1$ for some subset $S \subset T$ are simplifications of ψ . From the polarity of ψ , we have $M \not\models \varphi[\psi \leftarrow \langle S \rangle \psi_1] \Rightarrow \forall S' \subset S, M \not\models \varphi[\psi \leftarrow \langle S' \rangle \psi_1]$. Hence if some S is insufficient to show the redundancy of T then no $S' \subset S$ can show the redundancy of T . Therefore it is sufficient to consider the maximal sets S such that $S \subset T$ in the replacement formula $\langle S \rangle \psi_1$. The following proposition can be readily established considering the different cases in the definition of Σ .

Algorithm 1 Redundancy detection by subformula simplification

algorithm *find_redundancies*(M, φ)
NonVac keeps track of non-vacuous subformulas;
subs returns a formula's immediate subformulas.

- 1: $mck(M, \varphi)$
- 2: $NonVac = \{\varphi\}$
- 3: $Red = \emptyset$
- 4: **while** $NonVac \neq \emptyset$ **do**
- 5: remove a ψ from $NonVac$
- 6: **for all** $\chi \in subs(\psi)$ **do**
- 7: **if** $\psi[\chi \leftarrow \perp_\chi] \not\equiv \perp_\psi$ and $mck(M, \varphi[\chi \leftarrow \perp_\chi]) = mck(M, \varphi)$ **then**
- 8: add χ to Red
- 9: **else**
- 10: add χ to $NonVac$
- 11: **if** χ is a modal formula **then**
- 12: **for all** $\chi' \in \Sigma(\chi)$ **do**
- 13: **if** $mck(M, \varphi[\chi \leftarrow \chi']) = mck(M, \varphi)$ **then**
- 14: add χ to Red
- 15: **if** $Red = \emptyset$ **then**
- 16: report “ φ is not redundant in M ”
- 17: return Red

Proposition 9 *A modal subformula ψ is redundant in φ with respect to model M if there is some $\psi' \in \Sigma(\psi)$ such that $M \models \varphi \iff M \models \varphi[\psi \leftarrow \psi']$.*

We can further reduce the number of model-checking runs needed to determine redundancy by exploiting the relationship between vacuity of a formula and its immediate subformulas. Since $\varphi[\chi \leftarrow \perp_\chi] \equiv \varphi[\psi \leftarrow \psi[\chi \leftarrow \perp_\chi]]$ when χ is a subformula of ψ , applying Theorem 4 we have the following proposition.

Proposition 10 *If χ is a subformula of ψ and $\psi[\chi \leftarrow \perp_\chi] \equiv \perp_\psi$, then φ is not χ -vacuous in M iff φ is not ψ -vacuous in M .*

Since redundancy extends vacuity, if φ is ψ -vacuous in M , then all subformulas of ψ are redundant in φ .

Note that the proof of the if-direction of the above proposition follows from Lemma 3 of [5]. Whether $\psi[\chi \leftarrow \perp_\chi]$ is equal to \perp_ψ can be checked by syntactic transformation techniques such as partial evaluation, using rules such as $false \wedge X \equiv false$ and $\mu X.p \wedge \langle - \rangle X \equiv false$.

The pseudo-code for the redundancy-checking algorithm is presented in Algorithm 1. For a given model M and formula φ , the algorithm returns the set of maximal subformulas that are redundant in φ . Finding the *maximal* subformula of φ for which φ is redundant is accomplished by traversing φ 's syntax tree in a top-down manner. For each subformula ψ , we first check if φ is ψ -vacuous by model checking $\varphi[\psi \leftarrow \perp_\psi]$, as required in [5]. If φ is ψ -vacuous, we do not traverse the children of ψ since we know from Proposition 10 that they are

all vacuous. If φ is not ψ -vacuous and ψ contains a modal operator, we check whether it is redundant by model checking φ after replacing ψ by some formula from $\Sigma(\psi)$. Note that in the worst case we will visit all subformulas and hence use $O(|\varphi|)$ model-checking runs.

5 Case Studies and Performance Measurements

In this section, we describe how we applied the redundancy checker to case studies provided with the XMC and CWB-NC [12] distributions. In summary, we found nontrivial redundancies in modal mu-calculus formulas taken from XMC's Meta-lock and Rether examples and CWB-NC's railway example. We also used the Meta-lock example as a performance benchmark for our redundancy checker.

5.1 Case Study 1: Java Meta-lock (XMC)

Meta-locking is a highly optimized technique deployed by the Java Virtual Machine (JVM) to ensure that threads have mutually exclusive access to object monitor queues [1]. An abstract specification of the Meta-lock algorithm was verified in XMC [2] and has since been distributed as a part of the XMC system. The specification, whose top-level process is denoted by `metaj(M, N)` is parameterized with respect to the number of threads (M) and number of objects (N) that are accessed by these threads.

One of the properties verified was `liveness(I, J)`, which states that a thread I requesting a meta-lock to object J , will eventually obtain this meta-lock:

```
liveness(I,J) -= [requesting_metalock(I,J)] formula(I,J)
               /\ [-] liveness(I,J).
```

```
formula(I,J) += <got_metalock(I,J)> tt
               /\ form(I,J)
               /\ [-] formula(I,J).
```

```
form(I,J) += <got_metalock(I,J)> tt
            /\ [-{requesting_metalock(_,_)})] form(I,J).
```

Our checker detected redundancy in the subformula `[-] formula(1,1)` with respect to the transition system generated by `metaj(2,1)`. In particular, our checker found that subformulas `<got_metalock(1,1)> tt` of `formula(1,1)`, and `<got_metalock(1,1)> tt` of `form(1,1)` were redundant. In retrospect, the problem with the Meta-lock liveness formula is that it attempted to use the alternation-free fragment of the mu-calculus supported by XMC to incorporate a notion of strong fairness into the property. Unfortunately, alternating fixed points must be used to express strong fairness.

Table 1. Redundancy checker performance measurements

ψ	(2,2)			(2,3)			(3,1)			(3,2)		
	Truth	Time	Mem	Truth	Time	Mem	Truth	Time	Mem	Truth	Time	Mem
0	true	2.15	11.9	true	28.24	142.00	true	1.44	10.29	true	55.27	285.47
1	x	x	x	x	x	x	x	x	x	x	x	x
2	false	0.79	12.11	false	10.27	144.4	false	0.50	10.45	false	18.90	289.82
3	<u>true</u>	1.36	12.51	<u>true</u>	17.79	148.15	<u>true</u>	0.88	10.69	<u>true</u>	34.82	296.88
4	-	-	-	-	-	-	-	-	-	-	-	-
5	false	0.81	12.76	false	10.41	150.60	false	0.51	10.85	false	19.23	301.24
6	<u>true</u>	0.93	13.08	<u>true</u>	11.47	152.95	<u>true</u>	0.70	11.18	false	21.16	307.34
7	-	-	-	-	-	-	-	-	-	<u>true</u>	51.81	325.95
8	-	-	-	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	false	21.07	330.91
10	-	-	-	-	-	-	-	-	-	false	21.07	335.88
11	<u>true</u>	0.82	13.64	<u>true</u>	10.67	158.79	<u>true</u>	0.55	11.50	<u>true</u>	20.75	340.29
12	<u>true</u>	1.71	14.31	<u>true</u>	22.18	165.34	<u>true</u>	1.02	11.84	<u>true</u>	40.22	349.39
13	x	x	x	x	x	x	x	x	x	x	x	x
total		8.58	14.31		111.11	165.34		5.61	11.84		304.63	349.39

Performance of Redundancy Checking. Performance results for the redundancy checker on the liveness formula are given in Table 5.1 for various values of M and N. The formula’s syntax tree is depicted in Figure 1, and associates the DFS number of each node with the corresponding subformula. Each row of the table corresponds to a subformula ψ (indexed by its DFS number in the syntax tree) and contains the time (in seconds) and space (in MB) needed to check the redundancy of ψ ; i.e., model-check $\varphi[\psi \leftarrow \psi']$. Model checking is not needed for two kinds of subformulas: those that can be determined to be non-redundant by Proposition 10 (marked by an ‘x’), and those that can be determined to be redundant because they are subformulas of formulas that have already been determined redundant (marked by ‘-’).

From Table 5.1 we notice that model checking is faster when a subformula is not redundant (corresponding to a model-checking result of false), as opposed to when a formula is redundant (a model-checking result of true). This is due to the combination of the formula structure and the fact that XMC’s model checker is *local*; i.e., it operates in a goal-directed manner, examining only those portions of the state space needed to prove or disprove the formula. In particular, the liveness formula has universal modalities, causing the entire state space to be searched to establish its truth. In contrast, if it is false, this can be determined after visiting only a small fraction of the reachable state space.

Effect of Caching. Redundancy checking involves multiple runs of the model checker, each involving the same model and formulas similar up to subformula replacement. Recognizing this situation, we exploited the memo tables built by the underlying XSB resolution engine to share computations across model-

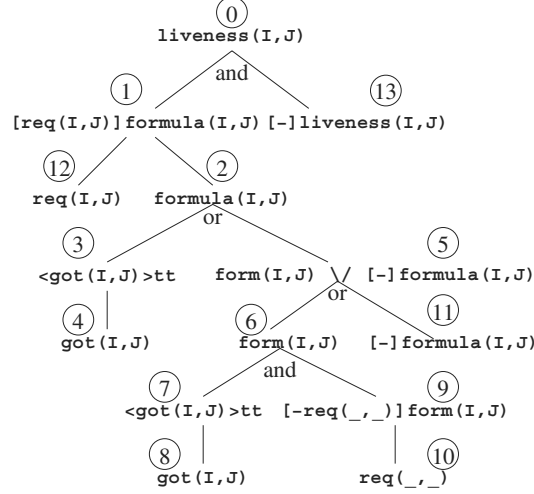


Fig. 1. Syntax tree of liveness property. Action labels `requesting_metalo`ck and `got_metalo`ck are abbreviated to `req` and `got` respectively

checking runs. Although retaining memo tables after each model-checking run resulted in the memory usage growing monotonically, the additional memory overhead is low (less than 3%) compared to the up to 25% savings in time.

5.2 Case Study 2: Rether (XMC)

Rether is a software-based, real-time ethernet protocol [7]. Its purpose is to provide guaranteed bandwidth for real-time (RT) data over commodity ethernet hardware, while also accommodating non-real-time (NRT) traffic. This protocol was first formally verified by Du et al. [13] using the Concurrency Factory.

Among the properties listed in [13], we found redundancies in a group of formulas that specify a “no starvation for NRT traffic” property. For each node i , a formula NS_i is defined as

$$NS_i = \nu X.([\neg]X \wedge [start]\nu Y.([\neg\{nrt_i, cycle\}]Y \wedge [cycle]\mu Z.([\neg nrt_i]Z \wedge \langle - \rangle tt))$$

which means that if during a particular cycle (between actions `start` and `cycle`) node i does not transmit any NRT data (signaled by action nrt_i), then eventually node i will transmit some NRT data after the cycle. Our checker shows the actions in the box modalities $[start]$, $[\neg\{nrt_i, cycle\}]$, and $[cycle]$ are redundant. Indeed, the simpler formula

$$NS'_i = \nu X.([\neg]X \wedge \mu Z.([\neg nrt_i]Z \wedge \langle - \rangle tt))$$

which means always eventually nrt_i will occur, holds for the protocol. Hence the cycle-based argument is unnecessarily complicated.

Redundancy checking also helped us find a bug in the model of Rether as encoded in the XMC system. At first we found another formula was also redundant:

$$RT_0 = \nu X.([\neg]X \wedge [reserve_0]\mu Y.([\neg cycle]Y \wedge [rt_0]ff \wedge [cycle]\mu Z.(\langle \neg \rangle tt \wedge [\neg rt_0]Z \wedge [cycle]ff)))$$

Redundancy was detected in the μY subformula by showing that $\nu X.([\neg]X \wedge [reserve_0]ff)$ is true. That implies action $reserve_0$ is never enabled, i.e. node 0 never reserves a real-time slot. Examining the model, we found a bug that causes node 0, which initially holds a real-time slot, to never release it, and hence has no need to reserve one! RT_0 became non-redundant after we fixed the bug.

5.3 Case Study 3: Railway Interlocking System (CWB-NC)

The CWB-NC’s railway example models a slow-scan system over a low-grade link in British Rail’s Solid State Interlocking system [11]. We found several instances of redundancy in this case study. The properties in question use the following four actions: $\overline{\text{fail_wire}}$ and $\overline{\text{fail_overflow}}$ report link failure due to a broken wire or due to a buffer overflow in a channel; $\overline{\text{det}}$ signals the detection of a failure; $\overline{\text{recovered}}$ marks the system’s reinitialization to the correct state.

The first property states that “a failure is possible”:

$$\begin{aligned} failures_possible = \mu X. & \langle \overline{\text{fail_overflow}}, \overline{\text{fail_wire}} \rangle tt \\ & \vee \langle \overline{\text{recovered}} \rangle X \end{aligned}$$

Since $\langle \overline{\text{fail_overflow}}, \overline{\text{fail_wire}} \rangle tt = \langle \overline{\text{fail_overflow}} \rangle tt \vee \langle \overline{\text{fail_wire}} \rangle tt$, $failures_possible$ can be converted to the disjunction of two stronger formulas: $fail_overflow_possible = \mu X. \langle \overline{\text{fail_overflow}} \rangle tt \vee \langle \overline{\text{recovered}} \rangle X$ and $fail_wire_possible = \mu X. \langle \overline{\text{fail_wire}} \rangle tt \vee \langle \overline{\text{recovered}} \rangle X$, which both turn out to be true. Therefore either $\overline{\text{fail_overflow}}$ or $\overline{\text{fail_wire}}$ (but not their simultaneous occurrence) does not contribute to the truth of $failures_possible$. This also holds for the derived formula

$$failure_possible_again = \nu Y. failure_possible \wedge [\neg]Y$$

The second property states that “a failure can be detected only after an occurrence of a failure”:

$$\begin{aligned} no_false_alarms = \nu X. & (\langle \overline{\text{det}} \rangle ff \vee \langle \overline{\text{fail_overflow}} \rangle tt) \\ & \wedge [\neg \overline{\text{fail_overflow}}, \overline{\text{fail_wire}}, \overline{\text{recovered}}] X \end{aligned}$$

which means that along a path without failure and recovery, either it is impossible to detect a failure or there is a failure. Removing $\overline{\text{recovered}}$ from the formula means that even if recovery occurs, the property still holds. Furthermore, the formula is redundant in $\langle \overline{\text{fail_overflow}} \rangle tt: \nu X. \langle \overline{\text{det}} \rangle ff \wedge [\neg \overline{\text{fail_overflow}},$

$\overline{\text{fail_wire}}]X$ is also true. Two types of redundancies also exist in the derived formula:

$$\text{no_false_alarms_again} = \nu Y. \text{no_false_alarms} \wedge [-]Y$$

These case studies illustrate that subtle redundancies can creep into complicated temporal properties. In some cases, the redundancies hide problems in the underlying system model. Redundancy detection provides significant guidance in deriving simpler and stronger temporal formulas for the intended properties.

6 Conclusions

In this paper, we have extended the notion of vacuity to the more encompassing notion of *redundancy*. Redundancy detection, unlike vacuity detection, can handle sources of vacuity that are not adequately treated by subformula replacement alone. The main idea behind redundancy checking is that a formula φ is redundant with respect to a model M if it has a subformula ψ that can be “simplified” without changing its validity. For this purpose, we have defined a simplification preorder for the modal mu-calculus that, among other things, addresses vacuity in the action structure of modal operators.

A simplification preorder for CTL that enables detection of unnecessarily strong temporal operators can also be defined. A simplification-based notion of redundancy addresses an open problem posed in [5], which was prompted by a question from the audience by Amir Pnueli at CAV ’97.

We have also implemented an efficient redundancy checker for the XMC verification tool set. The checker exploits the fact that XMC can be directed to cache intermediate model-checking results in memo tables, addressing another open problem in [5]. We have revisited several existing case studies with our redundancy checker and uncovered a number of instances of redundancy that have gone undetected till now. Like XMC, our redundancy checker is written declaratively in 300 lines of XSB tabled Prolog code.

For future work we are interested in pursuing the problem of generating interesting witness to valid formulas that are not vacuously true, à la [4,18,5]. The approach we plan to take to this problem will be based on traversing the proof tree generated by XMC’s proof justifier [23,15].

References

1. O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of OOPSLA ’99*, 1999. 156
2. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Scotland, April 2000. 149, 156

3. D. Beatty and R. Bryant. Formally verifying a multiprocessor using a simulation methodology. In *Design Automation Conference '94*, pages 596–602, 1994. 148
4. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV '97*, pages 279–290. LNCS 1254, Springer-Verlag, 1997. 148, 160
5. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, March 2001. 148, 149, 150, 151, 152, 154, 155, 160
6. J. Bradfield and C. Stirling. Modal logics and mu-calculi: An introduction. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001. 151
7. T. Chiueh and C. Venkatramani. The design, implementation and evaluation of a software-based real-time ethernet protocol. In *Proceedings of ACM SIGCOMM '95*, pages 27–37, 1995. 158
8. 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. 147
9. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986. 147
10. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996. 147
11. R. Cleaveland, G. Luetzgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17:50–62, 1996. 149, 159
12. R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag. 156
13. X. Du, K. T. McDonnell, E. Nanos, Y. S. Ramakrishna, and S. A. Smolka. Software design, specification, and verification: Lessons learned from the Rether case study. In *Proceedings of the Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Sydney, Australia, December 1997. Springer-Verlag. 158
14. X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. *Software Tools for Technology Transfer*, 2(3):219–241, November 1999. 149
15. H.-F. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *Proc. of 17th International Conference on Logic Programming (ICLP'01)*, November 2001. 160
16. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 147
17. D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, 1983. 148
18. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *CHARME 99*. LNCS 1703, Springer-Verlag, 1999. 148, 160
19. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. 148

20. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag. 147
21. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, LNCS 1254, Springer-Verlag, 1997. 148
22. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, et al. XMC: A logic-programming-based verification toolset. In *Proceedings of the 12th International Conference on Computer Aided Verification CAV 2000*. Springer-Verlag, June 2000. 148
23. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Proc. of Second International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, September 2000. 160
24. XSB. The XSB logic programming system v2.4, 2001. Available by anonymous ftp from <ftp.cs.sunysb.edu>. 148

On Solving Temporal Logic Queries

Samuel Hornus* and Philippe Schnoebelen

Lab. Spécification & Vérification, ENS de Cachan & CNRS UMR 8643
61, av. Pdt. Wilson, 94235 Cachan Cedex France
`phs@lsv.ens-cachan.fr`

Abstract. Temporal query checking is an extension of temporal model checking where one asks what propositional formulae can be inserted in a temporal query (a temporal formula with a placeholder) so that the resulting formula is satisfied in the model at hand.

We study the problem of computing all minimal solutions to a temporal query without restricting to so-called “valid” queries (queries guaranteed to have a unique minimal solution). While this problem is intractable in general, we show that deciding uniqueness of the minimal solution (and computing it) can be done in polynomial-time.

1 Introduction

Temporal model checking. Pnueli pioneered the use of *temporal logic* as a formal language for reasoning about reactive systems [Pnu77]. Temporal logic allows *model checking*, i.e. the automatic verification that a finite-state model of the system satisfies its temporal logic specifications [CGP99, BBF⁺01]. One limitation of model checking is that it gives simple “yes-or-no” answers: either the system under study satisfies the temporal formula, or it does not ¹.

Temporal queries. In a recent paper [Cha00], Chan proposed *temporal logic queries* as an extension of model checking (inspired by database queries). A temporal query is a temporal formula $\gamma(?)$ where the special symbol $?$ occurs as a placeholder (a query can have at most one $?$). A propositional formula f is a *solution* to the query if the system satisfies $\gamma(f)$ (γ with f in place of $?$), and *temporal query evaluation* is the process of computing the solutions to a temporal query.

When applicable, query answering extends model checking and is a more powerful and versatile tool for validating, debugging, and more generally understanding the formal model of a system [BG01].

* Now at Lab. GRAVIR, INRIA Rhône-Alpes, Montbonnot, France. Email: `Samuel.Hornus@inria.fr`. The research described in this paper was conducted while S. Hornus was at LSV.

¹ When the system does not satisfy the temporal formula, most model checkers provide a *diagnostic*, e.g. as a trace showing one possible way of violating the property.

Let us illustrate this with an example. Assume we are currently designing some system and rely on model checking to verify that it satisfies a typical temporal specification such as

$$G(\mathbf{request} \Rightarrow F \mathbf{grant}) \tag{S}$$

stating that “all requests are eventually granted”. Model checking will provide a yes-or-no answer: either the system satisfies (S) or it does not.

Temporal queries lead to a finer analysis of the system. The query

$$G(? \Rightarrow F \mathbf{grant}) \tag{Q}$$

asks for the conditions that always inevitably lead to **grant**. Computing the solutions to (Q) for our system will tell us, among other things, whether the system satisfies its specification: (S) is satisfied iff **request** is a solution to (Q). But it will tell us more. For example, if (S) is not satisfied, answering (Q) can lead to the discovery that, in reality, the property that our system satisfies is $G((\mathbf{request} \wedge \mathbf{free}) \Rightarrow F \mathbf{grant})$. Dually, if (S) is satisfied, query answering can lead to the discovery that, say, \top is a solution to (Q), i.e. $GF \mathbf{grant}$ is satisfied (grants need no request, likely to be a severe bug in our design).

We refer to [Cha00, BG01] for more motivations and examples on the role temporal queries can play in the modeling and validation of systems.

Minimal solutions. In general, evaluating a query γ on a system S will lead to many solutions. However, these solutions can be organized because of a fundamental monotonicity property: Say that γ is a *positive* (resp. a *negative*) query if the placeholder $?$ occurs under an even (resp. odd) number of negations in γ . Assume f is a solution to a positive γ and f' is a logical consequence of f , then f' too is a solution. (For negative queries, the implication goes the other way around.) Therefore, the set of solutions to a positive γ can be represented by its minimal elements, the *minimal solutions* (called *maximally strong* solutions in [BG01]). From a practical viewpoint, these solutions are the most informative [Cha00, BG01]. In our earlier example, (Q) is a negative query, and the minimal solutions are the weakest conditions that always ensure an eventual **grant**.

Unique minimal solutions. Chan proved monotonicity for CTL queries and introduced a notion of so-called *valid* queries, i.e. queries that always have a *unique minimal solution* for every system. He further defined CTL^v , a special subset that only contains valid CTL queries, and proposed a special-purpose algorithm for answering CTL^v queries.

While uniqueness of the minimal solution is a desirable feature (it certainly provides more intelligible answers), the fragment CTL^v is quite restricted. To begin with, CTL^v does not contain all valid queries. But the very notion of valid queries is a limitation by itself: for example, queries as simple as $EF?$ and $AF?$ are not valid. This limitation comes from the fact that valid queries have a unique minimal solution when evaluated over any system, while we are more interested

in queries that have a unique minimal solution *when evaluated over the system S at hand*. And when a query does not have a unique minimal solution over some S , we might be interested in knowing what are these several minimal solutions.

This prompted Bruns and Godefroid to study how one can compute, in a systematic way, the set of all minimal solutions to a query [BG01]. They showed how the automata-theoretic approach to model checking can be adapted to provide a *generic* algorithm computing all minimal solutions (generic in the sense that it works for all temporal logics). From a computational complexity viewpoint, their solution is in principle as costly as trying all potential solutions.

Our contribution. In this paper we study the problem of computing the set of minimal solutions to arbitrary temporal queries and look for feasible solutions. We show this problem can be computationally expensive: a query can have a huge number of minimal solutions and simply counting them is provably hard.

This does not mean all problems related to temporal query evaluation are intractable. Indeed, our main contribution is to show that deciding whether a given query has a unique minimal solution over a given system, and computing this solution, can be solved efficiently. Here “efficiently” means that we can reduce this to a linear number of model checking problems.

These methods are useful in more general situations. When there is not a unique minimal solution, we can compute a first minimal solution with a linear number of model checking calls, then compute a second minimal solution with a quadratic number of calls, then a third with a cubic number, etc.: every additional solution costs more. (We provide hardness results showing that some form of increasing costs is inescapable.)

We see these result as an *a posteriori* explanation of why it is desirable that a query only has a few minimal solutions: not only this makes the answers easier to understand and more informative, but *it also makes it easier to compute them*. Finally, we believe our approach has some advantages over Chan’s since it is not restricted to CTL^v (or more generally to valid queries) and it extends nicely to situations where a query only has a small number of minimal solutions.

Plan of the paper. We recall the formal definitions of temporal queries, their solutions, etc. in Section 2. Then we study the structure of the set of solutions in Section 3 and prove a few key results that provide the basis for the polynomial-time algorithms we exhibit in Section 4. Hardness results are given in Section 5.

2 Temporal Logic Queries

We quickly recall the syntax and semantics of CTL, LTL, and CTL^* , three of the main temporal logics used in model checking (we refer the reader to [Eme90, CGP99, BBF⁺01] for more background). We assume the reader is familiar with the main complexity classes (see e.g. [Pap94]).

2.1 Syntax of Temporal Logics

Assume $\mathcal{P} = \{P_1, P_2, \dots\}$ is a countable set of *atomic propositions*, CTL* formulae are given by the following syntax:

$$\varphi, \psi ::= \neg\varphi \mid \varphi \wedge \psi \mid E\varphi \mid X\varphi \mid \varphi U\psi \mid P_1 \mid P_2 \mid \dots$$

where E is the *existential path quantifier*, X is the *next-time* temporal modality, and U is the *until* temporal modality. Standard abbreviations are \perp (for $P_1 \wedge \neg P_1$), \top (for $\neg\perp$), $A\varphi$ (for $\neg E\neg\varphi$), $F\varphi$ (for $\top U\varphi$), $G\varphi$ (for $\neg F\neg\varphi$), as well as $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, \dots

LTL is the fragment of CTL* where the path quantifiers E and A are forbidden.

CTL is the fragment of CTL* where every modality U or X must appear immediately under the scope of a path quantifier E or A. CTL formulae are *state formulae*, that is, whether $\pi \models \varphi$ only depends on the current state $\pi(0)$. When φ is a state formula, we often write $q \models \varphi$ and say that φ holds in state q of S .

Boolean combinations of atomic propositions (i.e. no temporal combinator is allowed) are a special case of vacuously temporal formulae, called *propositional formulae*, and ranged over by f, g, \dots . Assuming the set of *atomic propositions* is $\mathcal{P} = \{P_1, P_2, \dots\}$, the propositional formulae are given by the abstract syntax

$$f, g ::= f \wedge g \mid \neg f \mid P_1 \mid P_2 \mid \dots$$

2.2 Semantics

A *Kripke structure* (shortly, a KS) is a tuple $S = \langle Q, q_{\text{init}}, \rightarrow, l \rangle$ where $Q = \{q, q', \dots\}$ is a finite set of *states*, $q_{\text{init}} \in Q$ is the *initial state*, $\rightarrow \subseteq Q \times Q$ is a total *transition relation* between states, and $l : Q \mapsto 2^{\mathcal{P}}$ labels the states with (finitely many) propositions. In the following we assume a given KS S .

A run π of S is an infinite sequence $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots$ of states linked by transitions (i.e. $(q_i, q_{i+1}) \in \rightarrow$ for all i). For π of the form $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots$, we write $\pi(i)$ for the i -th state q_i , and π^i for the i -th suffix $q_i \rightarrow q_{i+1} \rightarrow q_{i+2} \rightarrow \dots$ of π . Observe that π^i is a run. We write $\Pi(q)$ for the set of runs that start from q : since we assume \rightarrow is total, $\Pi(q)$ is never empty.

CTL* formulae describe properties of runs in a KS. We write $\pi \models \varphi$ when π satisfies φ in S . The definition is by induction on the structure of the formula:

$$\begin{aligned} \pi \models P_i & \stackrel{\text{def}}{\Leftrightarrow} P_i \in l(\pi(0)), \\ \pi \models \neg\varphi & \stackrel{\text{def}}{\Leftrightarrow} \pi \not\models \varphi, \\ \pi \models \varphi \wedge \psi & \stackrel{\text{def}}{\Leftrightarrow} \pi \models \varphi \text{ and } \pi \models \psi, \\ \pi \models E\varphi & \stackrel{\text{def}}{\Leftrightarrow} \text{there is a } \pi' \in \Pi(\pi(0)) \text{ s.t. } \pi' \models \varphi, \\ \pi \models X\varphi & \stackrel{\text{def}}{\Leftrightarrow} \pi^1 \models \varphi, \\ \pi \models \varphi U\psi & \stackrel{\text{def}}{\Leftrightarrow} \text{there is some } i \text{ s.t. } \pi^i \models \psi \text{ and } \pi^j \models \varphi \text{ for all } 0 \leq j < i. \end{aligned}$$

We write $S \models \varphi$ when $\pi \models \varphi$ for all $\pi \in \Pi(q_{\text{init}})$ and say that the KS S satisfies φ .

Semantical equivalences between temporal formulae are denoted $\varphi \equiv \psi$, while entailments are written $\varphi \supset \psi$.

2.3 Temporal Queries

Let TL be some temporal logic (e.g. CTL, LTL, CTL* or some other fragment of CTL*). A TL *query* γ is a TL formula in which the special placeholder symbol $?$ may appear in place where a propositional formula is allowed. Note that $?$ may appear at most once in γ . We say a query is *positive* (resp. *negative*) if $?$ appears under an even (resp. odd) number of negations. E.g. **(Q)** above is negative since “ $? \Rightarrow \text{grant}$ ” really is an abbreviation for $\neg? \vee \text{grant}$.

If f is a propositional formula, we write $\gamma(f)$ for the TL-formula obtained by replacing $?$ by f in γ . A *solution* of a query γ in a given KS S is a propositional formula f such that $S \models \gamma(f)$.

Lemmas 2.1 and 2.2 below state fundamental properties of the sets of solutions. Chan stated and proved them for CTL queries [Cha00] but they hold more generally.

Lemma 2.1 (Monotonicity). *Let γ be a CTL* query and f and g two propositional formulae:*

- *if γ is positive then $f \supset g$ entails $\gamma(f) \supset \gamma(g)$,*
- *if γ is negative then $f \supset g$ entails $\gamma(g) \supset \gamma(f)$.*

There is a duality principle linking positive and negative queries: let γ be a negative query. Then $\neg\gamma$ is a positive query and it is possible to compute the solutions of γ from the solutions of $\neg\gamma$: these are all f that are not solutions of $\neg\gamma$. This duality justifies the policy we now adopt: *from now on, we consider positive queries only.*

Lemma 2.2. *Let S be a Kripke structure and γ a CTL* query, then*

- *$S \models \gamma(\perp)$ iff $S \models \gamma(f)$ for all propositional formulae f ,*
- *$S \models \gamma(\top)$ iff $S \models \gamma(f)$ for some propositional formula f .*

Lemma 2.2 shows that deciding whether a query admits a solution in a given KS can be reduced to a model checking question: does $S \models \gamma(\top)$?

2.4 Minimal Solutions

A conclusion we draw from Lemmas 2.1 and 2.2 is that not all solutions to a temporal query problem are equally informative. The most informative solutions are the minimal ones:

Definition 2.3. *A minimal solution to a query γ in some KS S is a solution f that is not entailed by any other solution.*

Since the set of solutions to γ in S is closed w.r.t. entailment (also, *upward-closed*), the minimal solutions characterize the set of solutions.

The problem we are concerned with is, given a Kripke structure S and a temporal query γ , compute the set of minimal solutions to γ in S .

2.5 Relevant Solutions

There exists an important generalization of temporal query answering. Here one considers a given subset $\mathcal{RP} \subseteq \mathcal{P}$ of so-called *relevant* atomic propositions and only looks for *relevant solutions*, i.e. solutions that only use propositions from \mathcal{RP} . Note that, by Lemma 2.2, a query has relevant solutions iff it has solutions.

A *minimal relevant solution* is minimal among relevant solutions only. This is not the same as being minimal and relevant: a query γ may have a unique minimal solution and several minimal relevant solutions, or, dually, several minimal solutions and a unique minimal relevant solution.

Being able to look for relevant solutions greatly improves the scope of temporal query checking. Going back to our example in the introduction, it is very likely that a minimal solution to (Q) involves a lot of specific details about the system at hand. We will probably only get the informative solution `request` \wedge `free` if we restrict to a set of a few important propositions.

In the rest of this paper, we only look at the basic query answering problem. As the reader will see, all the results and algorithms we propose generalize directly to relevant solutions: it suffices to restrict the set of valuations one considers to valuations of \mathcal{RP} . While the generalization is crucial in practical applications, it is trivial and uninteresting in the more abstract analysis we develop.

3 The Lattice of Solutions

Assume we are given a fixed KS $S = \langle Q, q_{\text{init}}, \rightarrow, l \rangle$ where the states are labeled with propositions from a finite set $\mathcal{P} = \{P_1, \dots, P_m\}$ of m atomic propositions. Assuming finiteness of \mathcal{P} is not a limitation since anyway only the propositions appearing in S are useful.

A *valuation* is a mapping $v : \mathcal{P} \mapsto \{\perp, \top\}$ assigning a truth value to each atomic proposition. We write \mathcal{V} for the set of valuations on \mathcal{P} . Observe that each state q of S can be seen as carrying a valuation denoted v_q and given by $v_q(P_i) = \top \stackrel{\text{def}}{\iff} P_i \in l(q)$.

In the standard way, a propositional formula f denotes a set of valuations $\llbracket f \rrbracket \subseteq \mathcal{V}$, the valuations that satisfy f . We write $|f|_{\#}$ for the size of $\llbracket f \rrbracket$, i.e. the number of valuations that satisfy f .

3.1 The Lattice of Propositional Formulae

It is well known that the Boolean lattice $\langle 2^{\mathcal{V}}, \subseteq \rangle$ of subsets of \mathcal{V} is isomorphic to the lattice of propositional formulae ordered by entailment (when we do not distinguish between equivalent propositional formulae). We write \mathcal{L}_m for this lattice when \mathcal{P} contains m atomic propositions.

Fig. 1 displays \mathcal{L}_2 , the lattice of propositional formulae for $\mathcal{P} = \{P_1, P_2\}$. We use the exclusive-or operator, denoted \oplus , to shorten the notations of some formulae. In Fig. 1, the bold nodes delineate the sub-lattice of all propositional formulae entailed by $P_1 \wedge \neg P_2$.

There are 16 non-equivalent formulae in \mathcal{L}_2 . The huge size of \mathcal{L}_m can be explained with a few numbers: m atomic propositions allow 2^m valuations, so that there are 2^{2^m} non-equivalent propositional formulae.

\mathcal{L}_m can be sliced in $2^m + 1$ levels in the natural way, where a level collects all formulae f with same $|f|_{\#}$. The central level is the largest, containing

$$\Gamma_m \stackrel{\text{def}}{=} \binom{2^m}{2^{m-1}}$$

distinct formulae. Therefore a temporal query cannot have more than Γ_m minimal solutions (where m is the number of relevant propositions).

3.2 Bounding the Number of Minimal Solutions

We start by showing that the Γ_m upper bound for the number of minimal solutions is not very tight. Note that Γ_m is $2^{2^{\Omega(m)}}$.

In practice, rather than considering the number m of propositions, a better parameter is the number of different valuations that appear in the states of the Kripke structure. This is explained by the following lemma, where our notation “ $f \vee v$ ” treats valuation v as a propositional formula with the obvious meaning:

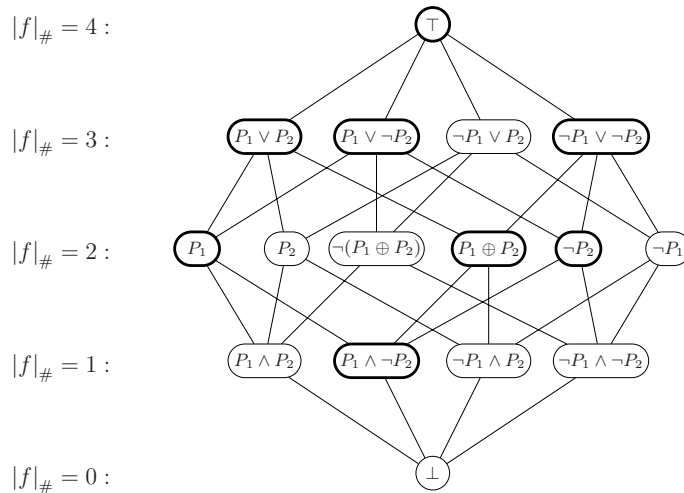


Fig. 1. \mathcal{L}_2 , the lattice of propositional formulae for $\mathcal{P} = \{P_1, P_2\}$

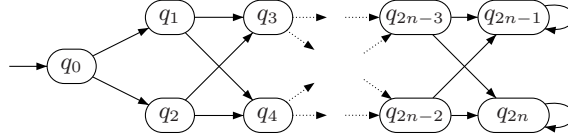


Fig. 2. S_n , a KS with 2^n minimal solutions to the query $\text{EG}?$

Lemma 3.1. *Let f be a propositional formula and v a valuation. If v does not label any state of S , then $S \models \gamma(f \vee v)$ iff $S \models \gamma(f)$.*

Proof. Obviously, for any state $q \in S$, $q \models f \vee v$ iff $q \models f$. This equivalence carries over to the whole of γ by structural induction. \square

Proposition 3.2. *Let γ be a CTL* query, and S a KS with n nodes. Then γ has at most $\binom{n}{\lfloor n/2 \rfloor}$ minimal solutions, i.e. less than 2^n . Furthermore, $|f|_{\#} \leq n$ for any minimal solution f .*

Proof. By Lemma 3.1, a minimal solution is a disjunction of the form $\bigvee_{q \in Q'} v_q$ for some subset $Q' \subseteq Q$ of states of S . Thus a minimal solution can account for at most n valuations. Also, there are 2^n subsets of Q , but if we want a large number of distinct Q' subsets s.t. none contains another one, then at most $\binom{n}{\lfloor n/2 \rfloor}$ subsets can be picked. \square

Now there does exist simple situations where a query has an exponential number of minimal solutions.

Consider the KS S_n from Fig. 2: it has $2n + 1$ states. If these $2n + 1$ states are labeled with different valuations, then we have:

Proposition 3.3. *In S_n , the query $\gamma = \text{EG}?$ has 2^n minimal solutions.*

Proof (Idea). Write v_0, v_1, \dots, v_{2n} for the valuations labeling q_0, q_1, \dots, q_{2n} . The minimal solutions are all f of the form $v_0 \vee \bigvee_{i=1}^n v_{k_i}$ where $k_i \in \{2i - 1, 2i\}$. \square

This example partly explains why it is difficult to compute the set of minimal solutions to an arbitrary query on an arbitrary KS: simply listing these minimal solutions takes exponential time. We show below that simply counting the minimal solutions is intractable (see Theorem 5.1).

4 Polynomial-Time Algorithms for Temporal Queries

In this section we exhibit two temporal-query problems that can be reduced efficiently to corresponding model checking problems.

Formally, these problems can be solved by a polynomial-time algorithm that uses model checking (for the underlying temporal logic) as an oracle. Using an oracle abstracts away from the specific details of the model checking algorithms for different temporal logics (and their different complexity).

Informally, we say that these problems can be solved “in P^{MC} ”. In practice, this means that they can be solved by simple extensions of standard model checking algorithms with little extra cost. In particular, *for CTL queries, these problems can be solved in polynomial-time*. More generally, this further holds of any temporal logic for which model checking is polynomial-time, like the alternation-free mu-calculus [CS92], or some other extensions of CTL like [KG96, LST00, LMS02, PBD⁺02]. Obviously, since model checking is a special case of temporal query solving, we cannot expect to do much better and provide efficient query solving for temporal logics where model checking is intractable.

Remark 4.1. We do not provide a fine-grained evaluation of our algorithm and say they are in polynomial-time when it is sometimes obvious that, e.g., only a linear number of oracle calls is used.

4.1 Deciding Minimality

Minimality of solutions can be reduced to model checking. This requires that the candidate solutions be given in a manageable form:

Theorem 4.2. *The problem of deciding, given some KS $S = \langle Q, q_{\text{init}}, \rightarrow, l \rangle$, query γ and propositional formula f , whether f is a minimal solution to γ in S , is in P^{MC} when f is given in disjunctive normal form, or as an OBDD².*

Proof. One uses the following algorithm:

- (1) First check that $S \models \gamma(f)$, otherwise f is not a solution.
- (2) Then check whether $|f|_{\#} > |Q|$. If so, then f is not a minimal solution (by Prop. 3.2).
- (3) Otherwise, enumerate $\llbracket f \rrbracket$ as some $\{v_1, \dots, v_k\}$. f is minimal iff $S \not\models \gamma(f \wedge \neg v_i)$ for $i = 1, \dots, k$.

Therefore minimality can be decided with at most $1 + |Q|$ invocations to a model checking algorithm. \square

Here, the assumption that f is in disjunctive normal form, or an OBDD, permits efficient enumeration of $\llbracket f \rrbracket$ and decision of whether $|f|_{\#} > |Q|$ as we now explain.

Computing $|f|_{\#}$ when f is an OBDD uses simple dynamic programming techniques (see, e.g., [Bry92, § 5.4]). When f is a disjunctive normal form, computing $|f|_{\#}$ is a bit more difficult. However, we just need to check whether $|f|_{\#} > |Q|$, which can be done by enumerating $\llbracket f \rrbracket$ assuming we stop as soon as we see that $|f|_{\#}$ is too large.

Remark 4.3. If f can be an arbitrary propositional formula, then minimality of f is NP-hard since satisfiability can easily be reduced to minimality.

² I.e. *Ordered Binary Decision Diagrams* [Bry92]. They now are a standard way of efficiently storing and handling boolean formulae in model checking tools.

4.2 Computing Unique Minimal Solutions

Uniqueness of the minimal solution can be reduced to model checking. Furthermore, when it is unique, the minimal solution can be computed efficiently.

Theorem 4.4. *The problem of deciding, given some KS $S = \langle Q, q_{\text{init}}, \rightarrow, l \rangle$ and query γ , whether γ admits a unique minimal solution in S (and computing that solution) is in P^{MC} .*

Proof. One uses the following algorithm:

- (1) Let \mathcal{V}_Q be the set $\{v_q \mid q \in Q\}$ of valuations labeling a state of S . Write f_Q for $\bigvee_{v \in \mathcal{V}_Q} v$.
- (2) Let \mathcal{V}' be the set of all $v \in \mathcal{V}_Q$ s.t. $S \models \gamma(f_Q \wedge \neg v)$. Write f_\wedge for $\bigvee_{v \in \mathcal{V}_Q \setminus \mathcal{V}'} v$.
- (3) If $S \models \gamma(f_\wedge)$ then f_\wedge is the unique minimal solution. Otherwise, there is no solution, or the minimal solution is not unique.

The correctness of this algorithm is easily proved: by construction, f_\wedge is the infimum of all solutions and can only be a solution itself if it is the unique minimal solution.

Therefore uniqueness of the minimal solution can be decided (and that solution be computed) with at most $1 + |Q|$ invocations to a model checking algorithm. \square

Remark 4.5. Theorem 4.4 is not in contradiction with Theorem 1 of [Cha00], where the validity of a CTL query is shown to be EXPTIME-complete. Indeed, we ask here the question of the existence and uniqueness of a minimal solution for one query in the *given* model, and not in all models as in [Cha00].

4.3 Minimal Solutions Without Uniqueness

The ideas behind Theorem 4.4 can be adapted to situations where there is not a unique minimal solution.

Assume we are given a KS $S = \langle Q, q_{\text{init}}, \rightarrow, l \rangle$, a temporal query γ , and k (distinct) minimal solutions f_1, \dots, f_k . Then it is possible to tell whether f_1, \dots, f_k form the complete set of minimal solutions with $O(|Q|^k + |Q|)$ invocations to the underlying model checking algorithm. If the answer is negative, it is furthermore possible to compute an additional f_{k+1} minimal solution.

The algorithm proceeds as follows: assume each minimal solution f_i is given under the form $f_i = \bigvee_{j=1}^{n_i} v_{i,j}$ of a disjunction of n_i single valuations³. Assume f is a solution to γ that is not implied by any f_i . Then $f_i \not\supseteq f$ for every $i = 1, \dots, k$, i.e. there exists a valuation v_{i,r_i} s.t. $f \supseteq \neg v_{i,r_i}$, and $\neg v_{1,r_1} \wedge \dots \wedge \neg v_{k,r_k}$ is a solution. Finally the k minimal solutions f_1, \dots, f_k do not form a complete set iff there is a choice function $(r_i)_{i=1, \dots, k}$ with $1 \leq r_i \leq n_i$ s.t. $f_Q \wedge \neg v_{1,r_1} \wedge \dots \wedge \neg v_{k,r_k}$

³ It is easy to obtain these valuations from S if f_i is given in some other form.

is a solution. Observe that there are at most $n_1 \times \dots \times n_k$, which is $O(|Q|^k)$, candidate solutions.

Once we have a candidate solution f (with $f \supset f_Q$) we can compute a minimal solution f' that entails f with a linear number of invocation to the underlying model checking algorithm. One lists $\llbracket f \rrbracket$ as $\{v_1, \dots, v_n\}$, sets $f' := f$ and repeats the following for $i = 1$ to n : if $S \models \gamma[f' \wedge \neg v_i]$ then $f' := f' \wedge \neg v_i$. When the loop is finished, the resulting f' is a minimal solution. This algorithm is non-deterministic and what f' we finally obtain depends on what enumeration of $\llbracket f \rrbracket$ was started with.

In summary, our methods allow computing a first minimal solution with a linear number of model checking calls, a second one with a quadratic number, a third one with a cubic number, etc.

Every additional minimal solution costs more, and it seems that some form of increasing cost cannot be avoided, as we show in the next section.

5 Counting the Minimal Solutions

In section 4.3 we saw that, for any fixed k , one can decide with a polynomial number of model checking calls, whether there are at most k minimal solutions to a query in some KS (and compute these minimal solutions). Here the degree of the polynomial grows with k so that when k is not fixed (is an input) the reduction is not polynomial-time any more.

This seems inescapable: counting the number of minimal solutions is hard. For a formal proof, we introduce the following two problems: one is a decision problem (yes-or-no output) while the other is a counting problem (numeric output).

ManySols:

Input: a KS S , a query γ , an integer k in unary.

Answer: yes iff there are at least k minimal solutions to γ in S .

CountSols:

Input: a KS S , a query γ .

Answer: the number of minimal solutions to γ in S .

Theorem 5.1 (Hardness of temporal query solving.). *When considering CTL queries:*

1. **ManySols** is NP-complete.
2. **CountSols** is #P-complete.

Furthermore, hardness already appears with the fixed query EG ?.

Proof. That **ManySols** is in NP, and **CountSols** is in #P, is easy to see since a minimal solution can be presented succinctly (it is associated with a subset of the set of states), and minimality is in P for CTL queries (Theorem 4.2).

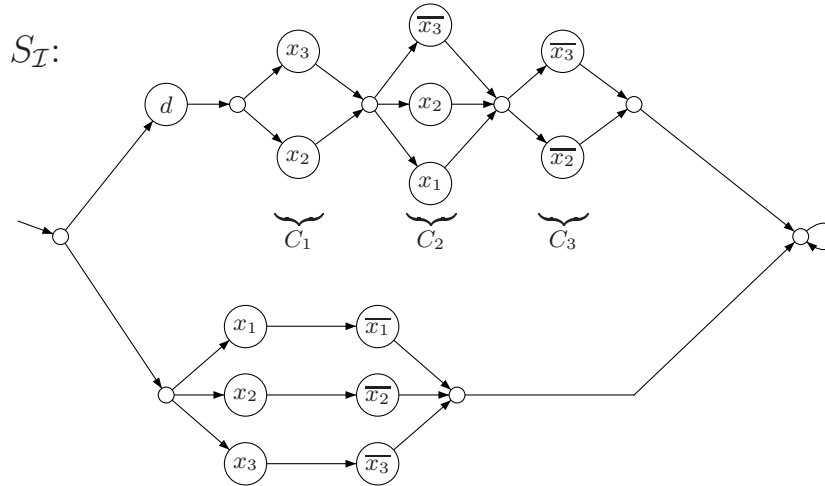


Fig. 3. The KS $S_{\mathcal{I}}$ associated with the SAT instance \mathcal{I} from (1)

Hardness of **ManySols** for NP is proved by reducing from SAT⁴. Assume \mathcal{I} is a SAT instance with n Boolean variables in the form of a CNF with m clauses. With \mathcal{I} we associate a KS $S_{\mathcal{I}}$.

We illustrate the construction on an example: with the following instance

$$\mathcal{I} : \underbrace{x_2 \vee x_3}_{C_1} \wedge \underbrace{x_1 \vee x_2 \vee \overline{x_3}}_{C_2} \wedge \underbrace{\overline{x_2} \vee \overline{x_3}}_{C_3} \tag{1}$$

we associate the KS depicted in Fig. 3.

$S_{\mathcal{I}}$, has two kind of paths: the *bottom paths*, where a variable and its negation are visited, and the *top paths*, where one literal from each clause is picked.

Nodes in $S_{\mathcal{I}}$ are labeled by symbols (or are blank) but these labels stand for valuations: two nodes carry the same valuation iff they are labeled the same in our picture (as in the proof of Prop. 3.2, the exact values of these valuations are not relevant). We consider the CTL query $\gamma = \text{EG?}$: a solution f to γ in $S_{\mathcal{I}}$ must entail all the valuations appearing along some path, and the minimal solutions have the form $\bigvee_{q \in \pi} v_q$ for π a path. Such a solution, written f_{π} , is minimal iff f_{π} is not entailed by some other $f_{\pi'}$.

Clearly, since top paths visit d (dummy), all f_{π} for π a bottom path are minimal. We claim that there exists other minimal solutions if, and only if, \mathcal{I} is satisfiable. Indeed, if a top path π is such that it is not entailed by any $f_{\pi'}$ for a bottom path π' , then π never picks a literal and its negation. Hence π picks a valuation, and that valuation satisfies \mathcal{I} . Reciprocally, if \mathcal{I} is satisfiable, the satisfying valuation gives us (a top path that provides) a solution not entailed by the n minimal “bottom paths” solutions.

⁴ This elegant proof was indicated by Stéphane Demri.

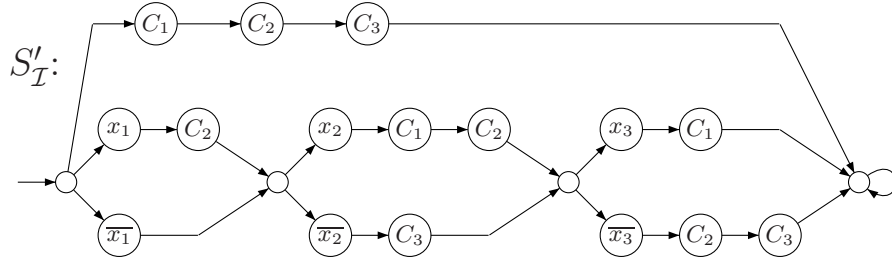


Fig. 4. The KS $S_{\mathcal{I}}$ associated with the SAT instance \mathcal{I} from (1)

Finally, γ has at least $n + 1$ solutions in $S_{\mathcal{I}}$ iff \mathcal{I} is satisfiable.

Hardness of **CountSols** for #P is proved by reducing from #SAT. We illustrate the construction on our earlier example: with (1) we associate the KS depicted in Fig. 4.

In $S'_{\mathcal{I}}$, the top path (denoted by π_{top}) lists all clauses in \mathcal{I} and the other paths provide a way for enumerating all assignments for $\{x_1, \dots, x_n\}$ by visiting, for every Boolean variable, the positive literal x_i , or the negative \bar{x}_i . When a path visits a literal, it immediately visits the clauses that are satisfied by this literal: e.g. C_2 and C_3 follow \bar{x}_3 in the bottom right corner of $S'_{\mathcal{I}}$ because \bar{x}_3 appears in C_2 and in C_3 .

Again, the labels in $S'_{\mathcal{I}}$ stand for valuations and we consider the query $\gamma = \text{EG ?}$. Clearly, if two paths π and π' correspond to different assignments, then they differ on some literal nodes, so that f_{π} and $f_{\pi'}$ do not entail one another. However, if π corresponds to an assignment that satisfies \mathcal{I} , then f_{π} is entailed by $f_{\pi_{\text{top}}}$.

Finally, the number of minimal solutions to γ in $S'_{\mathcal{I}}$ is $2^n + 1 - k$ where k is the number of satisfying assignments for \mathcal{I} . This provides the required weakly parsimonious reduction ⁵. \square

Remark 5.2. There is no contradiction between Theorem 5.1, saying it is hard to count the number of minimal solutions, and Theorem 4.4, saying it is easy to tell if there is a unique one. The situation is similar to #SAT which is #P-complete while it is easy to tell whether a CNF over n Boolean variables has 2^n satisfying assignments (i.e. is valid).

6 Conclusions

We studied the problem of computing the set of minimal solutions to arbitrary temporal queries over arbitrary Kripke structures. We showed this problem is intractable in general.

⁵ Obviously, parsimonious reductions cannot be used since telling whether γ has at least one minimal solution can be done in polynomial-time.

It turns out the problem is easier when one only asks for a few minimal solutions. *A fortiori*, this applies in situations where there is a unique minimal solution (or only a small number of them). Computing a single minimal solution can be done with a linear number of calls to a model checking procedure, i.e. in polynomial-time for CTL queries. Then a second minimal solution can be obtained with a quadratic number of calls, then a third solution with a cubic number, etc. This has some advantages over Chan’s approach where only a very restricted set of so-called valid queries can be handled.

We did not implement our algorithms. The next logical step for this study is to implement and evaluate them, using examples drawn from practical case studies. Such an evaluation will tell whether, in practice, temporal queries often have a small number of minimal solutions, whether only computing a partial sample of the full set of minimal solutions can be useful for understanding and validating the model at hand. In principle, and because it has less limitations, our algorithm must be at least as useful as Chan’s algorithm, for which he provided convincing application examples [Cha00].

Acknowledgments

We thank the anonymous referees who prompted us to develop the results that are now gathered in section 4.3, and Stéphane Demri who indicated the elegant reduction depicted in Fig. 3.

References

- [BBF⁺01] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001. 163, 165
- [BG01] G. Bruns and P. Godefroid. Temporal logic query checking (extended abstract). In *Proc. 16th IEEE Symp. Logic in Computer Science (LICS’2001), Boston, MA, USA, June 2001*, pages 409–417. IEEE Comp. Soc. Press, 2001. 163, 164, 165
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. 171
- [CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999. 163, 165
- [Cha00] W. Chan. Temporal-logic queries. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV’2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463. Springer, 2000. 163, 164, 167, 172, 176
- [CS92] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In *Proc. 3rd Int. Workshop Computer Aided Verification (CAV’91), Aalborg, Denmark, July 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 1992. 171

- [Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 16, pages 995–1072. Elsevier Science, 1990. 165
- [KG96] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996. 171
- [LMS02] F. Laroussinie, N. Markey, and Ph. Schnoebelen. On model checking durational Kripke structures (extended abstract). In *Proc. 5th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2002), Grenoble, France, Apr. 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 264–279. Springer, 2002. 171
- [LST00] F. Laroussinie, Ph. Schnoebelen, and M. Turuani. On the expressivity and complexity of quantitative branching-time temporal logics. In *Proc. 4th Latin American Symposium on Theoretical Informatics (LATIN'2000), Punta del Este, Uruguay, Apr. 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 437–446. Springer, 2000. 171
- [Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 165
- [PBD⁺02] A. C. Patthak, I. Bhattacharya, A. Dasgupta, P. Dasgupta, and P. P. Chakrabarti. Quantified Computation Tree Logic. *Information Processing Letters*, 82(3):123–129, 2002. 171
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA, Oct.-Nov. 1977*, pages 46–57, 1977. 163

Modelling Concurrent Behaviours by Commutativity and Weak Causality Relations^{*}

Guangyuan Guo and Ryszard Janicki

Department of Computing and Software, McMaster University
Hamilton, Ontario L8S 4L7, Canada
{gyguo,janicki}@cas.mcmaster.ca

Abstract. Complex concurrent behaviours that cannot be handled by causal partial orders are frequently modelled by a relational structures $(X, <, \sqsubset)$, where X is a set (of event occurrences), $<$ is *causality* and \sqsubset is *weak causality* relation ([5,9,7,13] and others). It was shown in [7] that the complex case require the pair of relations \diamond and \sqsubset , where \diamond is *commutativity*, however no axioms for the pair \diamond, \sqsubset were given. We present such axioms under the assumption that observations of concurrent behaviours are modelled by stratified partial orders (step-sequences).

1 Introduction

The classical "true concurrency" model semantics make the assumption that all relevant behavioural properties of non-sequential systems can be adequately expressed in terms of causal partial orders. This assumption is arbitrary and the model, although very successful in the majority of applications, is unable to describe properly some aspects of systems with priorities, error recovery systems, inhibitor nets, time testing etc (see for instance [9,7,10,14,17] and many others. The solution, first introduced by Lamport [13] (improved in [1]), Gaifman and Pratt [5], and Janicki and Koutny [6], later fully developed by Janicki and Koutny [9,7], is to use relational structures with two relations. The first relation, denoted by " $<$ " in [9,7], is "causality" (i.e. abstraction of "earlier than"), the second, denoted by " \sqsubset " in [9,7] is called "weak causality" and is an abstraction of "not later than" relation. The classical "interleaving" and "true concurrency" models are distinctive special cases. The papers [9,7] provide theoretical foundations of the model (results of [13,5] are special cases) and prove its soundness. The model has been successfully applied to inhibitor systems [8,2,11], priority systems [10,12], asynchronous races [19,20], synthesis [15], and influenced many other approaches [3,18].

The theory developed in [7] provides a hierarchy of models of concurrency, each model corresponds to a so called "paradigm", or general rule about the structures of behaviours. The paradigms are denoted by π_1 through π_8 . The

^{*} Partially supported by NSERC of Canada Grant and CITO of Ontario Grant.

most restrictive case, π_8 , corresponds to the classical "true concurrency" model where causal partial orders are sufficient to model all aspects of concurrent behaviour. The most general model corresponds to paradigm π_1 . The model where concurrent behaviours are represented by relational structures $(X, <, \sqsubset)$, with X being a set of event occurrences, $<$ interpreted as "causality" and \sqsubset as "weak causality", corresponds to the paradigm π_3 . All the results mentioned above use (mostly implicitly) π_3 assumption. It was proven in [7] that in the most general case concurrent behaviours are represented by relational structures of type (X, \diamond, \sqsubset) , where \diamond is interpreted as "commutativity" (an abstraction of "interleaving"), and that always $< = \diamond \cap \sqsubset$. To illustrate all the relations mentioned above, let us consider the following problem: What are all the *concurrent behaviours* (concurrent histories) that can be built from two events a and b , say under the assumption that both of them can be executed only once? There are four of them and they are all represented by the relations $<, \sqsubset, \diamond$ in the upper part of Figure 1, or, as sets of observations (runs, executions, instances - various names are used in the literature), by the transition systems in the middle part of Fig. 1. Petri nets are used to implement these behaviours under the assumption that we have only one instance of a and b . Problems like this one are frequently referred as "synthesis problems" [15]. Since a and b are "concurrent" then $<$ is empty in all four cases. The relations \diamond and \sqsubset suffice in all cases, while the relations $<$ and \sqsubset in second, third and fourth case from the left. The cases two, three and four conform to paradigm π_3 , the first case does not. All cases conform to paradigm π_1 .

In [9] axiomatic definitions and a comprehensive theory of the relational structures $(X, <, \sqsubset)$ under the paradigm π_3 were given. The general case was left open. *In this paper we generalize the approach of [9] to the relational structures (X, \diamond, \sqsubset) , i.e. to the most general case, however under the assumption that all the observations (executions, runs, instances, etc.) are modelled either by total or stratified partial orders (step-sequences).* In [7] four different classes of observations are considered: total, stratified, interval and general partial orders ([5] assumes stratified, while [13] interval orders). In Figure 1 stratified orders are assumed to be observations, and most of the applications restrict observations to either total or stratified orders. From the purely mathematical viewpoint, the results of both this paper as well as [9] can be seen as an extension of the famous Szpilrajn Theorem [16] to relational structures with two relations.

2 Relational Structure Model of Concurrency

In order to make this paper self-contained, we briefly recall all the main result of [7,9]. The model proposed in [7] supports three level of abstraction: the observation (run, execution, etc.) level, the behaviour (history, invariant) level and the system level, and the development is bottom-up, that is, from the observation level to the system level. In this model, causality is not the only invariant, rather it is merely one of the invariants underlying behaviours of a concurrent system.

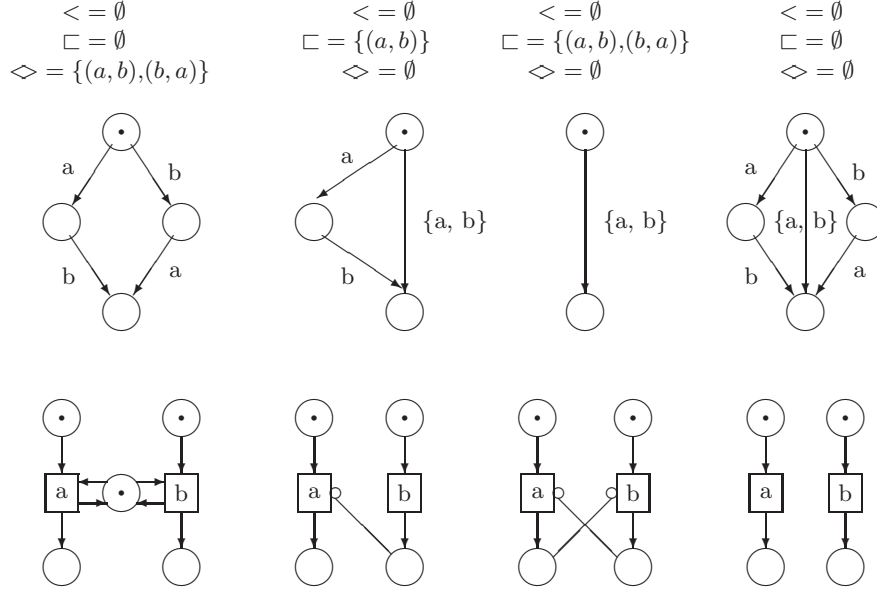


Fig. 1. Examples of *commutativity*, *causality*, and *weak causality*. $\{a,b\}$ denotes *simultaneous* execution of a and b , $<$ denotes *causality*, \sqsubset denotes *weak causality* (abstraction of "not later than"), \diamond denotes *commutativity*

A *partial order*, is a pair $po = (X, <)$ such that X is a non-empty set and $<$ is an irreflexive and transitive relation on X . We say X is the domain of po . Sometime we also say that $<$ is a partial order in X . Two distinct incomparable elements a and b of X will be denoted by $a \sim b$, and we will write $a < \sim b$ if $a < b$ or $a \sim b$.

A partial order $(X, <)$ is said to be

- *total* if for any distinct $a, b \in X$, either $a < b$ or $b < a$.
- *stratified* if $\sim \cup id_X$, where id_X is identity on X , is an equivalence relation;
- *interval*¹ if for all $a, b, c, d \in X$, $a < b \wedge c < d \implies a < d \vee c < b$.

It is easy to see that a total order is a stratified order and a stratified order is an interval one. We will denote the classes of total, stratified, interval and (general) partial orders by \mathcal{TO} , \mathcal{SO} , \mathcal{IO} and \mathcal{PO} respectively. Stratified orders correspond to step sequences. Modelling concurrency usually assume some form of discreteness, for instance the number of predecessors is finite, etc. This leads

¹ The name follows from Fishburn theorem [4] which says: $(X, <)$ is an interval order if and only if there is a total order $(Y, <)$ and two mappings $\varphi, \psi : X \rightarrow Y$ such that $\forall a, b \in X. \varphi(a) < \psi(a)$ and $a < b \iff \psi(a) < \varphi(b)$. Fishburn theorem is frequently used as a definition of interval orders. Usually $\varphi(a)$ is interpreted as the beginning of a and ψ as the end of a .

to the concept of initial finiteness. It turns out many results need separate proofs under initial finiteness assumption. A poset $po = (X, <)$ is said to be *initially finite* if for every $a \in X$, $\{b \mid b < \sim a\}$ is finite. In general, if \mathcal{C} is a class of partial orders we will denote by \mathcal{C}_{IF} the subclass of all initially finite partial orders in \mathcal{C} . A partial order p_1 is an extension of another partial order p_2 if they have the same domain and $<_{p_2} \subseteq <_{p_1}$.

Observation (run, instance of concurrent behaviour) is an abstract model of the execution of a concurrent system. It was argued in [7] that an observation must be an initially finite, either total, or stratified, or interval order. The results of [9] are valid for all kinds of partial orders, not necessarily initially finite nor interval, however separate proofs are frequently required for different cases. In various applications, stratified order (step-sequence) models seem to be the most popular. Following [9,7], we will make a distinction in notation between general posets and those used as observations. We will use $o = (X, \rightarrow_o)$ rather than $po = (X, <)$ to denote a generic observation, and use \leftrightarrow_o rather than \sim to denote incomparability.

Report sets are simply non-empty sets of observations with the same event of occurrences domain.

Invariants of a report set are intrinsic characteristics of the observations of the report set. To illustrate what we mean here, let X be a set of event occurrences and let $< \subseteq X \times X$ be a causality relation specifying a concurrent behaviour. Then every observation $o = (X, \rightarrow_o)$, where \rightarrow_o is the "earlier than" relation, consistent with this specification must satisfy $a < b \Rightarrow a \rightarrow_o b$. Thus the causality relation $<$ is a characteristic of all the observations that are consistent with the behaviour specification. It is an invariant of the report set of observations consistent with the specification.

Formally, for a report set Δ , a *simple relational invariant*, or simply an *invariant*, I of Δ is a relation on the domain of Δ , $dom(\Delta)$, such that

$$(a, b) \in I \Leftrightarrow a \neq b \wedge o \in \Delta. \Phi(a, b, o),$$

where $\Phi(a, b, o)$ is a formula defined by the grammar:

$$\Phi \rightarrow true \mid false \mid a \rightarrow_o b \mid a \leftarrow_o b \mid a \leftrightarrow_o b \mid \neg \Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi.$$

The set of (simple relational) invariants of Δ will be denoted by $SRI(\Delta)$. $SRI(\Delta)$ clearly includes the relations $<_\Delta$, $>_\Delta$, \leftrightarrow_Δ , \diamond_Δ , \sqsubset_Δ and \sqsupset_Δ defined on $dom(\Delta)$ as follows:

$$\begin{aligned} <_\Delta &= \bigcap_{o \in \Delta} \rightarrow_o & >_\Delta &= \bigcap_{o \in \Delta} (\rightarrow_o)^{-1} \\ \leftrightarrow_\Delta &= \bigcap_{o \in \Delta} \leftrightarrow_o & \diamond_\Delta &= \bigcap_{o \in \Delta} (\rightarrow_o \cup (\rightarrow_o)^{-1}) \\ \sqsubset_\Delta &= \bigcap_{o \in \Delta} (\rightarrow_o \cup \leftrightarrow_o) & \sqsupset_\Delta &= \bigcap_{o \in \Delta} ((\rightarrow_o)^{-1} \cup \leftrightarrow_o). \end{aligned}$$

The above six invariants are all the nontrivial simple relational invariants of Δ . More precisely, it can be shown [7] that $SRI(\Delta) = \{\emptyset, <_\Delta, >_\Delta, \leftrightarrow_\Delta, \diamond_\Delta, \sqsubset_\Delta, \sqsupset_\Delta, R_{-id}\}$, where $R_{-id} = dom(\Delta) \times dom(\Delta) - id_{dom(\Delta)}$. $<_\Delta$ and $>_\Delta$ are usually called or interpreted as *causalities*, \diamond_Δ as *commutativity*, \leftrightarrow_Δ as

synchronisation, and \sqsubset_{Δ} and \sqsupset_{Δ} as *weak causalities*. Since $<_{\Delta} = (>_{\Delta})^{-1}$ and $\sqsubset_{\Delta} = (\sqsupset_{\Delta})^{-1}$, we only need to consider four non-trivial invariants: $<_{\Delta}$, $>_{\Delta}$, \diamond_{Δ} and \sqsubset_{Δ} . Furthermore, since $<_{\Delta} = \sqsubset_{\Delta} \cap \diamond_{\Delta}$ and $\leftrightarrow_{\Delta} = \sqsubset_{\Delta} \cap (\sqsupset_{\Delta})^{-1}$, we actually only need to consider \sqsubset_{Δ} and \diamond_{Δ} .

Given a report set Δ , we define a report set $\Delta^{(SRI)}$ associated with Δ to be the set of all observations on the domain X of Δ such that for all $a, b \in X$,

$$\begin{aligned} a <_{\Delta} b &\Rightarrow a \rightarrow_o b & a \leftrightarrow_{\Delta} b &\Rightarrow a \leftrightarrow_o b \\ a \diamond_{\Delta} b &\Rightarrow (a \rightarrow_o b \vee b \rightarrow_o a) & a \sqsubset_{\Delta} b &\Rightarrow (a \rightarrow_o b \vee a \leftrightarrow_o b). \end{aligned}$$

Clearly $\Delta^{(SRI)} \supseteq \Delta$ by definition.

Definition 1 ([7]). *A concurrent behaviour or concurrent history, is a report set Δ such that $\Delta = \Delta^{(SRI)}$.* \square

To illustrate the above definition, assume that $\diamond_{\Delta} = \leftrightarrow_{\Delta} = \emptyset$. Then $<_{\Delta} = \sqsubset_{\Delta}$. In such a case $\Delta^{(SRI)}$ is just the set of all total order extensions of $<_{\Delta}$. This case is handled by Szpilrajn Theorem [16], that says "each partial order is completely defined (as an intersection) by the set of all its total extensions". Of course each partial order defines uniquely its set of total extensions, so the relationship between a partial order and the set of all its total extensions is "one-to-one". The problem is how to find axioms for the relations \diamond and \sqsubset such that their partial order extensions could be interpreted as some $\Delta^{(SRI)}$. To solve this problem the notion of a *paradigm* has been introduced.

As we mentioned earlier, a paradigm is a superposition or a statement about the structure of a history. For instance, let Δ be a concurrent history. The classical causality based approach usually stipulates that if there is an observation $o \in \Delta$ such that $a \leftrightarrow_o b$, then there must be an observation such a precedes b and an observation such that b precedes a . Such stipulations or rules relating different observations in a history are called paradigms. Formally, *paradigms*, $\omega \in Par$, are defined by

$$\omega := true | false | \Psi_1 | \Psi_2 | \Psi_3 | \neg\omega | \omega \vee \omega | \omega \wedge \omega | \omega \Rightarrow \omega,$$

where $\Psi_1(\beta, \gamma) = \exists o. \beta \rightarrow_o \gamma$, $\Psi_2(\beta, \gamma) = \exists o. \beta \leftarrow_o \gamma$ and $\Psi_3(\beta, \gamma) = \exists o. \beta \leftrightarrow_o \gamma$.

A history Δ satisfies a paradigm $\omega \in Par$ if for all distinct $a, b \in dom(\Delta)$, $\omega(a, b)$ holds. It can be shown (see [7]) that in the study of concurrent histories, we only need to consider 8 paradigms, denoted by π_1, \dots, π_8 . The most restrictive paradigm, π_8 , admits concurrent histories Δ such that

$$\exists o \in \Delta. a \leftrightarrow_o b \iff (\exists o \in \Delta. a \leftarrow_o b) \wedge (\exists o \in \Delta. a \rightarrow_o b).$$

It was proven that in this case causality, $<$, is the only necessary invariant, so the use of it to model concurrent behaviour is justified. The paradigm π_3 , which is general enough to deal with most problems that cannot be dealt with under π_8 , admits concurrent histories Δ such that

$$(\exists o \in \Delta. a \leftarrow_o b) \wedge (\exists o \in \Delta. a \rightarrow_o b) \implies \exists o \in \Delta. a \leftrightarrow_o b.$$

It was proven that in this case, causality, $<$, and weak causality (an abstraction of "not later than"), \sqsubset , are the only necessary invariants. The axioms for relational structures $(X, <, \sqsubset)$, such that the sets of their partial order extensions can be interpreted as concurrent histories $\Delta^{(SRI)}$, were provided in [9]. We briefly show them below.

Definition 2 ([9]). A two relation structure, or simply a structure, is a triple $S = (X, <, \sqsubset)$ where X is a non-empty set and $<, \sqsubset$ are two irreflexive binary relations on X such that for all $a, b \in X$, we have $a < b \Rightarrow \neg b \sqsubset a$. \square

Note that at this point we do not assume any other properties of $<$ and \sqsubset ! Until further notice $<$ and \sqsubset do not have any interpretation now. If $po = (X, <)$ is a partially ordered set, then $(X, <, \sim)$ is a structure which we denote by $\mathcal{S}(po)$.

For a set of relational structures with the same domain, we define their intersection component-wise. We say a relational structure T is an extension of S and write $S \subseteq T$ if $X_S = X_T$, $<_S \subseteq <_T$ and $\sqsubset_S \subseteq \sqsubset_T$.

A relational structure is said to be saturated if for all distinct $a, b \in X$,

$$\neg a < b \Rightarrow b \sqsubset a.$$

For any poset po , the structure $\mathcal{S}(po)$ is saturated. If \mathcal{R} is a class of relational structures, we will denote by \mathcal{R}^{sat} the subclass of all the saturated ones in \mathcal{R} .

A structure $(X, <, \sqsubset)$ is said to be *initially finite* if X is countable and $\{b \mid b \sqsubset a\}$ is finite for all $a \in X$. As with partial orders, if \mathcal{R} is a class of structures, we denote by $\mathcal{R}_{IF} \subseteq \mathcal{R}$ the subclass consisting of initially finite structures.

Definition 3 ([9]). A relational structure $S = (X, <, \sqsubset)$ is called a total, stratified, interval and partial order structure if the following conditions T1-T3, S1-S4, I1-I6 and P1-P4 are satisfied respectively:

$$\begin{array}{ll} T1. a \not\sqsubset a & T3. a < b < c \Rightarrow a < c \\ T2. a < b \Leftrightarrow a \sqsubset b & \\ \\ S1. a \not\sqsubset a & S3. a \sqsubset b \sqsubset c \Rightarrow a \sqsubset c \vee a = c \\ S2. a < b \Rightarrow a \sqsubset b & S4. a \sqsubset b < c \vee a < b \sqsubset c \Rightarrow a < c. \\ \\ I1. a \not\sqsubset a & I4. a < b \sqsubset c \vee a \sqsubset b < c \Rightarrow a \sqsubset c \\ I2. a < b \Rightarrow a \sqsubset b & I5. a < b \sqsubset c < d \Rightarrow a < d \\ I3. a < b < c \Rightarrow a < c & I6. a \sqsubset b < c \sqsubset d \Rightarrow a \sqsubset d \vee a = d \\ \\ P1. a \not\sqsubset a & P3. a < b < c \Rightarrow a < c \\ P2. a < b \Rightarrow a \sqsubset b & P4. a \sqsubset b < c \vee a < b \sqsubset c \Rightarrow a \sqsubset c. \quad \square \end{array}$$

We will denote by \mathcal{T} , \mathcal{S} , \mathcal{I} and \mathcal{P} respectively the class of total, stratified, interval and partial order structures. Note that total order structures are nothing but partial orders. A discussion of differences between the above axioms (from [9]) and those of [13,5] can be found in [9].

Definition 4 ([9]). A class \mathcal{R} of structures is extension complete if for every $S \in \mathcal{R}$, the set $Q = \{T \in \mathcal{R}^{sat} \mid S \subseteq T\}$ is non-empty and $S = \bigcap_{T \in Q} T$. \square

In other words, \mathcal{R} is extension complete if any structure in \mathcal{R} can be represented using its saturated extensions comprised by \mathcal{R} .

For an order structure $S = (X, <, \sqsubset)$, let $\mathcal{P}(S)$ be the pair $(X, <)$. The following lemma shows that saturated structures are essentially partial orders.

Lemma 1 ([9]). Let $\mathcal{X} \in \{\mathcal{T}, \mathcal{S}, \mathcal{I}, \mathcal{P}\}$. Then

$$\begin{array}{ll} i. \mathcal{P}(\mathcal{X}^{sat}) = \mathcal{X}\mathcal{O} & iii. \mathcal{S}(\mathcal{X}\mathcal{O}) = \mathcal{X}^{sat} \\ ii. \mathcal{P}(\mathcal{X}_{IF}^{sat}) = \mathcal{X}\mathcal{O}_{IF} & iv. \mathcal{S}(\mathcal{X}\mathcal{O}_{IF}) = \mathcal{X}_{IF}^{sat}. \end{array} \quad \square$$

The main result of [9] is the following theorem.

Theorem 1 (Generalisation of Szpilrajn Theorem [9]). The classes of ordered structures: $\mathcal{T}, \mathcal{S}, \mathcal{I}, \mathcal{P}, \mathcal{T}_{IF}, \mathcal{S}_{IF}, \mathcal{I}_{IF}, \mathcal{P}_{IF}$ are extension complete. \square

Since the saturated extensions of ordered structures are appropriate partial orders, the sets Q from the Definition 4 can be proven to be concurrent histories. This means the ordered structures can be used to model concurrent behaviours conforming to paradigm π_3 . It is important to point out that the result of [9,13,5] are only valid under π_3 , which suffice for most of the application, but it is not the most general case. Under π_3 we *have to* model the leftmost case of Figure 1 by two sequential behaviours instead of more natural one concurrent behaviour.

3 Generalized Order Structures and Concurrent Histories

This chapter is devoted to the new results. We start with the main new definition.

Definition 5. A generalised structure is a triple $GS = (X, \diamond, \sqsubset)$ such that

- (a) $X \neq \emptyset$, \diamond and \sqsubset are two irreflexive relations on X , and \diamond is symmetric.
- (b) If $< = \diamond \cap \sqsubset$, then $S_{GS} = (X, <, \sqsubset)$ is a structure. \square

For a set of generalised structures with the same domain, we define their intersection component-wise. We say a generalised structure GS_2 is an extension of GS_1 and write $GS_1 \subseteq GS_2$ if $X_{GS_1} = X_{GS_2}$, $\diamond_{GS_1} \subseteq \diamond_{GS_2}$ and $\sqsubset_{GS_1} \subseteq \sqsubset_{GS_2}$.

Corresponding to four classes of order structures, we also have four classes of generalised order structures (GOS).

- Definition 6.** (a) GS is total if S_{GS} in (b) of Definition 5 is a total order structure (axioms T1-T3) and additionally $\diamond = X \times X - id_X$.
- (b) GS is stratified, interval and partial if S_{GS} in (b) of the Definition 5 is a stratified order structure (axioms S1-S4), an interval order structure (axioms I1-I6) or a partial order structure (axioms P1-P4), respectively. \square

We shall use \mathcal{GT} , \mathcal{GS} , \mathcal{GI} and \mathcal{GP} to denote respectively the classes of total, stratified, interval and partial order GOS's. Again note that total GOS's are just posets in disguise.

A GOS $GS = (X, \diamond, \sqsubset)$ is said to be *initially finite* if X is countable and $\{b \mid b \sqsubset a\}$ is finite for all $a \in X$. If \mathcal{H} is a class of GOS's, then \mathcal{H}_{IF} will denote the subclass of all initially finite GOS's in \mathcal{H} . Thus, for example, \mathcal{GS}_{IF} will denote the class of initially finite stratified GOS's.

If $po = (X, <)$ is a total order, then $(X, X \times X - id_X, <^\sim)$ is a GOS which we denote by $\mathcal{G}(po)$. If $po = (X, <)$ is a partial but not total order, then $(X, < \cup <^{-1}, <^\sim)$ is a GOS denoted by $\mathcal{G}(po)$.

Similarly, if $S = (X, <, \sqsubset)$ is a total structure, then $(X, X \times X - id_X, \sqsubset)$ is a GOS which we denote by $\mathcal{G}(S)$. If $S = (X, <, \sqsubset)$ is a stratified, interval, or partial order structure, then $(X, < \cup <^{-1}, \sqsubset)$ is a *stratified, interval, or partial order GOS* denoted by $\mathcal{G}(S)$.

A (total, stratified, interval, partial) GOS $G = (X, \diamond, \sqsubset)$ is said to be *saturated* if there is a saturated (total, stratified, interval, partial respectively) order structure S such that $G = \mathcal{G}(S)$. If \mathcal{H} is a class of GOS, we will denote by \mathcal{H}^{sat} the subclass of all the saturated ones in \mathcal{H} .

Definition 7. A class \mathcal{H} of (total, stratified, interval, partial) GOS's is *extension complete* if for every $GS \in \mathcal{H}$, the set $Q = \{GT \in \mathcal{H}^{sat} \mid GS \subseteq GT\}$ is non-empty and $GS = \bigcap_{GT \in Q} T$. \square

For a GOS $GS = (X, \diamond, \sqsubset)$, let $\mathcal{P}(GS)$ be the partial order $(X, \diamond \cap \sqsubset)$.

Lemma 2. For $\mathcal{X} \in \{\mathcal{GT}, \mathcal{GS}, \mathcal{GI}, \mathcal{GP}\}$, we have $\mathcal{P}(\mathcal{X}^{sat}) = \mathcal{X}\mathcal{O}$, $\mathcal{G}(\mathcal{X}\mathcal{O}) = \mathcal{X}^{sat}$, $\mathcal{P}(\mathcal{X}_{IF}^{sat}) = \mathcal{X}\mathcal{O}_{IF}$, and $\mathcal{G}(\mathcal{X}\mathcal{O}_{IF}) = \mathcal{X}_{IF}^{sat}$.

Proof. Directly from the definitions and Lemma 1 \square

In other words, *saturated GOS's are partial orders*. The main mathematical results of this papers are the following four lemmas.

We say that a partial order $po = (X, <)$ *extends* a GOS GS if $GS \subseteq \mathcal{G}(po)$.

Lemma 3 (Necessary and sufficient conditions for extension completeness). Let $GS = (X, \diamond, \sqsubset)$ be a GOS and Q be any non-empty set of partial orders that extend GS . Then, $GS = \bigcap_{T \in Q} \mathcal{G}(T)$, if and only if for all distinct $a, b \in X$ we have:

- (a) $\neg a \diamond b \implies \exists T \in Q. (b <_{\tilde{T}} a \wedge a <_{\tilde{T}} b)$.
- (b) $\neg a \sqsubset b \implies \exists T \in Q. b <_T a$.

Proof. (\implies) Suppose $G = \bigcap_{T \in Q} \mathcal{G}(T)$, that is, $\diamond = \bigcap_T (<_T \cup <_T^{-1})$ and $\sqsubset = \bigcap_{T \in Q} <_{\tilde{T}}$. If $\neg a \diamond b$, $\exists T \in Q$ such that $\neg a <_T b$ and $\neg b <_T a$. It follows that $b <_{\tilde{T}} a \wedge a <_{\tilde{T}} b$. Similarly, if $\neg a \sqsubset b$, then $\exists T \in Q$ such that $\neg a <_{\tilde{T}} b$. Thus $b <_T a$.

(\impliedby) Suppose (a) and (b) hold. Clearly $G \subseteq \bigcap_{T \in Q} \mathcal{G}(T)$. We show that $G \supseteq \bigcap_{T \in Q} \mathcal{G}(T)$, that is, $\diamond \supseteq \bigcap_T (<_T \cup <_T^{-1})$ and $\sqsubset \supseteq \bigcap_{T \in Q} <_{\tilde{T}}$. We argue by

contradiction. If $\neg \diamond \supseteq \bigcap_T (<_T \cup <_T^{-1})$, then there is a pair of distinct $a, b \in X$ such that $\neg a \diamond b$ and $\forall T \in Q, a <_T b \vee b <_T a$, contradicting (a). Similarly, if $\neg \sqsubset \supseteq \bigcap_{T \in Q} <_{\tilde{T}}$, then there is a pair of distinct $a, b \in X$ such that $\neg a \sqsubset b$ and $\forall T \in Q, a <_{\tilde{T}} b$, contradicting (b). \square

A poset $(Q, <)$ such that for any $r, s \in Q$ there is a $t \in Q$ satisfying $r \preceq t \wedge s \preceq t$ where $\preceq = < \cup \text{Id}_Q$ is called a lattice. In particular, if $<$ is a total order in Q , then $(Q, <)$ is a lattice.

Lemma 4. *Let $(Q, <)$ be a lattice and $\{S_t = (X, <_t, \sqsubset_t), t \in Q\}$ be a class of (stratified or interval) order structures such that $t_1 \preceq t_2 \Rightarrow S_{t_1} \subseteq S_{t_2}$, then*

$$S = \bigcup_{t \in Q} S_t$$

is also a (stratified or interval, respectively) order structure.

Proof. We show that if every $S_t, t \in Q$ is a stratified order structure, then $\bigcup_{t \in Q} S_t$ is also a stratified order structure. The proof for the other cases are the same.

Write $<_S = \bigcup_{t \in Q} (<_t)$ and $\sqsubset_S = \bigcup_{t \in Q} (\sqsubset_t)$.

S1: Since $a \not\sqsubset_t a$ for any $t \in Q$, $a \not\sqsubset_S a$.

S2: If $a <_S b$, then by definition there is $t \in Q$ such that $a <_t b$, hence $a \sqsubset_t b$.

So $a \sqsubset_S b$.

S3: If $a \sqsubset_S b \sqsubset_S c$, then by definition there exist t_1 and t_2 such that $a \sqsubset_{t_1} b$ and $b \sqsubset_{t_2} c$. By the hypothesis, Q is a lattice, so there is a $t \in Q$ such that $t_1 \preceq t$ and $t_2 \preceq t$, hence $S_{t_1} \subseteq S_t$ and $S_{t_2} \subseteq S_t$. It follows that $a \sqsubset_{S_t} b$ and $b \sqsubset_{S_t} c$, hence $a \sqsubset_{S_t} c \vee a = c$. Therefore $a \sqsubset_S c \vee a = c$.

S4: Similar to the proof of S3. \square

For later use, for any order structure $S = (X, <, \sqsubset)$ we define the following sets [9]:

$$\begin{aligned} (W_S)_a &= \{a\} \cup \{c \mid c \sqsubset a\} & (W_S)^a &= \{a\} \cup \{c \mid a \sqsubset c\} \\ (Y_S)_a &= \{a\} \cup \{c \mid c < a\} & (Y_S)^a &= \{a\} \cup \{c \mid a < c\}. \end{aligned}$$

We also define $\sum_S^a = (W_S)_a = \{a\} \cup \{c \mid c \sqsubset a\}$.

Lemma 5. *Let $G = (X, \diamond, \sqsubset) \in \mathcal{GS}_{IF}$ ($G \in \mathcal{GS}$) and let $a, b \in X$ be any two distinct elements such that $\neg b \sqsubset a$. There exists $T \in \mathcal{SO}_{IF}$ ($T \in \mathcal{SO}$) such that T extends G and $a <_T b$.*

Proof. By Lemma 2, it suffices to show that there exists a $T \in \mathcal{S}_{IF}^{sat}$ that extends G and $a <_T b$.

By definition, $S = (X, <, \sqsubset)$, where $< = \diamond \cap \sqsubset$, is in \mathcal{S}_{IF} . Consider the structure $S_0 = (X, <_0, \sqsubset_0)$ where

$$<_0 = < \cup (W_S)_a \times (W_S)^b,$$

and

$$\sqsubset_0 = \sqsubset \cup (W_S)_a \times (W_S)^b.$$

Clearly $S \subseteq S_0$, $a <_0 b$ and $<_0 \cap \diamond = \sqsubset_0 \cap \diamond$. We show that $S_0 \in \mathcal{S}_{IF}$.

S1 and S2 hold trivially. To show that S3 holds, suppose $x \sqsubset_0 y \sqsubset_0 z$, since $\neg b \sqsubset a$, $(W_S)_a \cap (W_S)^b = \emptyset$, so $(x, y) \in (W_S)_a \times (W_S)^b \wedge (y, z) \in (W_S)_a \times (W_S)^b$ is impossible, thus i) $x \sqsubset y \sqsubset z$, or ii) $x \sqsubset y \wedge (y, z) \in (W_S)_a \times (W_S)^b$, or iii) $(x, y) \in (W_S)_a \times (W_S)^b \wedge y \sqsubset z$. If i) holds, then $x \sqsubset z \vee x = z$, thus $x \sqsubset_0 z \vee x = z$; If ii) holds, $x \sqsubset y \sqsubset a$, implying $x \sqsubset a \vee x = a$, hence $x \in (W_S)_a$. Then $(x, z) \in (W_S)_a \times (W_S)^b$, therefore $x \sqsubset_0 z$; Similarly, if iii) holds, then $z \in (W_S)^b$ and $(x, z) \in (W_S)_a \times (W_S)^b$, so $x \sqsubset_0 z$. So in any case, $x \sqsubset_0 y \sqsubset_0 z \implies x \sqsubset_0 z \vee x = z$, that is S3 holds. Similarly, S4 is satisfied by S_0 . Finally, it is easy to see that for any $h \in X$,

$$\sum_{S_0}^h \subseteq \sum_S^h \cup \sum_S^a.$$

Therefore S_0 is also initially finite.

Since S is initially finite, X is countable, hence we may assume that there is a well order, $<$ say, of X such that each element in X is preceded only by finite elements. We extend $<$ lexicographically to $X \times X$ and we denote $\preceq = < \cup id_X$.

We now show, by induction, that for each pair $(c, d) \in X \times X$, there is an $S_{cd} \in \mathcal{S}_{IF}$ such that for all pairs $(c, d), (e, f) \in X \times X$,

$$S_0 \subseteq S_{cd}, \quad (I)$$

$$(c, d) \prec (e, f) \implies S_{cd} \subseteq S_{ef}, \quad (II)$$

$$c \neq d \implies c <_{S_{cd}} d \vee d \sqsubset_{S_{cd}} c, \quad (III)$$

$$<_{S_{cd}} \cap \diamond = \sqsubset_{S_{cd}} \cap \diamond. \quad (IV)$$

Moreover, for all $g \in X$,

$$\sum_{S_{ef}}^g \subseteq \sum_{S_0}^g \cup (\bigcup_{h \preceq e} \sum_{S_0}^h). \quad (V)$$

Let (m, m) be the minimal pair in $X \times X$. Set $S_{mm} = S_0$. Of course $S_{mm} \in \mathcal{S}_{IF}$, and clearly it satisfies (I), (II), (III), (IV) and (V).

Now let $(e, f) \in X \times X$ be a pair such that $S_{cd} \in \mathcal{S}_{IF}$ are defined for all $(c, d) \prec (e, f)$ and satisfy (I), (II), (III), (IV), and (V). There are only finite such (c, d) 's, so there is a maximal pair, (c_M, d_M) say. If $e = f \vee e <_{S_{c_M d_M}} f \vee f \sqsubset_{S_{c_M d_M}} e$, then define $S_{ef} = S_{c_M d_M}$; else set $S_{ef} = (X, <_{S_{ef}}, \sqsubset_{S_{ef}})$ where

$$<_{S_{ef}} = <_{S_{c_M d_M}} \cup (W_{S_{c_M d_M}})_e \times (W_{S_{c_M d_M}})^f,$$

and

$$\sqsubset_{S_{ef}} = \sqsubset_{S_{c_M d_M}} \cup (W_{S_{c_M d_M}})_e \times (W_{S_{c_M d_M}})^f.$$

As shown above, $S_{ef} \in \mathcal{S}_{IF}$. It is also clear that S_{ef} satisfies (I), (II), (III) and (IV) above. For any $g \in X$,

$$\sum_{S_{ef}}^g \subseteq \sum_{S_{c_M d_M}}^g \cup \sum_{S_{c_M d_M}}^e.$$

By (V), we obtain

$$\sum_{S_{ef}}^g \subseteq (\sum_{S_0}^g \cup (\bigcup_{h \preceq c_M} \sum_{S_0}^h)) \cup (\sum_{S_0}^e \cup (\bigcup_{h \preceq c_M} \sum_{S_0}^h)) \subseteq \sum_{S_0}^g \cup (\bigcup_{h \preceq e} \sum_{S_0}^h).$$

Thus S_{ef} also satisfies (V).

Next define

$$T = \bigcup_{(c,d) \in X \times X} S_{cd}.$$

By Lemma 4, $T \in \mathcal{S}$. Clearly, $a <_T b$. By (III) T is saturated. To show that T is also initially finite, observe that for any $g \in X$,

$$\sum_T^g \subseteq \bigcup_{(c,d) \in X \times X} \sum_{S_{cd}}^g = \bigcup_{d \in X} ((\bigcup_{c \prec g} \sum_{S_{cd}}^g) \cup \sum_{S_{gd}}^g \cup (\bigcup_{g \prec c} \sum_{S_{cd}}^g)).$$

By (II), it is clear that

$$\bigcup_{c \prec g} \sum_{S_{cd}}^g \subseteq \sum_{S_{gd}}^g.$$

Suppose $x \in \bigcup_{g \prec c} \sum_{S_{cd}}^g$. If $x = g$, then $x \in \sum_{S_{gd}}^g$. Else there is a pair (c, d) with $g \prec c$ such that $x \sqsubset_{S_{cd}} g$ which implies $\neg g <_{S_{cd}} x$. By (II), $\neg g <_{S_{gd}} x$, which in turn, by (III), implies that $x \sqsubset_{S_{gd}} g$. It follows that $x \in \sum_{S_{gd}}^g$. Thus,

$$\bigcup_{g \prec c} \sum_{S_{cd}}^g \subseteq \sum_{S_{gd}}^g.$$

Therefore

$$\sum_T^g \subseteq \bigcup_{d \in X} \sum_{S_{gd}}^g \subseteq \bigcup_{h \preceq g} \sum_{S_0}^h. \quad (*)$$

Since there are only finitely many elements preceding g and S_0 is initially finite, we see that the last union in $(*)$ is a finite set. This shows that T is initially finite.

Finally, we show that T is an extension of G . By construction of T , $\sqsubset \subseteq \sqsubset_T$. We only need to show that $\diamond \subseteq (<_T \cup (<_T)^{-1})$. Suppose $c \diamond d$, then $c \neq d$. By (III) $c <_T d \vee d <_T c$. If $c <_T d$, we are done. Otherwise $d <_T c$. By (IV), $<_T \cap \diamond = \sqsubset_T \cap \diamond$, hence $d <_T c$. So in any case, $c <_T d \vee d <_T c$. This completes the proof of Lemma 5 for initially finite stratified GOS. For just a stratified GOS we proceed identically, we only need to omit all references to initial finiteness. \square

Lemma 6. *Let $G = (X, \diamond, \sqsubset) \in \mathcal{G}_{IF}$ ($G \in \mathcal{GS}$) and $a, b \in X$ be any distinct pair such that $\neg b \diamond a$. There exists $T \in \mathcal{T} \in \mathcal{SO}_{IF}$ (SO) such that T extends G and $a <_{\tilde{T}} b \wedge b <_{\tilde{T}} a$.*

Proof. Since \diamond is symmetric, we also have $\neg a \diamond b$. Let S be as in Lemma 5. We first show that there is an $S_0 \in \mathcal{S}_{IF}$ such that S_0 extends S , $a \sqsubset_{S_0} b$, $b \sqsubset_{S_0} a$ and $<_{S_0} \cap \diamond = \sqsubset_{S_0} \cap \diamond$.

Define $S'_0 \in \mathcal{S}_{IF}$ as follows: If $a \sqsubset b$, then $S'_0 = S$; else $S'_0 = (X, <_{S'_0}, \sqsubset_{S'_0})$ where

$$<_{S'_0} = < \cup (Y_{S'}^a \times (Y_S)^b - (a, b),$$

and

$$\sqsubset_{S'_0} = \sqsubset \cup (Y_{S'}^a \times (Y_{S'}^b).$$

Clearly $S' \subseteq S'_0$ and $a \sqsubset_{S'_0} b$. And a routine check shows that $S'_0 \in \mathcal{S}_{IF}$. Moreover, it is also clear that $<_{S'_0} \cap \diamond = \sqsubset_{S'_0} \cap \diamond$.

Define S_0 as follows: If $b \sqsubset_{S'_0} a$, then $S_0 = S'_0$; else $S_0 = (X, <_{S_0}, \sqsubset_{S_0})$ where

$$<_{S_0} = <_{S'_0} \cup (Y_{S'_0}^b \times (Y_{S'_0}^a)^a - \{(b, a)\},$$

and

$$\sqsubset_{S_0} = \sqsubset_{S'_0} \cup (Y_{S'_0}^b \times (Y_{S'_0}^a)^a.$$

Again it is clear that $S' \subseteq S'_0 \subseteq S_0$ and $a \sqsubset_{S_0} b$ and $b \sqsubset_{S_0} a$. And a routine check shows that $S'_0 \in \mathcal{S}_{IF}$. Moreover, it is also clear that $<_{S_0} \cap \diamond = \sqsubset_{S_0} \cap \diamond$.

Now by the same construction in the proof of Lemma 5 above, we see that there exists a $T \in \mathcal{S}_{IF}^{sat}$ such that T extends G and $a \sqsubset_T b$ and $b \sqsubset_T a$. By Lemma 1, there is a $T \in \mathcal{SO}_{IF}^{sat}$ that extends G and $a <_{\tilde{T}} b$ and $b <_{\tilde{T}} a$. This completes the proof of Lemma 6 for initially finite stratified GOS. For just a stratified GOS we proceed identically, we only need to omit all references to initial finiteness. \square

Lemmas 5 and 6 imply that the necessary and sufficient conditions in Lemma 3 for extension completeness are satisfied by the class \mathcal{GS}_{IF} and \mathcal{GS} . Therefore the classes \mathcal{GS}_{IF} and \mathcal{GS} are extension complete. *The main results of this paper can now be formulated as follows* (please compare with Theorem 1):

Theorem 2. *The classes of generalized ordered structures \mathcal{GT} , \mathcal{GS} , \mathcal{GT}_{IF} , \mathcal{GS}_{IF} are extension complete.* \square

Below we discuss axiomatic representations of histories satisfying π_1 . Our main result is that *every initially finite stratified GOS corresponds in a natural way to a history, if we restrict observations to initially finite stratified orders*. Whether or not this is also true for the more general interval GOS is currently unknown. We believe it is also true. Many properties of GOS and concurrent history we used to show the result for stratified GOS also hold for interval GOS. And the first two results below are stated for both interval and stratified GOS.

Proposition 1. *If observations are modelled by initially finite stratified (interval) orders and Δ is a history, then $GS_{\Delta} = (X_{\Delta}, \diamond_{\Delta}, \sqsubset_{\Delta}) \in \mathcal{GS}_{IF}$ ($GS_{\Delta} \in \mathcal{GT}_{IF}$).*

Proof. We prove the result for observations modelled on interval orders. The proof of the other case is similar.

Clearly \diamond_{Δ} and \sqsubset_{Δ} are both irreflexive, and \diamond_{Δ} is symmetric. Also, $<_{\Delta} = (\diamond_{\Delta} \cap \sqsubset_{\Delta})$. Finally, it follows from Proposition 3.3 of [9] that $(X_{\Delta}, <_{\Delta}, \sqsubset_{\Delta}) \in \mathcal{I}_{IF}$. \square

Proposition 2. *Let Δ and Δ' be two histories. $\Delta = \Delta'$ iff $GS_{\Delta} = GS_{\Delta'}$.*

Proof. (\implies) Trivial.

(\impliedby) Suppose $GS_{\Delta} = GS_{\Delta'}$, then $X_{\Delta} = X_{\Delta'}$, $\diamond_{\Delta} = \diamond_{\Delta'}$ and $\sqsubset_{\Delta} = \sqsubset_{\Delta'}$. It follows that $<_{\Delta} = \diamond_{\Delta} \cap \sqsubset_{\Delta} = \diamond_{\Delta'} \cap \sqsubset_{\Delta'} = <_{\Delta'}$ and $<_{\Delta} = \sqsubset_{\Delta} \cap (\sqsubset_{\Delta})^{-1} = \sqsubset_{\Delta'} \cap (\sqsubset_{\Delta'})^{-1} = <_{\Delta'}$. Thus $\Delta = \Delta^{(SRI)} = \Delta'^{(SRI)} = \Delta'$. \square

For the rest of results, observations are restricted to initially finite stratified orders. For each initially finite stratified GOS GS , we can define

$$obs(GS) = \{(X, \rightarrow) \in \mathcal{SO}_{IF} \mid GS \subseteq (X, < \cup (<)^{-1}, < \sim)\}$$

Proposition 3. *For every $GS = (X, \diamond, \sqsubset) \in \mathcal{GS}_{IF}$,*

- (a) $GS = \bigcap_{o \in obs(T)} \mathcal{G}(o)$.
- (b) $obs(GS)$ is a history.

Proof. (a) This is a direct consequence of Theorem 2.

(b) By (a), $\diamond = \diamond_{obs(GS)}$ and $\sqsubset = \sqsubset_{obs(GS)}$. Thus $obs(GS)^{(SRI)} \subseteq obs(GS)$. It follows $obs(GS)^{(SRI)} = obs(GS)$, i.e. $obs(GS)$ is a history. \square

The main result of this last part now follows readily.

Theorem 3. *Let Δ be a report set consisting of initially finite stratified orders. Then the following are equivalent.*

(a) (a) *There is a $GS \in \mathcal{GS}_{IF}$ such that $\Delta = obs(GS)$.*

(b) (b) *Δ is a history.*

Proof. (a) \implies (b) This follows directly from (b) of Proposition 3

(b) \implies (a) This is a direct consequence of Propositions 1, 2 and (a) of Proposition 3. \square

Theorem 3 says that under the provision that observations are restricted to initially finite stratified orders (i.e. step sequences), every initially finite stratified GOS can be interpreted as a specification of a history conforming to π_1 . This is the key application result of this paper.

4 Conclusive Remarks

In this paper, we introduced the notion of Generalized Order Structure or GOS, proved that the class of initially finite stratified GOS is extension complete. The result is then used to give an axiomatic representation of general concurrent behaviours or histories (histories conforming to the most general paradigm π_1), when observations are restricted to initially finite stratified orders.

A natural possible continuation of the work here is to study the general case when observations are modelled by interval orders or just general partial orders. Indeed our original attempt was to prove the result for the general case. However, there were some technical problems concerning the proof analogues of Lemmas 3, 5 and 6 for interval and general GOS. We believe that the GOS approach will work, but it may require some modification of the definition of GOS in order to solve the problem in the interval and general case.

An immediate application of the obtained results seems to be in the Synthesis Problem area. We believe that the approach introduced in [15] could now, after employing the results of this paper, handle the cases like the most left case from Figure 1.

References

1. U. Abraham, S. Ben-David, M. Magodor, On global-time and inter-process communication, in *Semantics for Concurrency*, Workshops in Computing, Springer 1990, 311-323. 178
2. P. Baldan, N. Busi, A. Corradini, M. Pinna, Functorial Concurrent Semantics for Petri Nets with Read and Inhibitor Arcs, *Lecture Notes in Computer Science* 1877, Springer 2000, 115-127. 178
3. E. Best, F. de Boer, C. Palamedissi, Partial Order and SOS Semantics for Linear Constraint Programs, *Lecture Notes in Computer Science* 1282, Springer 1997, 256-273. 178
4. P. C. Fishburn, Intransitive indifference with unequal indifference intervals, *J. Math. Psych.* 7 (1970) 144-149. 180
5. H. Gaifman, V. Pratt, Partial order models of concurrency and the computation of functions, *Proc. of LICS'87*, 72-85. 178, 179, 183, 184
6. R. Janicki and M. Koutny, Invariants and Paradigms of Concurrency Theory. Proc. of PARLE'91, *Lecture Notes in Computer Science* 506(1991), 59-74. 178
7. R. Janicki and M. Koutny, Structure of Concurrency, *Theoretical Computer Science* 112 (1993), 5-52. 178, 179, 181, 182
8. R. Janicki and M. Koutny, Semantics of Inhibitor Nets, *Information and Computation*, 123, 1(1995), 1-16. 178
9. R. Janicki and M. Koutny, Fundamentals of modelling concurrency using discrete relational structures, *Acta Informatica*, 34, 367-388, 1997. 178, 179, 181, 183, 184, 186, 189
10. R. Janicki and M. Koutny, On Causality Semantics of Nets with Priorities, *Fundamenta Informaticae* 38(1999), 222-255. 178
11. J. Kleijn, M. Koutny, Process Semantics of P/T-Nets with Inhibitor Arcs, *Lecture Notes in Computer Science* 1825, Springer 2000, 261-281. 178
12. H. Kludel, F. Pommereau, A Class of Composable and Preemptible High-Level Petri Nets with an Application to a Multi-Tasking System, *Fundamenta Informaticae*, to appear. 178
13. L. Lamport, The mutual exclusion problem: Part I - a theory of interprocess communication; Part II - statements and solutions, *Journal of ACM* 33,2 (1986) 313-326. 178, 179, 183, 184
14. L. Lamport, What It Means for a Concurrent Programm to Satisfy a Specification: Why No One Has Specified Priority, *Proc. 12th ACM Symp. on Programming Languages*, 1985, 78-83. 178
15. M. Pietkiewicz-Koutny, The Synthesis Problem for Elementary Net Systems, *Fundamenta Informaticae* 40,2,3 (1999) 310-327. 178, 179, 190
16. E. Szpilrajn, Sur l'extension de l'ordre partial, *Fundamenta Mathematicae* 16(1930), 386-389. 179, 182
17. W. Vogler, Timed Testing of Concurrent Systems, *Information and Computation* 121(1995), 149-171. 178
18. W. Vogler, Partial Order Semantics and Inhibitor Arcs, *Lecture Notes in Computer Science* 1295, Springer 1997, 508-517. 178
19. R. Wollowski, J. Beister, Precise Petri Net Modelling of Critical Races in Asynchronous Arbiters and Synchronizers, *Proc. 1st Workshop on Hardware Design and Petri Nets*, Lisbon 1998, 46-65. 178
20. R. Wollowski, J. Beister, Comprehensive Causal Specification of Asynchronous Controller and Arbiter Behaviour, in A. Yakovlev, L. Gomes, L. Lavagno (eds.) *Hardware Design and Petri Nets*, Kluwer 2000. 178

An Algebra of Non-safe Petri Boxes

Raymond Devillers¹, Hanna Klaudel², Maciej Koutny³, and Franck Pommereau²

¹ Département d'Informatique, Université Libre de Bruxelles
B-1050 Bruxelles, Belgium rdevil@ulb.ac.be

² Université Paris 12, LACL
61 avenue du général de Gaulle, 94010 Créteil, France
{[klaudel,pommereau](mailto:klaudel,pommereau@univ-paris12.fr)}@univ-paris12.fr

³ Department of Computing Science, University of Newcastle upon Tyne
NE1 7RU, United Kingdom
Maciej.Koutny@newcastle.ac.uk

Abstract. We define an algebraic framework based on non-safe Petri nets, which allows one to express operations such as iteration, parallel composition, and transition synchronisation. This leads to an algebra of process expressions, whose constants and operators directly correspond to those used in Petri nets, and so we are able to associate nets to process expressions compositionally. The semantics of composite nets is then used to guide the definition of a structured operational semantics of process expressions. The main result is that an expression and the corresponding net generate isomorphic transition systems. We finally discuss a partial order semantics of the two algebras developed in this paper.

Keywords: Petri nets, process algebra, operational semantics.

1 Introduction

To relate process algebras, such as CCS [17] or CSP [13], and Petri nets [20], the approaches proposed in the literature often aim at providing a Petri net semantics to an existing process algebra as, *e.g.*, in [5,6,10,11,12,18]. Another way is to translate elements from Petri nets into process algebras as, *e.g.*, in [1].

A specific framework we are concerned with here is the *Petri Net Algebra* (PNA [4]) and its precursor, *Petri Box Calculus* (PBC [3]). This framework is composed of an algebra of process expressions (called *box expressions*) together with a fully compositional translation into labelled safe Petri nets (called *boxes*). (Recall that in a safe Petri net no place ever holds more than one token.)

The variant of the safe box algebra relevant to this paper comprises: *sequence* $E; F$ (the execution of E is followed by that of F); *choice* $E \square F$ (either E or F can be executed); *parallel composition* $E \parallel F$ (E and F can be executed concurrently); *iteration* $E \otimes F$ (E can be executed an arbitrary number of times, and then be followed by F); and *scoping* $E \text{ sc } a$ (all handshake synchronisations involving pairs of a - and \hat{a} -labelled transitions are enforced). Consider, as an example, three processes modelling a critical section and two user processes:

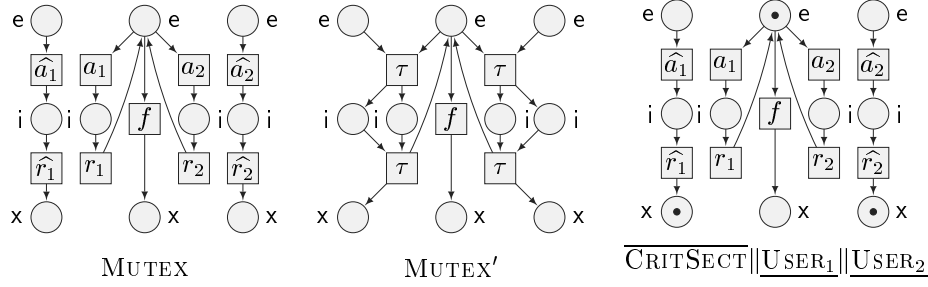


Fig. 1. Boxes and corresponding box expressions

$\text{CRITSECT} \stackrel{\text{df}}{=} ((a_1; r_1) \square (a_2; r_2)) \otimes f$, $\text{USER}_1 \stackrel{\text{df}}{=} \widehat{a}_1; \widehat{r}_1$ and $\text{USER}_2 \stackrel{\text{df}}{=} \widehat{a}_2; \widehat{r}_2$. The atomic actions a_1 and a_2 (together with the matching \widehat{a}_1 and \widehat{a}_2) model getting the access to a shared resource, r_1 and r_2 (together with \widehat{r}_1 and \widehat{r}_2) model its release, and f models a final action. The box expression where these three processes operate in parallel is $\text{MUTEX} \stackrel{\text{df}}{=} \text{CRITSECT} || \text{USER}_1 || \text{USER}_2$ (operators like parallel composition associate to the right), and the corresponding box is shown on the left of figure 1. In a box, places are labelled by their status (e for entry, x for exit and i for internal) while transitions are labelled by CCS-like communication actions; such as a_1 , \widehat{a}_1 and τ (as in CCS, τ is an internal action).

Though the box of MUTEX specifies the three constituent processes, it does not allow for interprocess communication. This can be achieved by applying the scoping w.r.t. the synchronisation actions a_i and r_i , which results in $\text{MUTEX}' \stackrel{\text{df}}{=} \text{MUTEX} \text{ sc } a_1 \text{ sc } a_2 \text{ sc } r_1 \text{ sc } r_2$, with the corresponding box shown in figure 1.

The operational semantics of box expressions is given through SOS rules in Plotkin's style [19]. However, instead of rules like $a.E \xrightarrow{a} E$ in CCS, the current state of an evolution is represented using overbars and underbars, marking respectively the initial and final states of (sub)expressions. E.g., in figure 1, the box on the right represents MUTEX after the two user processes have terminated, and the critical section is still in its initial state. There are two kinds of SOS rules: structural rules specify when distinct expressions denote the same state, e.g., one can deduce that $\overline{\text{CRITSECT}} || \overline{\text{USER}_1} || \overline{\text{USER}_2} \equiv \overline{\text{CRITSECT}} || \overline{\text{USER}_1} || \overline{\text{USER}_2} \equiv \overline{\text{CRITSECT}} || \overline{\text{USER}_1} || \overline{\text{USER}_2}$, while evolution rules specify when we may have a state change due to the execution of some of the actions, e.g., one can deduce that $((a_1; r_1) \square (a_2; r_2)) \otimes \overline{f} \xrightarrow{\{f\}} ((a_1; r_1) \square (a_2; r_2)) \otimes \underline{f}$. The two algebras of PNA are fully compatible, in the sense that a box expression and the corresponding box generate isomorphic transition systems. It will be our goal here to retain this property in a more expressive framework based on *non-safe* boxes.

Recently, [7,15] introduced a novel feature into the above model, aimed at the modelling of asynchronous communication (used, e.g., to model time-dependent or preemptive concurrent systems [15,16]). Consider the following process expressions modelling a producer and consumer processes (each can perform exactly *one* action, after which it terminates): $\text{PRODONE} \stackrel{\text{df}}{=} pb^+$ and $\text{CONSONE} \stackrel{\text{df}}{=} cb^-$. The pb^+ is an atomic action whose role is to 'produce' a token (resource) in a

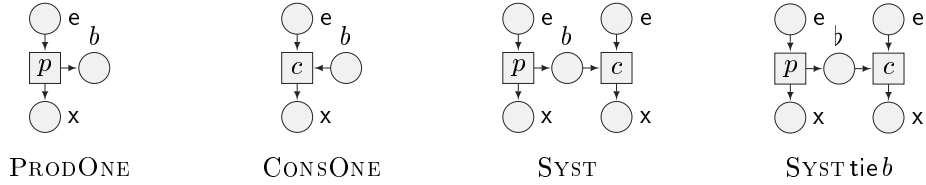


Fig. 2. Asynchronous communication

buffer identified by b ; in doing so, it generates the visible label p . The cb^- is an atomic action which ‘consumes’ a resource from buffer b , generating the visible label c . The boxes of these processes are shown in figure 2, where the buffer places are identified by the b labels.

An intuitive meaning of a b -labelled place is that some transitions can insert tokens into it, while other transitions can later remove them. This gives rise to asynchronous communication as, *e.g.*, in the parallel composition of the producer and consumer processes, $\text{SYST} \stackrel{\text{df}}{=} \text{PRODONE} \parallel \text{CONSONE}$, and the corresponding box in figure 2, where the two b -labelled buffer places are merged into a single b -labelled place (this place can later be merged with other b -labelled buffer places). An abstraction mechanism for asynchronous communication comes in the form of the *buffer restriction* operator, $\text{tie } b$, which changes the b -labelled buffer place into a b -labelled one. Such a place can no longer be merged with other buffer places, and so b -labelled places may be viewed as internal places. This is illustrated in figure 2 for $\text{SYSTtie}b$. The resulting model is no longer based on safe Petri nets since the buffer places are in general non-safe.

In this paper, we develop further the above approach, introducing the *Asynchronous Box Calculus* (or ABC) model. In particular, we no longer enforce the safeness of the non-buffer places, which may be undesirable for practical applications since this leads to the imposition of awkward syntactic constraints (see [4]). Safeness was needed in the original PNA to support a simple concurrency semantics of boxes and expressions based on causal partial orders. In this paper, it is replaced by *auto-concurrency freeness* which is a property guaranteeing concurrency semantics in terms of event structures as shown in [14], and so is highly relevant from a theoretical point of view. Moreover, we argue that in the case of ABC without buffer places (but still with non-safe places), one can retain the simple causal partial order semantics.

In short, ABC will comprise an algebra of non-safe boxes, and an algebra of box expressions. The two algebras will be related through a mapping which, for a box expression, returns a corresponding box with an isomorphic transition system. All the proofs can be found in the technical report [8].

Two examples. Consider three process expressions: $\text{PROD} \stackrel{\text{df}}{=} \text{PRODONE} \otimes f$, $\text{PRODPAR} \stackrel{\text{df}}{=} (\text{PRODONE} \parallel \text{PRODONE}) \otimes f$ and $\text{CONS} \stackrel{\text{df}}{=} \text{CONSONE} \otimes f$. When operating in parallel, PROD can repeatedly send a token to a b -labelled buffer place, which can then be repeatedly removed by the CONS process.

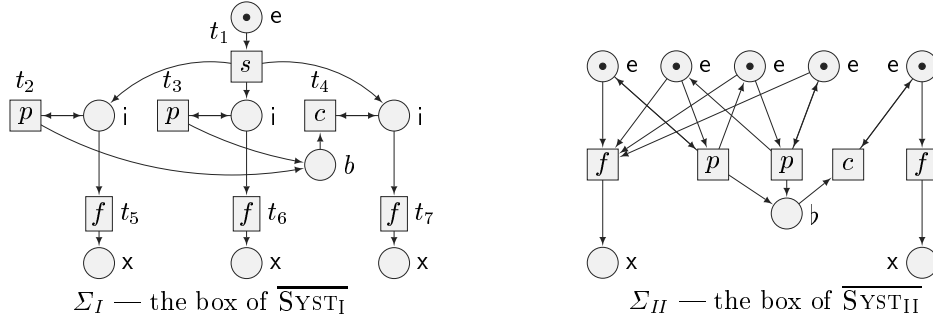


Fig. 3. Boxes for the two execution scenarios (double-headed arrows are self-loops)

The first example, $\text{SYST}_I \stackrel{\text{df}}{=} s; (\text{PROD} \parallel \text{PROD} \parallel \text{CONS})$ on the left of figure 3, models a system composed of two producers and one consumer operating in parallel and preceded by a ‘startup’ action s . The example $\text{SYST}_{II} \stackrel{\text{df}}{=} (\text{PRODPAR} \parallel \text{CONS ONE})$ tie b , shown on the right of figure 3, illustrates the encapsulating feature of buffer restriction which makes the buffer place b -labelled, and so no longer available for future merging. This example also shows that even if we disregard the buffer place, the resulting box is still non-safe as executing either of the p -labelled transitions adds a second token to one of the e -labelled places. Hence SYST_{II} cannot be handled by the preliminary version of ABC presented in [7].

The operational semantics is illustrated using the following two execution scenarios (in each case the system starts from its implicit initial state, *e.g.*, we consider $\overline{\text{SYST}_I}$ rather than SYST_I). In the first scenario, we consider the net Σ_I shown in figure 3 and the following evolution:

- the system is started up by executing the s -labelled transition;
- the two producers send a token each to the b -labelled (buffer) place;
- the consumer takes one of the two tokens from the buffer place and, at the same time, the first producer sends there another;
- the two producers and the consumer finish their operation by simultaneously executing the three f -labelled transitions.

This corresponds to $\Sigma_I [\{t_1\}\{t_2, t_3\}\{t_2, t_4\}\{t_5, t_6, t_7\}] \Sigma'_I$, which is a step sequence such that Σ'_I is Σ_I with two tokens in the buffer place, one token in each of the x -labelled places, and no token elsewhere. Since each x -labelled place has exactly one token, we consider Σ'_I to be in a *final* marking (or state). In terms of labelled step sequences, we have $\Sigma_I [\{s\}\{p, p\}\{p, c\}\{f, f, f\}] \Sigma'_I$.

In the second scenario, we only consider labelled steps, for the net Σ_{II} in figure 3, and the following evolution:

- the system begins by executing the left p -labelled transition, which puts a token in the buffer place, and another one in the third e -labelled place;

- the consumer takes the token from the buffer place and the right p -labelled transition puts another token in the buffer place;
- the system finishes by executing the two f -labelled transitions.

Such a scenario corresponds to $\Sigma_{II} [\{p\}\{c\}\{f, f\}] \Sigma'_{II}$, which is a labelled step sequence such that Σ'_{II} is Σ_{II} with one token in the buffer place and each of the x -labelled places, and no token elsewhere. Notice that Σ'_{II} is also in a final state.

Basic notations and definitions. Throughout the paper, we use the standard Petri net notions (see, *e.g.*, [20]) and (multi)set notation. In particular, $+$ and \cdot denote multiset addition and multiplication by a natural number, $\mathbf{mult}(X)$ comprises all finite multisets over a set X , and \leq is used to compare multisets. A subset of a set X may be treated as a multiset over X , through its characteristic function, and a singleton set can be identified with its sole element. The specific Petri net framework is sketched below.

We assume that there is a set \mathbb{A}_τ of (atomic) *actions* representing synchronous interface activities used, in particular, to model handshake communication. Similarly as in CCS, $\mathbb{A}_\tau \stackrel{\text{df}}{=} \mathbb{A} \uplus \{\tau\}$ and, for every $a \in \mathbb{A}$, \hat{a} is an action in \mathbb{A} such that $\hat{\hat{a}} = a$. There is also a finite set \mathbb{B} of *buffer symbols* for asynchronous communication.

A (*marked*) *labelled net* is here a tuple $\Sigma \stackrel{\text{df}}{=} (S, T, W, \lambda, M)$ such that: S and T are finite disjoint sets of respectively *places* and *transitions*; W is a *weight function* from the set $(S \times T) \cup (T \times S)$ to \mathbb{N} ; λ is a *labelling* for places and transitions such that for every place $s \in S$, $\lambda(s)$ is a symbol in $\{e, i, x, b\} \uplus \mathbb{B}$, and for every transition $t \in T$, $\lambda(t)$ is an action in \mathbb{A}_τ ; and M is a *marking*, *i.e.*, a multiset over S .

If the labelling of a place s is e , i or x , then s is an *entry*, *internal* or *exit* place, respectively. If the labelling is b then s is a *closed buffer* place, and if it is $b \in \mathbb{B}$, then s is an *open buffer* place. Collectively, the e -, i - and x -labelled places are called *control (flow)* places. Moreover, the set of all entry (resp. exit) places will be denoted by ${}^\circ\Sigma$ (resp. Σ°). To avoid ambiguity, we will sometimes decorate the various components of Σ with the index Σ and, to simplify diagrams, omit isolated unmarked buffer places.

For every place (transition) x , we use $\bullet x$ to denote its pre-set, *i.e.*, the set of all transitions (places) y such that there is an arc from y to x , that is, $W(y, x) > 0$. The post-set x^\bullet is defined in a similar way.

For a marking M of a labelled net Σ , we use M^{ctr} to denote M restricted to the control places. Then we say that M is *clean* if ${}^\circ\Sigma \leq M^{ctr} \Rightarrow M^{ctr} = {}^\circ\Sigma$ and $\Sigma^\circ \leq M^{ctr} \Rightarrow M^{ctr} = \Sigma^\circ$. Moreover, M is *ac-free* if, for every $t \in T$, there is a control place $s \in \bullet t$ such that $M(s) < 2 \cdot W_\Sigma(s, t)$, *i.e.*, the marking of s does not allow *auto-concurrency* of t .

2 An Algebra of Boxes

To model concurrent systems, we use a class of labelled nets called *asynchronous boxes*. To model operations on such nets, we use *operator boxes* (a particular

kind of labelled nets where all transitions are intended to be substituted by asynchronous boxes) and the *net substitution* meta-operator (called also refinement [4]), which allows one to realise this substitution.

An (*asynchronous*) *box* is a labelled net Σ such that each transition is labelled by an action in \mathbb{A}_τ , and the net is:

- *ex-restricted*: there is at least one entry place and at least one exit place;
- \mathbb{B} -*restricted*: for every $b \in \mathbb{B}$, there is exactly one b -labelled place;
- *control-restricted*: for every transition t there is at least one control place in $\bullet t$, and at least one control place in t^\bullet .

The execution semantics of Σ is based on finite steps, which capture the potential concurrency in its behaviour. A step is a finite multiset of transitions U and, when enabled, it can be executed leading to a new net Θ ; we denote this by $\Sigma[U]\Theta$ or $\Theta \in [\Sigma]$. Transition labelling may be extended to steps, leading to labelled steps of the form $\Sigma[\Gamma]_\lambda \Sigma'$ which means that there is a step U such that $\Sigma[U] \Sigma'$ and $\Gamma = \lambda(U)$. Although we will use the same term ‘step’ to refer both to a finite multiset of transitions and to a finite multiset of labels, it will always be clear from the context which one is meant.

A box Σ is *static* (resp. *dynamic*) if $M_\Sigma^{ctr} = \emptyset$ (resp. $M_\Sigma^{ctr} \neq \emptyset$) and all the markings reachable from M_Σ^{ctr} , ${}^\circ\Sigma$ or Σ° in the box Σ^{ctr} are both clean and ac-free, where Σ^{ctr} is Σ with all its buffer places and adjacent arcs removed. The asynchronous boxes, static boxes and dynamic boxes will respectively be denoted by **abox**, **abox^{stc}** and **abox^{dyn}**. Static boxes do not admit non-empty steps, and if Σ is a dynamic box, then U is a *set* of transitions (but if $\Sigma[\Gamma]_\lambda \Sigma'$, then the labelled step Γ may be a true *multiset* of actions, as illustrated in the execution scenarios in the introduction) and Σ' is a dynamic box.

The upper row in figure 4 shows four kinds of static boxes Σ_α used in ABC, where $\alpha \in \mathcal{A} \stackrel{\text{df}}{=} \{a, ab^+, ab^-, ab^\pm \mid a \in \mathbb{A}_\tau \wedge b \in \mathbb{B}\}$. They are the basic building blocks, from which other static and dynamic boxes are constructed.

The complete behaviour of a static or dynamic box can be represented by a transition system. And, since we have two kinds of possible steps, we introduce two kinds of transition systems. The *full transition system* of a dynamic box Σ is $\text{fts}_\Sigma \stackrel{\text{df}}{=} (V, L, A, \text{init})$ where $V \stackrel{\text{df}}{=} [\Sigma]$ are the states; $L \stackrel{\text{df}}{=} 2^{T_\Sigma}$ are arc labels; $A \stackrel{\text{df}}{=} \{(\Sigma', U, \Sigma'') \in V \times L \times V \mid \Sigma'[U] \Sigma''\}$ is the set of arcs; and $\text{init} \stackrel{\text{df}}{=} \Sigma$ is the initial state. For a static box Σ , $\text{fts}_\Sigma \stackrel{\text{df}}{=} \text{fts}_{\overline{\Sigma}}$. The *labelled transition system* of a static or dynamic box Σ , denoted by lts_Σ , is defined as fts_Σ with each arc label U changed to $\lambda_\Sigma(U)$ (note that different arcs between two states in fts_Σ may give rise to a single arc in lts_Σ).

The presentation of the operators of ABC starts with *marking operators*, which modify the marking of a box Σ (below $b \in \mathbb{B}$ and $B \in \text{mult}(\mathbb{B})$):

- $\Sigma.B$ adds $B(b)$ tokens to the b -labelled open buffer place of Σ , for each $b \in \mathbb{B}$; moreover, $\Sigma.b \stackrel{\text{df}}{=} \Sigma.\{b\}$ (this operation is called *buffer stuffing*);
- $\overline{\Sigma}$ (resp. $\underline{\Sigma}$) is Σ with one additional token in each entry (resp. exit) place, *i.e.*, $M_{\overline{\Sigma}} \stackrel{\text{df}}{=} M_\Sigma + {}^\circ\Sigma$ (resp. $M_{\underline{\Sigma}} \stackrel{\text{df}}{=} M_\Sigma + \Sigma^\circ$);
- $\llbracket \Sigma \rrbracket$ is Σ with the empty marking.

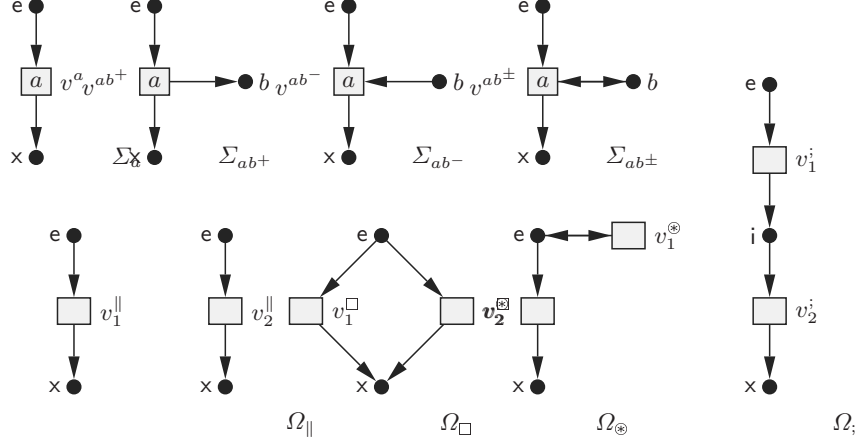


Fig. 4. Asynchronous and operator boxes of ABC, where $a \in \mathbb{A}_\tau$ and $b \in \mathbb{B}$

For the remaining operators, the identities of transitions will play a key role, especially when defining the SOS semantics of process expressions. More precisely, transition identities will come in the form of finite labelled trees retracing the operators used to construct a box.

We assume that there is a set η of *basic* transition identities and a corresponding set of basic labelled trees with a single node labelled with an element of η . All the transitions in figure 4 are assumed to be of that kind. To express more complex (unordered) finite trees, or sets of trees, used as transition identities in nets obtained through net substitution, we will use the following linear notations (see figure 5 page 200 for an illustration):

- $v \triangleleft \mathbb{T}$, where $v \in \eta$ is a basic transition identity and \mathbb{T} is a finite set of finite labelled trees, denotes a tree where the trees of the set \mathbb{T} are appended to a v -labelled root;
- $v \triangleleft \mathbb{t}$ denotes $v \triangleleft \{\mathbb{t}\}$ and $v \blacktriangleleft \mathbb{T}$ denotes the set of trees $\{v \triangleleft \mathbb{t} \mid \mathbb{t} \in \mathbb{T}\}$.

A similar naming discipline could be used for the places of the constructed nets following the scheme used in [4]. However, since place trees were essentially needed for the definition of recursion which is not considered in the present paper, we will not use them. Instead, when applying net substitution, we will assume that the place sets of the operands are renamed to make them disjoint. We then construct new places by aggregating the existing ones; *e.g.*, if s_1 and s_2 are places from some boxes, (s_1, s_2) may be the identity of a newly constructed place. We start with the two unary operations (we define them directly, rather than through net substitution).

Scoping $\Sigma \text{sc } a$. Parameterised by a communication action $a \in \mathbb{A}$ and with the domain of application $\text{dom}_{\text{sc } a} \stackrel{\text{df}}{=} \text{abox}^{\text{stc}} \cup \text{abox}^{\text{dyn}}$, $\Sigma \text{sc } a$ is a labelled net which

is like Σ with the only difference that the set of transitions comprises all trees $w \stackrel{\text{df}}{=} v^{\text{sc}a} \triangleleft \{t, u\}$ with $t, u \in T$ such that $\{\lambda(t), \lambda(u)\} = \{a, \widehat{a}\}$, as well as all trees $z \stackrel{\text{df}}{=} v^{\text{sc}a} \triangleleft r$ with $r \in T$ such that $\lambda(r) \notin \{a, \widehat{a}\}$. The label of w is τ , and that of z is $\lambda(r)$; the weight function is given by $W_{\Sigma^{\text{sc}a}}(p, w) \stackrel{\text{df}}{=} W(p, t) + W(p, u)$ and $W_{\Sigma^{\text{sc}a}}(p, z) \stackrel{\text{df}}{=} W(p, r)$, and similarly for $W_{\Sigma^{\text{sc}a}}(w, p)$ and $W_{\Sigma^{\text{sc}a}}(z, p)$.

Buffer restriction $\Sigma^{\text{tie}b}$. Parameterised by a buffer $b \in \mathbb{B}$ and with the domain of application $\text{dom}_{\text{tie}b} \stackrel{\text{df}}{=} \text{abox}^{\text{stc}} \cup \text{abox}^{\text{dyn}}$, $\Sigma^{\text{tie}b}$ is like Σ with the only difference that the identity of each transition t is changed to $v^{\text{tie}b} \triangleleft t$, the label of the only b -labelled place is changed to b , and a new unmarked unconnected b -labelled place is added.

Choice $\Sigma_1 \square \Sigma_2$, iteration $\Sigma_1 \otimes \Sigma_2$, sequence $\Sigma_1 ; \Sigma_2$, and parallel composition $\Sigma_1 \parallel \Sigma_2$. We consider here a binary operator $\text{box } \Omega_{\text{bin}} \in \{\Omega_{\square}, \Omega_{\otimes}, \Omega_{;}, \Omega_{\parallel}\}$, as shown in figure 4 (the labels of the transitions will be irrelevant here), and its application $\Sigma_1 \text{ bin } \Sigma_2$. The first three binary operator boxes specify different ways to sequentially compose behaviours and have the domain of application $\text{dom}_{\text{bin}} \stackrel{\text{df}}{=} (\text{abox}^{\text{stc}})^2 \cup (\text{abox}^{\text{dyn}} \times \text{abox}^{\text{stc}}) \cup (\text{abox}^{\text{stc}} \times \text{abox}^{\text{dyn}})$. The last one has the domain of application $\text{dom}_{\parallel} \stackrel{\text{df}}{=} (\text{abox}^{\text{stc}})^2 \cup (\text{abox}^{\text{dyn}})^2$, *i.e.*, its components may evolve concurrently. The net $\Phi = \Sigma_1 \text{ bin } \Sigma_2$ is defined as follows.

The transitions of Φ are the set of all trees $v_i^{\text{bin}} \triangleleft t$ (with $t \in T_{\Sigma_i}$ and $i = 1, 2$). The label of each $v_i^{\text{bin}} \triangleleft t$ is that of t . Each i -labelled or b -labelled place $p \in S_{\Sigma_j}$ belongs to S_{Φ} . Its label and marking are unchanged and, for every transition $v_i^{\text{bin}} \triangleleft t$, the weight function is given by $W_{\Phi}(p, v_i^{\text{bin}} \triangleleft t) \stackrel{\text{df}}{=} W_{\Sigma_i}(p, t)$ if $j = i$ and $W_{\Phi}(p, v_i^{\text{bin}} \triangleleft t) \stackrel{\text{df}}{=} 0$ otherwise, and similarly for $W_{\Phi}(v_i^{\text{bin}} \triangleleft t, p)$.

For every place $s \in S_{\Omega_{\text{bin}}}$ with $\bullet s = \{v_{l_1}^{\text{bin}}, \dots, v_{l_k}^{\text{bin}}\}$ and $s^{\bullet} = \{v_{j_1}^{\text{bin}}, \dots, v_{j_m}^{\text{bin}}\}$ ($k, m \in \{0, 1, 2\}$), we construct in S_{Φ} all the places of the form $p \stackrel{\text{df}}{=} (x_1, \dots, x_k, e_1, \dots, e_m)$, where each x_r (if any) is an exit place of Σ_{l_r} , and each e_q (if any) is an entry place of Σ_{j_q} . The label of p is that of s , its marking is the sum of the markings of $x_1, \dots, x_k, e_1, \dots, e_m$, and for every transition $v_i^{\text{bin}} \triangleleft t$, the weight function is given by:

$$W_{\Phi}(p, v_i^{\text{bin}} \triangleleft t) \stackrel{\text{df}}{=} \begin{cases} W_{\Sigma_i}(x_r, t) + W_{\Sigma_i}(e_q, t) & \text{if } v_i^{\text{bin}} \in \bullet s \cap s^{\bullet} \text{ and } i = l_r = j_q \\ W_{\Sigma_i}(x_r, t) & \text{if } v_i^{\text{bin}} \in \bullet s \setminus s^{\bullet} \text{ and } i = l_r \\ W_{\Sigma_i}(e_q, t) & \text{if } v_i^{\text{bin}} \in s^{\bullet} \setminus \bullet s \text{ and } i = j_q \\ 0 & \text{otherwise,} \end{cases}$$

and similarly for $W_{\Phi}(v_i^{\text{bin}} \triangleleft t, p)$.

For every $b \in \mathbb{B}$, there is a unique b -labelled place $p^b \stackrel{\text{df}}{=} (p_1^b, p_2^b) \in S_{\Phi}$ which merges the two b -labelled places, p_1^b and p_2^b , of the two operands. The marking of p^b is the sum of the markings of p_1^b and p_2^b , and for each transition $v_i^{\text{bin}} \triangleleft t$, the weight function is given by $W_{\Phi}(p^b, v_i^{\text{bin}} \triangleleft t) \stackrel{\text{df}}{=} W_{\Sigma_i}(p_i^b, t)$, and similarly for $W_{\Phi}(w \triangleleft t, p^b)$.

Properties. The net operations of ABC (other than $\overline{\Sigma}$ and $\underline{\Sigma}$) always return a syntactically valid object, *i.e.*, a box with a clean and ac-free marking. If one

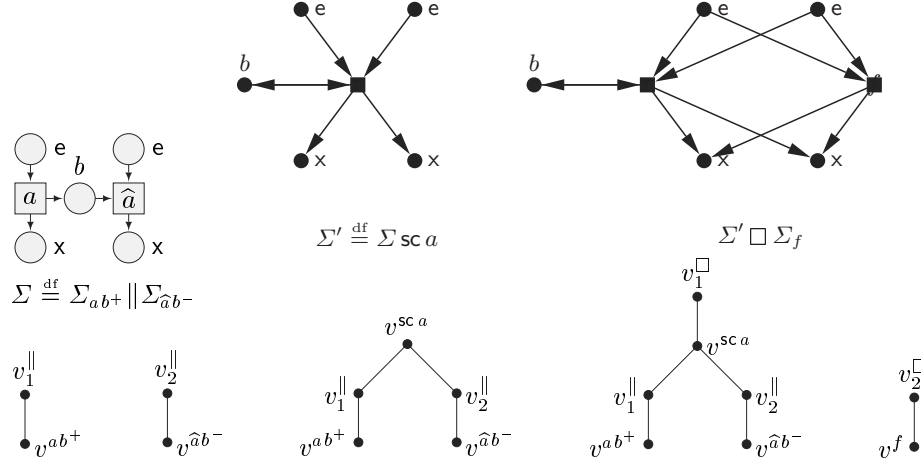


Fig. 5. The trees give the identities of the transitions (from left to right) of the boxes in the upper row; *e.g.*, the fourth tree is $v_1^\square \triangleleft v^{sc a} \triangleleft \{v_1^\parallel \triangleleft v^{ab^+}, v_2^\parallel \triangleleft v^{\widehat{a}b^-}\}$

makes no use of buffer stuffing nor buffer restriction nor basic nets other than Σ_a , one gets net operations similar to those defined in the standard box algebra (see [3,4]), except for the additional b -labelled places which are all isolated and unmarked.

The net operations are illustrated in figure 5, where explicit transition identities are shown for various stages of the construction, from the basic net and transition identities shown in figure 4.

Relating structure and behaviour. We now investigate how the behaviour of composite boxes depends on the behaviours of the boxes being composed. We provide full details for sequential composition (other operators are treated in [8]).

First, we capture situations where different application of sequential composition lead to the same result. For two pairs of boxes, $\Sigma \stackrel{\text{df}}{=} (\Sigma_1, \Sigma_2)$ and $\Theta \stackrel{\text{df}}{=} (\Theta_1, \Theta_2)$, we define:

- $\Sigma \equiv' \Theta$ if there are boxes Ψ_1 and Ψ_2 such that $\{\Sigma, \Theta\} = \{(\Psi_1, \Psi_2), (\Psi_1, \overline{\Psi_2})\}$. In other words, if the first operand has reached a final state, then this is equivalent to the second operand being in an initial state;
- $\Sigma \equiv'' \Theta$ if there are boxes Ψ_1 and Ψ_2 and $B_1, B_2, B'_1, B'_2 \in \text{mult}(\mathbb{B})$ such that $B_1 + B_2 = B'_1 + B'_2$, $\Sigma = (\Psi_1.B_1, \Psi_2.B_2)$ and $\Theta = (\Psi_1.B'_1, \Psi_2.B'_2)$. In other words, Σ and Θ are the same except perhaps the distribution of tokens in open buffer places corresponding to the same b but coming from different components (buffer stuffing never changes the marking of closed buffer places). The sequence operator will glue the corresponding open buffer places thus merging their markings.

We then define \equiv_Ω to be $(\equiv' \cup id_{\text{abox}}) \circ \equiv''$, where id_{abox} is the identity on abox .

Relations $\equiv_{\Omega_{\text{bin}}}$ like that above are defined for all binary operators of ABC. It may be shown that, if $\Sigma \equiv_{\Omega_{\text{bin}}} \Theta$, then $\Sigma \in \text{dom}_{\text{bin}} \Rightarrow \Theta \in \text{dom}_{\text{bin}}$ and $\llbracket \Sigma_i \rrbracket = \llbracket \Theta_i \rrbracket$ ($i = 1, 2$); and if $\llbracket \Sigma_i \rrbracket = \llbracket \Theta_i \rrbracket$ ($i = 1, 2$), then $\Sigma_1 \text{ bin } \Sigma_2 = \Theta_1 \text{ bin } \Theta_2$ iff $\Sigma \equiv_{\Omega_{\text{bin}}} \Theta$. Hence, when restricted to dom_{bin} , $\equiv_{\Omega_{\text{bin}}}$ is an equivalence relation which identifies the tuples of boxes which give rise to the same composite nets.

The next result captures the *behavioural compositionality* of our model, *i.e.*, the way the behaviours of composite nets (in terms of enabled steps) are related to the behaviours of the composed nets. Basically, we want to establish what steps are enabled by $\Sigma_1 \text{ bin } \Sigma_2$ knowing the steps enabled by Σ_1 and Σ_2 .

Theorem 1. *Let $\Sigma_1 \text{ bin } \Sigma_2$ be a valid application of a binary ABC operator bin.*

- *If $\Sigma_i [U_i] \Psi_i$ ($i = 1, 2$), then $\Sigma_1 \text{ bin } \Sigma_2 [V] \Psi_1 \text{ bin } \Psi_2$, where $V = (v_1^{\text{bin}} \blacktriangleleft U_1) \uplus (v_2^{\text{bin}} \blacktriangleleft U_2)$.*
- *If $\Sigma_1 \text{ bin } \Sigma_2 [V] \Delta$ then there are Θ, Ψ, U_1 and U_2 , such that: $\Theta \equiv_{\Omega_{\text{bin}}} \Sigma$, $\Theta_i [U_i] \Psi_i$ ($i = 1, 2$), $\Psi_1 \text{ bin } \Psi_2 = \Delta$ and V is as above.*

Various important consequences may be derived from the result presented above and a similar result which holds for the unary operators; *e.g.*, the way static and dynamic boxes are composed guarantees that the result is a static or dynamic box when the domain of application of the operators is respected, *i.e.*, every composite net of ABC is a static or dynamic box.

3 An Algebra of Asynchronous Box Expressions

We consider an algebra of process expressions over the signature:

$$\mathcal{A} \cup \{ \overline{(\cdot)}, (\cdot) \} \cup \{ \parallel, ;, \square, \otimes \} \cup \{ \text{sc } a \mid a \in \mathbb{A} \} \cup \{ \text{tie } b, .b \mid b \in \mathbb{B} \},$$

where \mathcal{A} is the set of constants. The binary operators $\parallel, ;, \square$ and \otimes will be used in the infix mode; the unary operators $\text{sc } a, \text{tie } b$ and $.b$ will be used in the postfix mode; and $\overline{(\cdot)}$ and (\cdot) are two positional unary operators (the position of the argument being given by the dot).

There are two classes of process expressions corresponding respectively to the static and dynamic boxes: the *static* E and *dynamic* D expressions, denoted respectively by $\text{aexpr}^{\text{stc}}$ and $\text{aexpr}^{\text{dyn}}$. Collectively, we will refer to them as the (asynchronous) *box expressions*, aexpr . Their syntax is given by:

$$\begin{aligned} E &::= \alpha \mid E \text{ sc } a \mid E \text{ tie } b \mid E.b \mid E \parallel E \mid E \text{ bin } E \\ D &::= \overline{E} \mid \underline{E} \mid D \text{ sc } a \mid D \text{ tie } b \mid D.b \mid D \parallel D \mid D \text{ bin } E \mid E \text{ bin } D \end{aligned}$$

where $\alpha \in \mathcal{A}, a \in \mathbb{A}, b \in \mathbb{B}$, and bin is a binary operator other than \parallel . In the following, we will use E or F to denote any static expression, J or K any dynamic expression, and G or H any static or dynamic expression. For an expression G , $\llbracket G \rrbracket$ is G with all occurrences of $\overline{(\cdot)}, (\cdot)$ and $.b$ removed.

Essentially, a box expression encodes the structure of a box, together with the current marking of the control places (with the bars) and the buffer places

Table 1. Structural similarity, and two operational semantics for ABC, where $a \in \mathbb{A}_\tau$, $c \in \mathbb{A}$, $b \neq b' \in \mathbb{B}$ and $\text{bin} \in \{\|, ;, \square, \otimes\}$

$\frac{G \equiv H}{G \text{ una} \equiv H \text{ una}}$	$\frac{G \equiv H, G' \equiv H'}{G \text{ bin } G' \equiv H \text{ bin } H'}$	$\frac{E \equiv F}{\overline{E} \equiv \overline{F}, \underline{E} \equiv \underline{F}}$
$\overline{E \square F} \equiv \overline{E} \square \overline{F}$	$\overline{E \square F} \equiv E \square \overline{F}$	$\underline{E} \square F \equiv \underline{E} \square F$
$E \square \underline{F} \equiv \underline{E} \square F$	$\overline{E \ F} \equiv \overline{E} \ \overline{F}$	$\underline{E} \ F \equiv \underline{E} \ F$
$\overline{E \otimes F} \equiv \overline{E} \otimes \overline{F}$	$\underline{E \otimes F} \equiv \underline{E} \otimes \underline{F}$	$\underline{E} \otimes F \equiv \underline{E} \otimes \overline{F}$
$E \otimes \underline{F} \equiv \underline{E} \otimes F$	$\overline{E; F} \equiv \overline{E}; F$	$\underline{E}; F \equiv \underline{E}; \overline{F}$
$E; \underline{F} \equiv \underline{E}; F$	$(G.b) \text{ bin } H \equiv (G \text{ bin } H).b$	$G \text{ bin } (H.b) \equiv (G \text{ bin } H).b$
$\overline{E \text{ una}} \equiv \overline{E} \text{ una}$	$(G.b) \text{ una}' \equiv (G \text{ una}').b$	$\underline{E} \text{ una} \equiv \underline{E} \text{ una}$
$\text{una} \in \{\text{sc } c, \text{tie } b, .b\} \quad \text{una}' \in \{\text{sc } c, \text{tie } b', .b'\}$		
$\frac{}{ab^\pm.b \xrightarrow{\{v^{ab^\pm}\}} \underline{ab^\pm}.b}$	$\frac{}{ab^+ \xrightarrow{\{v^{ab^+}\}} \underline{ab^+}.b}$	$\frac{}{ab^-.b \xrightarrow{\{v^{ab^-}\}} \underline{ab^-}}$
$\frac{}{\overline{a} \xrightarrow{\{v^a\}} \underline{a}}$		$\frac{}{\overline{a} \xrightarrow{\{a\}} \underline{a} (*)}$
$\frac{}{ab^-.b \xrightarrow{\{a\}} \underline{ab^-} (*)}$	$\frac{}{ab^\pm.b \xrightarrow{\{a\}} \underline{ab^\pm}.b (*)}$	$\frac{}{ab^+.b \xrightarrow{\{a\}} \underline{ab^+}.b (*)}$
$\frac{G \xrightarrow{U} H, G' \xrightarrow{U'} H'}{G \text{ bin } G' \xrightarrow{V} H \text{ bin } H'}$	$\frac{G \xrightarrow{U} H}{G \text{ tie } b \xrightarrow{v^{\text{tie } b} \blacktriangleleft U} H \text{ tie } b}$	$\frac{D \xrightarrow{\{t_1, u_1 \dots t_k, u_k, z_1 \dots z_l\}} D'}{D \text{ sc } c \xrightarrow{\{y_1, \dots, y_k, x_1 \dots x_l\}} D' \text{ sc } c}$
$V = (v_1^{\text{bin}} \blacktriangleleft U) \uplus (v_2^{\text{bin}} \blacktriangleleft U') \quad \begin{cases} \{\lambda(t_i), \lambda(u_i)\} = \{c, \hat{c}\} \\ \lambda(z_j) \notin \{c, \hat{c}\} \end{cases} \quad \begin{cases} y_i = v^{\text{sc } c} \triangleleft \{t_i, u_i\} \\ x_j = v^{\text{sc } c} \triangleleft z_j \end{cases}$		
$G \xrightarrow{\emptyset} G (\dagger)$	$\frac{G \equiv G' \xrightarrow{U} H' \equiv H}{G \xrightarrow{U} H} (\dagger)$	$\frac{G \xrightarrow{U} H}{G.b \xrightarrow{U} H.b} (\dagger)$
$\frac{G \xrightarrow{\Gamma} H}{G \text{ tie } b \xrightarrow{\Gamma} H \text{ tie } b} (*)$	$\frac{G \xrightarrow{\Gamma} H, G' \xrightarrow{\Gamma'} H'}{G \text{ bin } G' \xrightarrow{\Gamma + \Gamma'} H \text{ bin } H'} (*)$	$\frac{D \xrightarrow{\Gamma + k \cdot \{c, \hat{c}\}} D'}{D \text{ sc } c \xrightarrow{\Gamma + k \cdot \{\tau\}} D' \text{ sc } c} (*)$

(with the $.b$'s). Thus, a box expression \overline{E} represents E in its initial state (in terms of nets, this corresponds to the initially marked box of E). Similarly, \underline{E} represents E in final state. Note that the $.b$ notation is needed for static as well as for dynamic box expressions because a static part of a dynamic box expression may have $.b$'s which can be used by the active part.

The denotational semantics of box expressions, $\text{box} : \text{aexpr} \rightarrow \text{abox}$, is given compositionally. Assuming that $\alpha \in \mathcal{A}$, $b \in \mathbb{B}$, una stands for an unary and bin for a binary operator of ABC, we have: $\text{box}(\alpha) \stackrel{\text{df}}{=} \Sigma_\alpha$, $\text{box}(\overline{E}) \stackrel{\text{df}}{=} \overline{\text{box}(E)}$, $\text{box}(\underline{E}) \stackrel{\text{df}}{=} \underline{\text{box}(E)}$, $\text{box}(G \text{ una}) \stackrel{\text{df}}{=} \text{box}(G) \text{ una}$ and $\text{box}(G \text{ bin } H) \stackrel{\text{df}}{=} \text{box}(G) \text{ bin } \text{box}(H)$. One can show that the semantical mapping always returns a static or dynamic box, and that $\text{box}(G)$ is static iff G is static. We now set out to define an operational semantics of box expressions.

A *structural similarity* relation on box expressions, denoted by \equiv , is defined as the least equivalence relation on box expressions such that all the equations in the upper part of table 1 are satisfied. The rules either directly follow those of the original PNA or capture the fact that an asynchronous message, produced by a ab^+ expression and represented by $.b$, can freely move within a box expression in order to be received by some action of the form ab^- . However, the $.b$ may never cross the boundary imposed by the tie b operator (notice that moving outside a buffer restriction context is only allowed if $b \neq b'$). It may be observed that the equivalence relation so defined is in fact a congruence for all the operators of the algebra. It is easy to see that the structural similarity relation is closed in the domain of expressions, in the sense that if a box expression matches one of the sides of any rule then the other side defines a legal box expression. Moreover, it captures the fact that box expressions have the same net translation, *i.e.*, $\text{box}(G) = \text{box}(H)$ iff $G \equiv H$, provided that $\llbracket G \rrbracket = \llbracket H \rrbracket$.

SOS semantics. In developing the operational semantics of ABC, we first introduce operational rules based on transitions of boxes which provide the denotational semantics of box expressions. Based on these, we formulate our key consistency result. Then we introduce the label based rules, together with the derived consistency results.

Consider the set \mathbb{T} of all transition trees in the boxes derived through the box mapping. The first operational semantics has moves of the form $G \xrightarrow{U} H$ such that G and H are box expressions and $U \in \mathbb{U}$, where \mathbb{U} is the set of all finite subsets of \mathbb{T} . The idea here is that U is a valid step for the boxes associated with G and H , *i.e.*, that $\text{box}(G)[U]\text{box}(H)$ holds. Note that each $t \in \mathbb{T}$ has always the same label in the boxes derived through the box mapping; such a label will be denoted by $\lambda(t)$.

Formally, we define a ternary relation \longrightarrow which is the least relation comprising all $(G, U, H) \in \text{aexpr} \times \mathbb{U} \times \text{aexpr}$ such that the relations in table 1 (middle and bottom parts) other than those marked by $(*)$ hold, where $G \xrightarrow{U} H$ denotes $(G, U, H) \in \longrightarrow$. In the rule for binary operators, we make no restriction on U and U' but the domain of application of bin ensures that this rule will always be used with the correct static/dynamic combination of boxes. *E.g.*, in the case of the choice operator, U or U' will always be empty.

Let D be a dynamic box expression. We will use $[D]$ to denote all the box expressions derivable from D , *i.e.*, the least set of expressions containing D such that if $J \in [D]$ and $J \xrightarrow{U} K$, for some $U \in \mathbb{U}$, then $K \in [D]$. Moreover, $[J]_{\equiv}$ will denote the equivalence class of \equiv containing J . The *full transition system* of D is $\text{fts}_D \stackrel{\text{df}}{=} (V, L, A, \text{init})$, where $V \stackrel{\text{df}}{=} \{[J]_{\equiv} \mid J \in [D]\}$ are the states; $L \stackrel{\text{df}}{=} \mathbb{U}$ are

arc labels; $A \stackrel{\text{df}}{=} \{([J]_{\equiv}, U, [K]_{\equiv}) \in V \times \mathbb{U} \times V \mid J \xrightarrow{U} K\}$ is the set of arcs; and $init \stackrel{\text{df}}{=} [D]_{\equiv}$ is the initial state. For a static box expression E , $\text{fts}_E \stackrel{\text{df}}{=} \text{fts}_{\overline{E}}$. Note that we base transition systems of box expressions on the equivalence classes of \equiv , rather than on box expressions themselves, since we may have $D \xrightarrow{\emptyset} J$ for two different expressions D and J , whereas in the domain of boxes, $\Sigma[\emptyset] \Theta$ always implies $\Sigma = \Theta$.

We now state a fundamental result which demonstrates that the operational and denotational semantics of a box expression capture the same step based operational behaviour, in arguably the strongest sense.

Theorem 2. *For every G , $\text{iso}_G \stackrel{\text{df}}{=} \{([H]_{\equiv}, \text{box}(H)) \mid [H]_{\equiv} \text{ is a node of } \text{fts}_G\}$ is an isomorphism between the full transition systems fts_G and $\text{fts}_{\text{box}(G)}$.*

To define a label based operational semantics of box expressions, we first retain the structural similarity relation \equiv on box expressions without any change. Next, we define moves of the form $G \xrightarrow{\Gamma} H$, where G and H are box expressions as before, and Γ is a finite multiset of labels in \mathbb{A}_{τ} . Referring to table 1, we keep the rules marked with (†) with \emptyset now denoting the empty multiset of labels, and U being changed to Γ , and add those marked with (*) (instead of the corresponding rules based on transitions).

The two types of operational semantics are clearly related; essentially, each label based move is a transition based move with only transitions labels being recorded. As a result, the properties concerning transition based operational semantics directly extend to the label based one. For a box expression G , the label based operational semantics is captured by its *labelled transition system*, denoted by lts_G , and defined as fts_G with each arc label U changed to $\lambda(U)$. We then obtain the following result.

Theorem 3. *For every G , $\text{iso}_G \stackrel{\text{df}}{=} \{([G]_{\equiv}, \text{box}(H)) \mid [H]_{\equiv} \text{ is a node of } \text{lts}_G\}$ is an isomorphism between the labelled transition systems lts_G and $\text{lts}_{\text{box}(G)}$.*

Hence ABC supports two consistent (in a very strong sense since the corresponding transition systems are isomorphic and not only bisimilar, as in [7]) concurrent semantics for a class of process expressions with both synchronous and asynchronous communication.

The rules of the label based operational semantics are put into work below, where we use the second scenario presented in the introduction:

$$\begin{aligned}
& \overline{((pb^+ \| pb^+) \otimes f) \| (cb^- \otimes f)} \text{ tie } b & \equiv & \overline{((\overline{pb^+} \| \overline{pb^+}) \otimes f) \| (cb^- \otimes f)} \text{ tie } b \\
& \xrightarrow{\{p\}} \overline{((\underline{pb^+}.b \| \overline{pb^+}) \otimes f) \| (cb^- \otimes f)} \text{ tie } b & \equiv & \overline{((\underline{pb^+} \| \overline{pb^+}) \otimes f) \| (cb^- . b \otimes f)} \text{ tie } b \\
& \xrightarrow{\{p,c\}} \overline{((\underline{pb^+} \| \underline{pb^+}.b) \otimes f) \| (cb^- \otimes f)} \text{ tie } b & \equiv & \overline{((pb^+ \| pb^+ . b) \otimes \overline{f}) \| (cb^- \otimes \overline{f})} \text{ tie } b \\
& \xrightarrow{\{f,f\}} \overline{((pb^+ \| pb^+ . b) \otimes \underline{f}) \| (cb^- \otimes \underline{f})} \text{ tie } b & \equiv & \overline{((pb^+ \| pb^+) \otimes f) \| (cb^- \otimes f)} \text{ tie } b
\end{aligned}$$

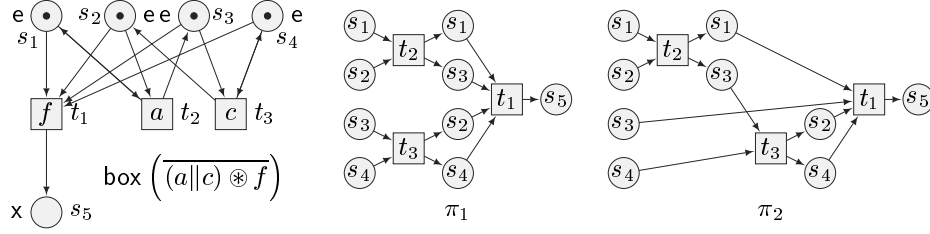


Fig. 6. A box and two of its processes

4 Causality in Boxes and Box Expressions

We now discuss causality in the fragment of the ABC model without buffer-specific construct. This still leads to the possibility of deriving boxes which are non-safe, unlike in PNA (and PBC). Consider, for example, the expression $D \stackrel{\text{df}}{=} \overline{(a||c)} \otimes f$ and the box $\Sigma \stackrel{\text{df}}{=} \text{box}(D)$ in figure 6, focussing on the behaviours involving a single execution of each of the transitions of Σ .

As we have seen, the step sequences are exactly the same for D and Σ (see theorems 2 and 3). Let us investigate whether the same will hold for causality semantics. For Σ , we have three possible process nets in this case (see, e.g., [2] for the definition of process nets of Petri nets), two of which are shown in figure 6: π_1 specifies that the occurrences of t_2 and t_3 are concurrent and both precede the occurrence of t_1 , while π_2 specifies that the occurrences are causally ordered as $t_2 t_3 t_1$ (the third possibility is π_3 with the occurrences ordered $t_3 t_2 t_1$).

Whereas the causality expressed by π_1 is something to be expected and can easily be matched using a suitable execution of the expression D , the case of π_2 is less clear. For instance, consider the only possible (up to the structural equivalence of expressions) execution corresponding to the scenario captured by π_2 :

$$\overline{(a||c)} \otimes f \equiv (\overline{a}||\overline{c}) \otimes f \xrightarrow{\{t_2\}} (\underline{a}||\overline{c}) \otimes f \xrightarrow{\{t_3\}} (\underline{a}||\underline{c}) \otimes f \equiv (a||c) \otimes \overline{f} \xrightarrow{\{t_1\}} (a||c) \otimes \underline{f}$$

It is not difficult to see that in the above the occurrences of t_2 and t_3 are totally *unrelated*. Indeed, if we replace the a with *any* expression (even one which is deadlocked), we still can derive:

$$\overline{(\dots||c)} \otimes f \equiv (\dots||\overline{c}) \otimes f \xrightarrow{\{t_3\}} (\dots||\underline{c}) \otimes f,$$

indicating that the overbar responsible for the occurrence of t_3 does not depend on executing anything else. Thus, on the level of box expressions, the partial order semantics does not seem to be suffering from the *token swapping* phenomenon discussed in [2] and which is illustrated by π_1 and π_2 . As a result, the causality relationship is as in the case of safe nets, despite the fact that we are now working with non-safe nets. This is highly relevant both from the theoretical and practical point of view (e.g., the unfolding technique based on branching

processes is more efficient if token swapping is absent, see [9]). Briefly, to implement such a semantics one can annotate the overbars and underbars with the positions of the atomic actions responsible for their creation. For our example, this would yield:

$$\begin{aligned} \overline{(a\|c) \otimes f}^\emptyset &\equiv (\overline{a}^\emptyset \|\overline{c}^\emptyset) \otimes f \xrightarrow{\{t_2\}} (\underline{a}_{\xi_1} \|\overline{c}^\emptyset) \otimes f \\ &\xrightarrow{\{t_3\}} (\underline{a}_{\xi_1} \|\underline{c}_{\xi_2}) \otimes f \equiv (a\|c) \otimes \overline{f}^{\xi_1, \xi_2} \xrightarrow{\{t_1\}} (a\|c) \otimes \underline{f}_{\xi_2} \end{aligned}$$

where $\xi_1 = v_1^\otimes v_1^\parallel v^a$, $\xi_2 = v_1^\otimes v_2^\parallel v^c$ and $\xi_3 = v_2^\otimes v^f$. Then we generate a causal precedence relation between occurrences of t_i and t_j provided that there is a path ξ_l from the root to a leaf of t_i labelling an overbar used to generate the occurrence of t_j . Since, in our case, $t_1 = v_2^\otimes \triangleleft v^f$, $t_2 = v_1^\otimes \triangleleft v_1^\parallel \triangleleft v^a$ and $t_3 = v_1^\otimes \triangleleft v_2^\parallel \triangleleft v^c$, the causality relation generated is exactly as in π_1 , and not as in π_2 .

To transfer the above way of generating causal partial orders for box expressions into the domain of boxes amounts to requiring that whenever there is a choice between two occurrences of the same place (like s_3 in figure 6 needed to generate an instance of t_3), then the one which leads to lesser causal orderings is chosen (and so neither π_2 nor π_3 would be generated).

5 Concluding Remarks

In this paper, we proposed a framework which supports two consistent (in a very strong sense, since the corresponding transition systems are isomorphic and not only bisimilar) concurrent semantics for a class of process expressions with both synchronous and asynchronous communication. We eliminated the need to maintain the safeness of the non-buffer places (required in the original PNA), which might be awkward for practical applications. It was replaced by auto-concurrency freeness which is a property still guaranteeing suitable concurrency semantics. We then discussed how a partial order semantics of boxes in a fragment of PNA could be obtained using additional annotations in process expressions.

Acknowledgements

We would like to thank the referees for helpful comments. This research was supported by the ARC JIP and EPSRC BEACON projects.

References

1. T.Basten and M.Voorhoeve: An Algebraic Semantics for Hierarchical P/T Nets. *ICATPN'95*, Springer, LNCS 935 (1995) 45–65. [192](#)
2. E. Best and R. Devillers: Sequential and Concurrent Behaviour in Petri Net Theory. *Theoretical Computer Science* 55 (1988) 87–136. [205](#)

3. E.Best, R.Devillers and J.Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. In: *Advances in Petri Nets 1992*, G.Rozenberg (Ed.). Springer, LNCS 609 (1992) 21–69. [192](#), [200](#)
4. E.Best, R.Devillers and M.Koutny: *Petri Net Algebra*. EATCS Monographs on TCS, Springer (2001). [192](#), [194](#), [197](#), [198](#), [200](#)
5. G.Boudol and I.Castellani: Flow Models of Distributed Computations: Three Equivalent Semantics for CCS. *Information and Computation* 114 (1994) 247–314. [192](#)
6. P.Degano, R.De Nicola and U.Montanari: A Distributed Operational Semantics for CCS Based on C/E Systems. *Acta Informatica* 26 (1988) 59–91. [192](#)
7. R.Devillers, H.Klaudel, M.Koutny, E.Pelz and F.Pommereau: Operational Semantics for PBC with Asynchronous Communication. *HPC'02*, SCS (2002) 314–319. [193](#), [195](#), [204](#)
8. R.Devillers, H.Klaudel, M.Koutny and F.Pommereau: Asynchronous Box Calculus. Technical Report CS-TR-759, Dept. of Comp. Sci., Univ. of Newcastle (2002). [194](#), [200](#)
9. J.Esparza, S.Römer and W.Vogler: An Improvement of McMillan's Unfolding Algorithm. *TACAS'96*, Springer, LNCS 1055 (1996) 87–106. [206](#)
10. R. J.van Glabbeek and F. V.Vaandrager: Petri Net Models for Algebraic Theories of Concurrency. *PARLE'87*, Springer, LNCS 259 (1987) 224–242. [192](#)
11. U.Goltz and R.Loogen: A Non-interleaving Semantic Model for Nondeterministic Concurrent Processes. *Fundamentae Informaticae* 14 (1991) 39–73. [192](#)
12. R.Gorrieri and U.Montanari: On the Implementation of Concurrent Calculi in Net Calculi: two Case Studies. *Theoretical Computer Science* 141(1-2) (1995) 195–252. [192](#)
13. C. A. R.Hoare: *Communicating Sequential Processes*. Prentice Hall (1985). [192](#)
14. P. W.Hoogers, H. C. M.Kleijn and P. S.Thiagarajan: An Event Structure Semantics for General Petri Nets. *Theoretical Computer Science* 153 (1996) 129–170. [194](#)
15. H.Klaudel and F.Pommereau: Asynchronous links in the PBC and M-nets. *ASIAN'99*, Springer, LNCS 1742 (1999) 190–200. [193](#)
16. H.Klaudel and F.Pommereau: A concurrent and Compositional Petri Net Semantics of Preemption. *IFM'2000*, Springer, LNCS 1945 (2000) 318–337. [193](#)
17. R.Milner: *Communication and Concurrency*. Prentice Hall (1989). [192](#)
18. E. R.Olderog: *Nets, Terms and Formulas*. Cambridge Tracts in Theoretical Computer Science 23, Cambridge University Press (1991). [192](#)
19. G. D.Plotkin: A Structural Approach to Operational Semantics. Technical Report FN-19, Computer Science Department, University of Aarhus (1981). [193](#)
20. W.Reisig: *Petri Nets. An Introduction*. EATCS Monographs, Springer (1985). [192](#), [196](#)

Refusal Simulation and Interactive Games

Irek Ulidowski

Department of Mathematics and Computer Science
University of Leicester, University Road, Leicester LE1 7RH, U.K.
I.Ulidowski@mcs.le.ac.uk

Abstract. We present refusal simulation relation as a candidate for the finest branching time relation that is solely based on all locally observable and testable properties of processes, and no other properties. Apart from several known characterisations, we introduce a new one in terms of interactive games. Two general formats of transition rules, based on the new Ordered Structural Operational Semantics approach [20], are given such that refusal simulation and its rooted version are preserved in arbitrary process languages definable within the respective formats.

1 Introduction

Process Languages (PLs) such as CCS [9], CSP [7,14] and ACP [3] are algebraic languages that comprise a small number of well chosen basic operators. There are operators for representing sequences of computations, nondeterministic choice between computations and a parallel composition of computations. We can write expressions in PLs, called processes, that represent specifications of concurrent systems such as communication protocols, and we can construct processes that describe possible designs or implementations of such systems. There are several methods in concurrency for establishing that a design or an implementation meets its specification, for example model checking, equational reasoning and term rewriting. In this paper, we shall concentrate on behavioural equivalences and preorders on processes. Using this method a design meets a specification if the pair of processes describing them is a member of a suitable relation on processes. It is common practice to define such relations in terms of the observable behaviour of processes, and as the notion of what is and what is not observable varies there are many interesting process relations in the concurrency literature. van Glabbeek compiled the branching time – linear time spectrum of process relations according to the type of observable behaviour that is used to distinguish processes [23]. In this paper we present refusal simulation relation which is a member of the branching time relations.

The development of refusal simulation and its properties have been driven by and are a result of three important requirements. The first concerns the style of definition of the relation: the choice of the basic observable properties and how they are used in the relation definition. Secondly, we require that our relation has alternative characterisations independent of its definition. Finally, we insist that the relation is well-behaved (namely, is a congruence) in an expressive class of process languages. Below, we further elaborate these three requirements.

Basic Observations and Style of Definition We distinguish between external behaviour of processes and their internal, unobservable and independent of the environment behaviour as represented by actions τ . We also take into account *divergence* of processes, which is the ability to perform an infinite sequence of silent actions. Hence, we are interested in *weak* relations on processes, for example weak bisimulation [9], as opposed to strong relations such as strong bisimulation [11,9]. We assume that the *basic* observations of process behaviour are the occurrences of actions, represented by the notation $\xrightarrow{\alpha}$, and the refusals of actions in stable states, represented by $\xrightarrow{\tau} \overset{\alpha}{\dashv}$. The second type of observation plays the crucial role in testing semantics [10,12,1,17] and in the failures/divergences model [7,14].

With these assumptions we define our relation in terms of the agreed basic observations on processes. We postulate that a process p is related to a process q if the observations of the behaviour of p can be matched by the corresponding observations about q . In other words, p is related to q if q can *simulate* actions and refusals of actions of p in step by step fashion. Moreover, we shall employ two forms of simulation, the *may* and the *must simulations* as in the testing scenarios, giving rise to *lower* and *upper* refusal simulation relations respectively.

Alternative Characterisations Refusal simulation has several natural and intuitive characterisations, and in this paper we present a new characterisation in terms of interactive games. The foremost characterisation is in terms of *copy+refusal tests* [17,18], which comprise tests that can make copies of processes and their derivatives, and tests them for the occurrences or refusals of actions. Another characterisation is in terms of a fragment of Hennessy Milner Logic [9]. Lower refusal simulation can be expressed as satisfaction of formulae $\phi ::= \text{tt} \mid \langle \mu \rangle \phi \mid \phi \wedge \phi$ and upper refusal simulation can be expressed as satisfaction of formulae $\phi ::= \text{ff} \mid [\mu] \phi \mid \phi \vee \phi$, where μ can be either an action or a refusal of action (in a *stable* state where no τ is possible) [18]. The third characterisation is in terms of a partial trace congruence generated by the ISOS format [17,18]. Finally, axiomatic characterisations are available in process languages from the ISOS and De Simone formats [18,19].

Congruence Property *Structural Operational Semantics* (SOS) [13] is a powerful and widely used method for assigning operational meaning to language operators (combinators) of process languages (algebras). In the SOS approach the meaning of process 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, called *format*, determines the kind of behavioural information that can be used to define operators. An operator is in a format if all its transition rules are in the format. A process language is in a format if all its operators are in the format.

The simplest of formats is the De Simone format [15]. All CCS and CSP operators are defined by rules in the De Simone format. De Simone rules with *negative premises* (expressions like $X \xrightarrow{a}$) and *copying* (multiple use of identical process variables) make up the GSOS format [5]. In [20] an extension of SOS method with orderings on transition rules, called *Ordered SOS* (OSOS), is developed to give an alternative and equally expressive formulation of the GSOS format. Instead of GSOS rules, which may contain negative premises, OSOS uses only positive GSOS rules but compensates for the lack of negative premises with the orderings on rules, the new feature that specifies the order of application of rules when deriving transitions of process terms. There are other formats, for example GSOS rules with composite terms in the premises are called *ntyft* rules [6], and additionally with predicates are called *panth* rules [24].

The important property that a good process relation should have is *congruence*: it should be *preserved* by all operators in a given format. Informally, an n -ary operator is said to preserve a process relation, if it produces two related processes from any two sets of n pairwise related subprocesses. It is well known that strong bisimulation is a congruence for any process language in all above listed formats. However, finding valid and elegant formats for weak process relations has proved more tricky due to the difficulty of guaranteeing the unobservable and uncontrollable character of silent actions in SOS rules.

In this paper we present a format for refusal simulation based on the OSOS approach. We impose several simple conditions on the structure of rules and on the orderings on rules. These conditions specify the usage of silent actions in rules, and restrict the orderings between rules with and without silent actions. For example, the *uncontrollable* and *independent of the environment* character of silent actions is represented via τ -rules, due to Bloom [4], by insisting that the set of rules for operator f contains a τ -rule τ_i of the form

$$\frac{X_i \xrightarrow{\tau} X'_i}{f(X_1, \dots, X_i, \dots, X_n) \xrightarrow{\tau} f(X_1, \dots, X'_i, \dots, X_n)}$$

for each *active* argument X_i . Informally, argument X_i of f is active if there is a rule for f with premises referring to 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 show how to solve this problem in Sect. 4.4.

A notion closely related to the uncontrollable character of silent actions is divergence. Results in [17,18,4] show that, in a setting with τ -rules, if one treats divergence as a form of deadlock, then rules with negative premises, as in the GSOS format, are unacceptable—we can construct operators defined by rules with negative premises that 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 a version of refusal simulation that is sensitive to divergence.

2 Refusal Simulation

Definition 1. A labelled transition system (or LTS, for short) 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 . Vis is a finite set of visible actions and it is ranged over by a, b, c . Action τ is the silent action and $\tau \notin \text{Vis}$. $\text{Act} = \text{Vis} \cup \{\tau\}$ 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*, and $p \xrightarrow{\alpha}$ when there is q such that $p \xrightarrow{\alpha} q$, and $p \not\xrightarrow{\alpha}$ otherwise. Sometimes we write $p \xrightarrow{\alpha} p'$ to mean $p \xrightarrow{\alpha} q$ and $p \equiv p'$. Expressions $p \xrightarrow{\tau} q$ and $p \xrightarrow{\alpha} q$, where $\alpha \neq \tau$, denote $p(\xrightarrow{\tau})^* q$ and $p(\xrightarrow{\tau})^* \xrightarrow{\alpha} q$ respectively. Note, $p \xrightarrow{\alpha} q$ does not stand for $p(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* q$ as in [9]. Given a sequence of actions $s = \alpha_1 \dots \alpha_n$, for $1 \leq i \leq n$, we say that q is a *s-derivative* of p if $p \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} q$. 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. A process is *strongly convergent* if all its derivatives are convergent. We assume that, if $\alpha = \tau$ then $p \xrightarrow{\alpha} p'$ means $p \xrightarrow{\tau} p'$ or $p \equiv p'$, else it is simply $p \xrightarrow{\alpha} p'$.

Refusal simulation [17,18] is a generalisation of ready simulation [5] and $\frac{2}{3}$ bisimulation [8] that treats τ actions as unobservable and is sensitive to divergence.

Definition 2. Let $(\mathcal{P}, A, \rightarrow)$ be an LTS. A relation $R \subseteq \mathcal{P} \times \mathcal{P}$ is a *lower refusal simulation* if, for all (p, q) in R , the following property holds for every α .

$$\begin{aligned} \forall p'. [p \xrightarrow{\alpha} p' \text{ implies } \exists q', q''. (q \xrightarrow{\tau} q' \xrightarrow{\hat{\alpha}} q'' \text{ and } p' R q'')] \text{ and} \\ p \xrightarrow{\tau} \not\xrightarrow{\alpha} \text{ implies } \exists q'. (q \xrightarrow{\tau} q' \xrightarrow{\tau} \not\xrightarrow{\alpha} \text{ and } p R q') \end{aligned}$$

A relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is an *upper refusal simulation* relation if, for all (p, q) in S the following property holds for every α .

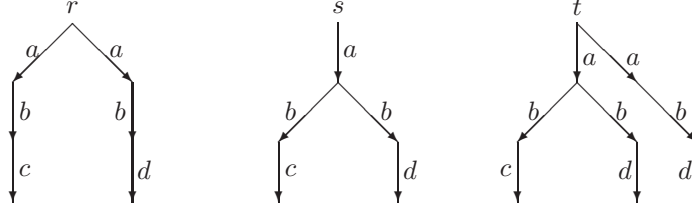
$$\begin{aligned} p \Downarrow \text{ implies } [q \Downarrow \text{ and} \\ \forall q'. [q \xrightarrow{\alpha} q' \text{ implies } \exists p', p''. (p \xrightarrow{\tau} p' \xrightarrow{\hat{\alpha}} p'' \text{ and } p' S q')] \text{ and} \\ q \xrightarrow{\tau} \not\xrightarrow{\alpha} \text{ implies } \exists p'. (p \xrightarrow{\tau} p' \xrightarrow{\tau} \not\xrightarrow{\alpha} \text{ and } p' S q)] \end{aligned}$$

Definition 3. $\xrightarrow{\sim}_L$, $\xrightarrow{\sim}_U$ and refusal simulation relation, $\xrightarrow{\sim}_{RS}$, are as follows:

$$\begin{aligned} p \xrightarrow{\sim}_L q &\equiv \exists R. R \text{ is a lower refusal simulation and } p R q, \\ p \xrightarrow{\sim}_U q &\equiv \exists S. S \text{ is an upper refusal simulation and } p S q, \\ p \xrightarrow{\sim}_{RS} q &\equiv p \xrightarrow{\sim}_L q \wedge p \xrightarrow{\sim}_U q. \end{aligned}$$

Relations $\xrightarrow{\sim}_L$, $\xrightarrow{\sim}_U$ and $\xrightarrow{\sim}_{RS}$ are clearly preorders. The equivalence associated with $\xrightarrow{\sim}_{RS}$, defined as $p \approx_{RS} q$ if $p \xrightarrow{\sim}_{RS} q$ and $q \xrightarrow{\sim}_{RS} p$, is refusal simulation equivalence.

Example 1. Consider processes r, s and t below.



Process r, s and t exhibit the difference between decorated trace relations, for example testing equivalence [10] or refusal equivalence [12], refusal simulation and bisimulation relations. It is easily checked that r and s are both testing equivalent and refusal equivalent as well as equivalent under all linear time process relations listed in [23]. But they are not refusal similar. For s can perform a after which both bc and bd are possible, and r after a must fail to perform either bc or bd . Refusal simulation is coarser than any of the bisimulation relations. Processes s and t are refusal similar but not bisimilar.

Example 2. Lower and upper refusal simulations are sensitive to divergence. Consider an LTS with $A = \{a, b\}$ and 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. We easily check that $q \sqsubseteq_{RS} p$: p is the same as q and it is ‘less’ divergent. On the other hand, $p \not\sqsubseteq_{RS} q$ since $p \not\sqsubseteq_L q$ ($p \xrightarrow{\tau} b$ but q cannot match this) and $p \not\sqsubseteq_U q$. The last holds because $p \downarrow$ and $q \uparrow$.

Example 3. A *faulty buffer* is a buffer which may lose or duplicate its messages. We shall abstract from the content of messages. A faulty buffer with capacity one holds no messages in its initial state; the only activity it can perform now is to input a message, represented by action in . Then, it can output this message by action out , duplicate the message and output both copies, or lose the message. Since the loss of a message may not be prevented or observed by the environment we represent it by action τ . After any of these three activities the buffer evolves to its initial state. Our buffer can be specified as follows:

$$B \equiv in.(out.B + out.out.B + \tau.B)$$

Informally, this specification says that each of the courses of actions after in , namely the output, the duplication and the loss, are ‘equally likely’ to happen. If, on the other hand, the duplication of messages was ‘relatively rare’ by virtue of not being possible in some runs of the buffer, we could specify this as follows:

$$B' \equiv in.(out.B' + out.out.B' + \tau.B') + in.(out.B' + \tau.B')$$

Both specifications of a faulty buffer are equally descriptive from the user’s point of view. They cannot be distinguished by finitary and local testing [18]:

they are copy+refusal testing equivalent. They are refusal simulation equivalent. This is one of four relations needed to prove the equivalence: $\{(B, B'), (out.B + out.out.B + \tau.B, out.B' + out.out.B' + \tau.B'), (out.B + out.out.B + \tau.B, out.B' + \tau.B'), (out.B, out.B')\}$. It proves $B \sqsubseteq_U B'$. However, B and B' are not weak bisimulation equivalent.

2.1 Rooted Refusal Simulation

It is well known that several important process operators, for example CCS ‘+’, do not preserve refusal simulation for to the same reason why they do not preserve weak bisimulation [9,3]. The standard solution is to use the *rooted* version of refusal simulation instead of the relation itself. The rooted versions of lower and upper refusal simulation are defined in the corresponding way as observation congruence is defined in terms of weak bisimulation [9]. Namely, $\overset{\alpha}{\Rightarrow}$ takes place of $\hat{\Rightarrow}$ for the first actions of the related processes. Rooted lower refusal simulation and rooted upper refusal simulation are denoted by \sqsubseteq_U^r and \sqsubseteq_L^r respectively. Rooted refusal simulation is written as \sqsubseteq_{RS}^r .

3 Interactive Games and Refusal Simulation

Refusal simulation is on a simple and intuitive concept that an observer interacts with processes and tries to simulate observations that she makes about of one of the processes by the corresponding observations about the other process. We have fixed our two atomic observations: they are the ability to perform an action, represented by $\overset{\alpha}{\rightarrow}$, and the ability to refuse an action, represented by $\overset{\tau}{\rightarrow} \overset{\alpha}{\rightarrow}$. Note, $p \overset{\tau}{\rightarrow} \overset{\alpha}{\rightarrow} p'$ means $p \overset{\tau}{\rightarrow} \overset{\alpha}{\rightarrow}$ and $p \equiv p'$. An alternative presentation of refusal simulation is with interaction games. We shall give two games, one for determining lower refusal simulation and the other for upper refusal simulation. In both types of games the moves that the players make are the same, and only the conditions for determining the winner are different. Our presentation builds upon game characterisations of bisimulation relations in [22,16].

An interactive game on a pair of processes (p_0, q_0) is played by two participants, player I and player II. Player I attempts to show that there is an observable difference in the behaviour of the initial processes, whereas player II tries to prevent this. Lower refusal simulation and upper refusal simulation games from a pair (p_0, q_0) are denoted by $\mathcal{G}_L(p_0, q_0)$ and $\mathcal{G}_U(p_0, q_0)$ respectively. A play of either of the games is a finite or an infinite sequence of the form $(p_0, q_0), \dots, (p_i, q_i), \dots$. For each i the pair (p_{i+1}, q_{i+1}) depends solely on the previous pair (p_i, q_i) and is decided by one of the following two moves.

- Player I makes an observation $p_i \overset{\alpha}{\rightarrow} p_{i+1}$, and then player II matches it with a corresponding observation from the other process: $q_i \overset{\tau}{\rightarrow} \overset{\hat{\alpha}}{\rightarrow} q_{i+1}$.
- Player I makes an observation $p_i \overset{\tau}{\rightarrow} \overset{\alpha}{\rightarrow} p_{i+1}$, and then player II matches is with a corresponding observation $q_i \overset{\tau}{\rightarrow} \overset{\tau}{\rightarrow} \overset{\alpha}{\rightarrow} q_{i+1}$.

Player I wins

1. The play is $(p_0, q_0), \dots, (p_n, q_n)$ and there is α such that either $p_n \xrightarrow{\alpha} p_{n+1}$ but not $(\exists q_{n+1}. q_n \xrightarrow{\tau} \hat{\alpha} q_{n+1})$, or $p_i \xrightarrow{\tau} \hat{\alpha} p_{i+1}$ but not $(\exists q_{n+1}. q_n \xrightarrow{\tau} \hat{\tau} \hat{\alpha} q_{n+1})$.

Player II wins

- 1'. The play is $(p_0, q_0), \dots, (p_n, q_n)$ and there is $i < n$ such that $p_i = p_n$ and $q_i = q_n$.
- 2'. The play has an infinite length.

Fig. 1. Winning conditions in lower refusal simulation games**Player I wins**

1. The play is $(p_0, q_0), \dots, (p_n, q_n), q_n \Downarrow$, and there is α such that either $p_n \xrightarrow{\alpha} p_{n+1}$ but not $(\exists q_{n+1}. q_n \xrightarrow{\tau} \hat{\alpha} q_{n+1})$, or $p_i \xrightarrow{\tau} \hat{\alpha} p_{i+1}$ but not $(\exists q_{n+1}. q_n \xrightarrow{\tau} \hat{\tau} \hat{\alpha} q_{n+1})$.
2. The play is $(p_0, q_0), \dots, (p_n, q_n), q_n \Downarrow$ but $p_n \Uparrow$.

Player II wins

- 1'. The play is $(p_0, q_0), \dots, (p_n, q_n), p_n \Downarrow$ and $q_n \Downarrow$, and there is some $i < n$ such that $p_i = p_n$ and $q_i = q_n$.
- 2'. The play is $(p_0, q_0), \dots, (p_n, q_n)$ and $q_n \Uparrow$.
- 3'. The play has an infinite length.

Fig. 2. Winning conditions in upper refusal simulation games

The first type of move is called an action move or an a move, and the second type is called an action refusal move or a refuse a move. If α is τ in the first move, then player II has an option of not evolving her process since $q_i \xrightarrow{\tau} \hat{\tau} q_i$. If α is τ in the second move, then the observation is that of stability. Note, that player I can always make a move since for every process there is an action that the process can either perform or refuse. Some plays may be infinite. This is not only when the process of player I is infinite. Consider the process $\mathbf{0}$. Player I can make the infinite sequence of observations $\mathbf{0} \xrightarrow{\tau} \hat{a} \xrightarrow{\tau} \hat{a} \xrightarrow{\tau} \hat{a} \dots$.

The game is played until one of the players wins. The sets of conditions for winning lower and upper refusal simulation games are given in Fig. 1 and Fig. 2 respectively. In bisimulation games [16] if a player is unable to make a move, then her opponent wins. This is the case for our player I. If Player II cannot match observations of player I, as in condition 1 in Figs. 1 and 2, then a difference in the behaviour of the initial processes has been exhibited. On the other hand, since player I can always make a move as argued above, the winning conditions for player II are not solely based on the inability of player I to make a move but depend on her exhausting all her options and repeating a configuration in the play. When this happens, condition 1' in Figs. 1 and 2, player I has failed to show an observable difference between the initial processes before finally making a move that has led to the repeat configuration. Player II also wins a play of either of the refusal simulation games when the play is infinite: condition 2'

in Fig. 1 and 3' in Fig. 2. As with bisimulation games [16] condition 1' is not necessary since it is subsumed by 2' in Fig. 1. However, we include it to stop plays of infinite length on finite state processes.

The last two conditions that we discuss are for upper refusal simulation game and determine the winner depending on the divergence/convergence of the involved processes. Player II wins when her process diverges: condition 2' in Fig. 2. Player I wins when she manages to lead player II to a configuration where her process diverges but her opponent's process converges: condition 2 in Fig. 2.

Remark 1. If both initial processes p and q are strongly convergent, then conditions 2 and 2' in Fig. 2 are obsolete, and the remaining conditions in Fig. 2 reduce to those in Fig. 1, thus making the two games the same: $\mathcal{G}_L(p, q) = \mathcal{G}_U(p, q)$.

A strategy for a player is a collection of rules which determine the player's next moves depending on play so far. A player uses a strategy in a play if every her move is according to the rules of the strategy. A strategy is a *winning strategy* if the player wins every play in which she uses the strategy.

Example 4. Consider processes p and q from Example 2. Any play in $\mathcal{G}_L(p, q)$ has the form (p, q) or $(p, q), (\mathbf{0}, \mathbf{0}), (\mathbf{0}, \mathbf{0})$. The first play is a result of player I making a refusal of action move, for example $p \xrightarrow{\tau} \overset{c}{\dashv} p$. Player II cannot match this move since her agent q has no τ -derivative that is stable. The second play follows from the a move by player I which player II is able to match, thus reaching the configuration $(\mathbf{0}, \mathbf{0})$. Now, player II always wins because player I can only perform action refusal moves, and they lead to a repeat configuration. Overall, although player II is able to win a play of $\mathcal{G}_L(p, q)$ player I has a winning strategy: make a refuse α move as the first move for $\alpha \neq a$. Note, $p \not\approx_L q$. Every play of $\mathcal{G}_L(q, p)$ has the form $(q, p), (q, p)$ or $(q, p), (\mathbf{0}, \mathbf{0}), (\mathbf{0}, \mathbf{0})$. In both cases player II is able to match the moves by player I, and player II always wins because there are repeat pairs. Note, $q \approx_L p$. Player II always wins in $\mathcal{G}_U(p, q)$ by condition 2' in Fig. 2 since $q \uparrow$. Note, $q \approx_U p$. Player I wins every play of $\mathcal{G}_U(q, p)$ because $q \uparrow$ and $p \downarrow$: condition 2 in Fig. 2. Note, $p \not\approx_U q$.

Example 5. Consider processes r, s and t in Example 1. In $\mathcal{G}_L(r, s)$ it is clear that player II can match every move of player I before player I repeats a configuration. Hence, player II has a winning strategy. Since r and s are strongly convergent, player II has also a winning strategy in $\mathcal{G}_U(r, s)$. The situation is different in $\mathcal{G}_L(s, r)$. Here, player I has an advantage. Her a move must be matched by player II making one of the a moves from r . Any of such moves commits player II to either bc or bd branch, whereas player I still has a choice of both bc and bd branches. Next, player I chooses the branch that is not available to player II. Hence, player I is guaranteed to win if she makes correct moves. Similarly, she has a winning strategy in $\mathcal{G}_U(s, r)$. Note, $s \not\approx_L r$ and $r \not\approx_U s$.

When player II has a winning strategy in $\mathcal{G}_L(p, q)$, we say that p is lower refusal simulation game related to q , which is abbreviated as $p \preceq_L q$. This means that player II is able to

- match any $p \xrightarrow{\alpha} p'$ move by an appropriate $q \xrightarrow{\tau} \hat{\alpha} q'$ move with p' and q' lower refusal simulation game related, and
- match any $p \xrightarrow{\tau} \alpha q$ move with $q \xrightarrow{\tau} \tau \alpha q'$ with p' and q' lower refusal simulation game related.

These are the first two conditions of lower refusal simulation. Hence:

Proposition 1. For processes p and q we have $p \preceq_L q$ if and only if $p \sqsubseteq_L q$.

Correspondingly, $p \preceq_U q$ is a shorthand for player II having a winning strategy in $\mathcal{G}_U(q, p)$. Notice, that p and q are swapped to match the reading of the correspondence between $p \preceq_U q$ and $p \sqsubseteq_U q$. Starting with the pair (q, p) , player II

- wins immediately because $p \uparrow$: condition 2' in Fig. 2, or
- $q \downarrow$ and $p \downarrow$, and she is able to
 - match any move $q \xrightarrow{\alpha} q'$ by an appropriate $p \xrightarrow{\tau} \hat{\alpha} p'$ move, with q' and p' upper refusal simulation game related, and
 - match any move $q \xrightarrow{\tau} \alpha q'$ with $p \xrightarrow{\tau} \tau \alpha p'$, with q' and p' upper refusal simulation game related.

We easily check that the above conditions are logically equivalent to the conditions in Definition 2 for an upper refusal simulation relation. Hence,

Proposition 2. For processes p and q we have $p \preceq_U q$ if and only if $p \sqsubseteq_U q$.

We say that p is refusal simulation game related to q , written as $p \preceq_{RS} q$, if $p \preceq_L q$ and $p \preceq_U q$.

Theorem 1. For processes p and q we have $p \preceq_{RS} q$ if and only if $p \sqsubseteq_{RS} q$.

3.1 Games for Rooted Refusal Simulation

In order to obtain games for rooted lower and rooted upper refusal simulations we only need to change the first move that player I and player II can make. The moves are as before except that when player I makes an observation $p_0 \xrightarrow{\tau} p_1$, player II must match it with $q_0 \xrightarrow{\tau} \tau q_1$, i.e. by performing at least one τ action. After the first move, all plays in both games proceed as before. With this change we define rooted lower and rooted upper refusal simulation game relations in a corresponding way, and denote them by \preceq_L^r and \preceq_U^r respectively.

We say that p is rooted refusal simulation game related to q , written as $p \preceq_{RS}^r q$, if $p \preceq_L^r q$ and $p \preceq_U^r q$.

Theorem 2. For processes p and q we have $p \preceq_{RS}^r q$ if and only if $p \sqsubseteq_{RS}^r q$.

4 Formats for Refusal Simulations

In this section we present the Ordered SOS method for defining process operators [20] and illustrate its usefulness. We establish several intuitive conditions on OSOS rules and their orderings such that OSOS operators that satisfy these conditions preserve \sqsubseteq_{RS} and \sqsubseteq_{RS}^r . The OSOS has the same expressive power as the GSOS approach [20], but we find OSOS more convenient for expressing restrictions that are required for the preservation of weak process relations.

4.1 SOS Rules with Orderings

\mathbf{Var} is a countable set of variables ranged over by X, X_i, Y, Y_i, \dots . 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, \dots . 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 $V \subseteq \mathbf{Var}$, denoted by $\mathbb{T}(\Sigma, V)$, is ranged over by t, t', \dots . $\mathit{Var}(t) \subseteq \mathbf{Var}$ is the set of variables in a term t .

The set of *closed terms*, written as $\mathbb{T}(\Sigma)$, is ranged over by p, q, u, v, \dots . In the setting of process languages these terms are called process terms. A Σ context with n holes $C[X_1, \dots, X_n]$ is a member of $\mathbb{T}(\Sigma, \{X_1, \dots, X_n\})$. If t_1, \dots, t_n are Σ terms, then $C[t_1, \dots, t_n]$ 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, \dots, p_n , for any $n \in \mathbb{N}$, will often be written as \mathbf{p} when the length is understood from the context. Given any binary relation R on closed terms and \mathbf{p} and \mathbf{q} of length n , we will write \mathbf{pRq} to mean $p_i R q_i$ for all $1 \leq i \leq n$. Moreover, instead of $f(X_1, \dots, X_n)$ we will often write $f(\mathbf{X})$ when the arity of f is understood. A preorder \sqsubseteq on $\mathbb{T}(\Sigma)$ is a precongruence if $\mathbf{p} \sqsubseteq \mathbf{q}$ implies $C[\mathbf{p}] \sqsubseteq C[\mathbf{q}]$ for all \mathbf{p} and \mathbf{q} of length n and all Σ contexts $C[\mathbf{X}]$ with n holes. Similarly, an operator $f \in \Sigma_n$ preserves \sqsubseteq if, for \mathbf{p} and \mathbf{q} as above, $\mathbf{p} \sqsubseteq \mathbf{q}$ implies $f(\mathbf{p}) \sqsubseteq f(\mathbf{q})$. A PL preserves \sqsubseteq if all its operators preserve \sqsubseteq .

A *substitution* is a mapping $\mathbf{Var} \rightarrow \mathbb{T}(\Sigma)$. Substitutions are ranged over by ρ and ρ' and they extend to $\mathbb{T}(\Sigma, \mathbf{Var}) \rightarrow \mathbb{T}(\Sigma)$ mappings in a standard way. For t with $\mathit{Var}(t) \subseteq \{X_1, \dots, X_n\}$ we write $t[p_1/X_1, \dots, p_n/X_n]$ or $t[\mathbf{p}/\mathbf{X}]$ 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

$$\frac{\{ X_i \xrightarrow{\alpha_{ij}} Y_{ij} \}_{i \in I, j \in J_i}}{f(\mathbf{X}) \xrightarrow{\alpha} C[\mathbf{X}, \mathbf{Y}]},$$

where \mathbf{X} is the sequence X_1, \dots, X_n and \mathbf{Y} is the sequence of all Y_{ij} , and all process variables in \mathbf{X} and \mathbf{Y} are distinct. Variables in \mathbf{X} are the *arguments* of f . Moreover, $I \subseteq \{1, \dots, n\}$ and all J_i are finite subsets of \mathbb{N} , and $C[\mathbf{X}, \mathbf{Y}]$ is a context. Let r be the above rule. Then, f is the *operator* of r and $\mathit{rules}(f)$ 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 $\mathit{pre}(r)$. The transition below the bar in r is the *conclusion*, written as $\mathit{con}(r)$. α in the conclusion of r is the *action* of r , written as $\mathit{act}(r)$, and $C[\mathbf{X}, \mathbf{Y}]$ is the *target* of r . Given $i \in I$, the set of all actions α_{ij} in the premises of r is denoted by $\mathit{actions}(r, i)$. The set of all actions in the premises of r is denoted by $\mathit{actions}(r)$. A rule is an *action rule* if $\tau \notin \mathit{actions}(r)$.

Next, we define *orderings* on rules [20]. Let $<_f$ be a binary relation on $\mathit{rules}(f)$. 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 $\bigcup_{f \in \Sigma} <_f$. Note, that orderings do not need to be transitive. We will write $higher(r)$ for $\{r' \mid r < r'\}$.

Example 6. Ordered rules have the same effect as rules with negative premises. We give an alternative definition of the sequential composition operator ‘;’ from [5]. The rules are given below, and 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 .

$$\frac{X \xrightarrow{a} X'}{X; Y \xrightarrow{a} X'; Y} r_{a*} \quad \frac{X \xrightarrow{\tau} X'}{X; Y \xrightarrow{\tau} X'; Y} \tau_1 \quad \frac{Y \xrightarrow{c} Y'}{X; Y \xrightarrow{c} Y'} r_{*c} \quad \frac{Y \xrightarrow{\tau} Y'}{X; Y \xrightarrow{\tau} Y'} \tau^2$$

Definition 5. An *Ordered SOS* (or OSOS, for short) PL is a tuple $(\Sigma, A, R, <)$, where Σ is a finite set of operators, $A \subseteq \text{Act}$, 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 PL $G = (\Sigma, A, R, <)$, we associate a unique transition relation \rightarrow with G as in [20].

Definition 6. Let $r \in \text{rules}(f)$ and $pre(r) = \{X_i \xrightarrow{\alpha_{ij}} Y_{ij} \mid i \in I, j \in J_i\}$. Rule r *applies* to $f(\mathbf{u})$ if $u_i \xrightarrow{\alpha_{ij}}$ for all relevant i and j . Rule r is *enabled* at term $f(\mathbf{u})$ if r applies to $f(\mathbf{u})$ and all rules in $higher(r)$ do not apply.

Now, we return to Example 6. 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. That is if $p \xrightarrow{\tau}$ and $p \xrightarrow{a}$ for all a . When p is a purely divergent, for example defined by $p \xrightarrow{\tau} p$ with p being a constant, then q will never start in $p; q$ since τ_1 is always applicable.

Having defined \rightarrow for a given OSOS PL $(\Sigma, A, R, <)$, we easily construct $(T(\Sigma), A, \rightarrow)$ as the LTS for the PL. Lower and upper refusal simulation relations, refusal simulation and their rooted versions are defined over the LTS as in Sect. 2.

4.2 Conditions on OSOS Rules and Orderings

Arbitrary OSOS operators do not preserve weak process relations. Consider operator $see\text{-}\tau$, defined by a single rule, that makes silent actions visible.

$$\frac{X \xrightarrow{\tau} X'}{see\text{-}\tau(X) \xrightarrow{a} see\text{-}\tau(X')}$$

Although CCS-like processes $\tau.\mathbf{0}$ and $\tau.\tau.\mathbf{0}$ are equivalent according to most weak process relations, they are not equivalent when placed in the $see\text{-}\tau$ context: we have $see\text{-}\tau(\tau.\tau.\mathbf{0}) \xrightarrow{a} \mathbf{0}$ but $see\text{-}\tau(\tau.\mathbf{0}) \xrightarrow{a} \tau$.

We introduce several conditions on OSOS rules and their orderings which guarantee that OSOS operators satisfying these conditions do indeed preserve

refusal simulation. For this purpose we introduce the notions of *active arguments*, *action rules*, *silent rules* and *copies*. The i th argument X_i is *active* in rule r , written as $i \in \text{active}(r)$, if it appears in a premise of r [4]. Overloading the notation we write $\text{active}(f)$ instead of $\{i \mid i \in \text{active}(r) \text{ and } r \in \text{rules}(f)\}$. The i th argument of $f(\mathbf{X})$ is active if it is active in a rule for f . Action rules are transition rules as in Definition 4 except that only visible actions are allowed in the premises. Hence, they have the following form.

$$\frac{\{ X_i \xrightarrow{a_{ij}} Y_{ij} \}_{i \in I, j \in J_i}}{f(\mathbf{X}) \xrightarrow{\alpha} C[\mathbf{X}, \mathbf{Y}]}$$

Other types of transition rules describe the unobservable behaviour of processes in terms of the unobservable behaviour of their components. We shall call such rules the *silent rules* and define them as rules r such that $\text{act}(r) = \tau$ and $\text{actions}(r) = \{\tau\}$. The most widely used rules among the silent rules are the τ -rules, also called *patience rules* [4], as defined in the Introduction. The τ -rule for the i th argument of f is denoted by τ_i when f is clear from the context. The second form of silent rules is the *silent choice rule* (first rule below).

$$\frac{X_i \xrightarrow{\tau} X'_i}{f(X_1, \dots, X_i, \dots, X_n) \xrightarrow{\tau} X'_i} \qquad \frac{X_i \xrightarrow{\alpha} X'_i}{f(X_1, \dots, X_i, \dots, X_n) \xrightarrow{\alpha} X'_i}$$

Such rules give us the ability to define operators like the CCS ‘+’, the prefix iteration operator [2], the *interrupt* operator [9] and many timed operators. The silent choice rule for the i th argument of f is denoted by τ^i when f is clear from the context. Given f , we write $\text{tau}(i)$ 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 i th argument if it has the form of the second rule above.

Multiple occurrences of process variables in rules are called *copies*. They can be divided into *explicit* and *implicit* copies [17,18]. 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 $\text{implicit-copies}(r)$ when f is clear from the context.

We now define general formats for refusal simulation and for rooted refusal simulation. Consider the conditions in Fig. 3, where f is an operator of an OSOS PL, $<$ is the ordering on the rules for f , r and r' range over $\text{rules}(f)$, and τ_i and τ^i are the τ -rule and the silent choice rule for the i th argument of f respectively. Conditions in Fig. 3 are implicitly quantified over all r and r' in $\text{rules}(f)$, and are also quantified over all appropriate i . Conditions (1)–(4) restrict the ordering on rules. The intuition for (1) is that before we apply r' we must check that no rules with higher priority, and thus their associated silent rules, are applicable. (2) 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 . (3) insists that silent rules are always enabled, and (4) allows only implicit copies of stable arguments.

$$\mathbf{if} \ r' < r \ \mathbf{and} \ i \in \mathit{active}(r) \ \mathbf{then} \ r' < \mathit{tau}(i) \quad (1)$$

$$\mathbf{if} \ \mathit{tau}(i) < r \ \mathbf{and} \ i \in \mathit{active}(r') \cup \mathit{active}(\mathit{higher}(r')) \ \mathbf{then} \ r' < r \quad (2)$$

$$\mathbf{not} \ (\mathit{tau}(i) < \mathit{tau}(i)) \quad (3)$$

$$\mathbf{if} \ i \in \mathit{implicit-copies}(r) \ \mathbf{then} \ r < \mathit{tau}(i) \quad (4)$$

Fig. 3. Conditions on the orderings of OSOS rules

4.3 Refusal Simulation Format

Definition 7. Assume operator f is defined by a set R of action rules and silent rules. If i th argument of f is active, then $\mathit{tau}(i) \in R$. The silent rules for all active arguments of f is 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* .

Definition 8. An operator f is τ -preserving if the set of its defining rules consists of OSOS actions rules, the associated with f τ -rules and no other rules, and the ordering on the rules satisfies the conditions in Fig. 3. A set of rules is in the *refusal simulation* format, or **rso** for short, if it is the set of the defining rules for a set of τ -preserving operators. An operator is in the **rso** format, if its defining rules are in the format. A PL is in the **rso** format, if all its operators are **rso**.

Clearly, an operator is in the refusal simulation format precisely when it is τ -preserving. Most of the popular process language operators, for example parallel composition, restriction and hiding operators [9,7], are **rso** operators as the silent rules that are associated with them are all τ -rules.

The reader is referred to [20] for examples of τ -preserving operators that show that (1)–(4) are necessary. Similar examples will work for τ -sensitive operators that we introduce in the next subsection.

Our format is more general than the ISOS format [17,18] as it permits implicit copies of arguments provided they can be shown to be stable. Of course, the **rso** format does not use negative premises but it achieves their effect by ordering the rules: see Example 6. The main result of this subsection is the following.

Theorem 3. All **rso** PLs preserve refusal simulation preorder \sqsubseteq_{RS} .

4.4 Rooted Refusal Simulation Format

The CCS ‘+’ is not **rso** operator as it uses silent choice rules for its two active arguments instead of τ -rules as required by Definition 8. But ‘+’ preserves rooted refusal simulation, so we seek a format that contains **rso** and also permits operators like ‘+’. This format is a reformulation of the **rebo** format in [21].

Definition 9. An operator f is τ -sensitive if the set R of rules for f consists of OSOS actions rules, the associated with f silent rules (either τ -rules or silent choice rules) and no other rules, and if R contains the silent choice rule for argument i , then all rules with active i are choice rules, and the ordering on the rules satisfies the conditions in Fig. 3. A set of rules is in the *rooted refusal simulation* format, or *rrso* for short, if it is the set of the defining rules for a set of operators that 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. An operator is in the *rrso* format, if its defining rules are in the format. A PL is in the *rrso* format, if all its operators are *rrso*.

The CCS ‘+’, ‘;’ from Example 6, the prefix iteration operator [2] and Milner’s interrupt operator $^{\sphericalangle}$ [9] are τ -sensitive *rrso* operators. The last two are defined below. Notice that the first and the last rule are choice rules. When α is τ the second last rule is the τ -rule for the first argument of $^{\sphericalangle}$, and the last rule is the silent choice rule for the second argument of $^{\sphericalangle}$.

$$\frac{X \xrightarrow{\alpha} X'}{a^*X \xrightarrow{\alpha} X'} \quad a^*X \xrightarrow{a} a^*X \quad \frac{X \xrightarrow{\alpha} X'}{X \wedge Y \xrightarrow{\alpha} X' \wedge Y} \quad \frac{Y \xrightarrow{\alpha} Y'}{X \wedge Y \xrightarrow{\alpha} Y'}$$

Theorem 4. [21] All *rrso* PLs preserve rooted refusal simulation preorder \sqsubseteq_{RS}^r .

The OSOS approach, and the proposed formats for refusal simulation, are very convenient for defining process languages with time where certain timed properties, such as time determinacy, time persistence and maximal progress, are required to hold. The reader is referred to [21] for details of such formulations of process languages that satisfy a variety of timed properties.

5 Conclusion

We have presented refusal simulation preorder as a candidate for the finest branching time relation on processes that captures precisely all locally observable and testable properties of processes, and no other properties. In [18] it was shown that refusal simulation coincides with copy+refusal testing equivalence for image-finite processes. We have developed a new characterisation of refusal simulation in terms of interactive games. Process p is refusal similar to process q if and only if player II has winning strategies in two types of interactive games, namely lower and upper refusal simulation games.

Two general formats of transition rules, based on the new Ordered Structural Operational Semantics approach [20], have been presented. We have argued that refusal simulation and its rooted version are preserved in arbitrary PLs definable within the respective formats. Several process operators have been defined within the formats to demonstrate their usefulness.

References

1. S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987. 209
2. L. Aceto, W. Fokkink, R. van Glabbeek, and A. Ingólfssdóttir. Axiomatizing prefix iteration with silent steps. *Information and Computation*, 127:1–52, 1996. 219, 221
3. J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990. 208, 213
4. B. Bloom. Structural operational semantics for weak bisimulations. *Theoretical Computer Science*, 146:27–68, 1995. 210, 219
5. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995. 210, 211, 218
6. J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118(2):263–299, 1993. 210
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 208, 209, 220
8. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1992. 211
9. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. 208, 209, 211, 213, 219, 220, 221
10. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984. 209, 212
11. D. M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conference on Theoretical Computer Science*, volume 104 of *LNCS*. Springer, 1981. 209
12. I. C. C. Phillips. Refusal testing. *Theoretical Computer Science*, 50:241–284, 1987. 209, 212
13. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. 209
14. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. 208, 209
15. R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985. 210
16. C. Stirling. *Modal and Temporal Properties of Processes*. Graduate Texts in Computer Science. Springer, 2001. 213, 214, 215
17. I. Ulidowski. Equivalences on observable processes. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science LICS'92*, pages 148–159. IEEE, Computer Science Press, 1992. 209, 210, 211, 219, 220
18. I. Ulidowski. *Local Testing and Implementable Concurrent Processes*. PhD thesis, Imperial College, University of London, 1994. 209, 210, 211, 212, 219, 220, 221
19. I. Ulidowski. Finite axiom systems for testing preorder and De Simone process languages. *Theoretical Computer Science*, 239(1):97–139, 2000. 209
20. 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. To appear in *Information and Computation*. 208, 210, 216, 217, 218, 220, 221

21. I. Ulidowski and S. Yuen. Process languages for rooted eager bisimulation. In D. Miller and C. Palamidessi, editors, *Proceedings of the 11th Conference on Concurrency Theory CONCUR 2000*, volume 1877 of *LNCS*. Springer, 2000. 220, 221
22. J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic*, volume 130 of *Studies in Logic*. North-Holland, 1991. 213
23. R. J. van Glabbeek. The linear time - branching time spectrum I. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier Science, 2001. 208, 212
24. C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2(2):274–302, 1995. 210

A Theory of May Testing for Asynchronous Calculi with Locality and No Name Matching

Prasanna Thati, Reza Ziaei, and Gul Agha

University of Illinois at Urbana-Champaign
{thati,ziaei,agha}@uiuc.edu

Abstract. We present a theory of may testing for asynchronous calculi with locality and no name matching. Locality is a non-interference property that is common in systems based on object-paradigm. Concurrent languages such as Join and Pict disallow name matching, which is akin to pointer comparison in imperative languages, to provide for an abstract semantics that would allow useful program transformations. May testing is widely acknowledged to be an effective notion for reasoning about safety properties. We provide a trace-based characterization of may testing for versions of asynchronous π -calculus with locality and no name matching, which greatly simplifies establishing equivalences between processes. We also exploit the characterization to provide a complete axiomatization for the finitary fragment of the calculi.

1 Introduction

Experience with applying the π -calculus [10] to distributed systems has shown that it is necessary to make additional ontological commitments. Specifically, variants of π -calculus with *asynchrony*, *locality*, and *absence* of name matching have received wide attention recently [2, 4, 7, 9]. Asynchronous message passing is more common in distributed systems than synchronous communication that is assumed primitive in the π -calculus. The discipline of locality, which disallows a process from receiving messages targeted to a name previously received by the process, is typical in systems based on an object paradigm [1]. Name matching is analogous to pointer comparison in imperative languages; disallowing it enables certain performance optimizations. In fact, name comparisons are disallowed by concurrent languages such as Pict [12]. In any case, comparing names is rarely useful in programming; the behavior observed while communicating at a name is all that matters and not the specific name used for communication. A variant of the π -calculus that embodies these three features is $L\pi$ [9].

We develop a theory of *may* testing for two subcalculi of asynchronous π -calculus [3]: one with only locality, called $L\pi_{=}$, and the other with both locality and no name matching, called $L\pi$. May testing [11] is a specific instance of the general notion of behavioral equivalence where two processes are said to be equivalent if they have the same success properties in all contexts. A context in

may testing consists of an observing process that runs in parallel and interacts with the process being tested, and success is defined as the observer signaling a special event. The non-determinism in execution may give rise to different runs. A process is said to pass a test proposed by an observer, if there exists a run that leads to a success. By viewing a success as something bad happening, may testing can be used for reasoning about safety properties.

Because the definition of may testing involves a universal quantification over contexts, it is very difficult to prove equivalences directly from the definition. A typical approach to circumvent the problem, is to find an alternate characterization of the equivalence, which involves only the processes being compared. We provide an alternate characterization of may testing in $L\pi_{=}$ and $L\pi$. The characterizations are *trace* based, and directly build on the known characterization for asynchronous π -calculus [3]. In fact, we generalize the usual definition of may testing to a parameterized version, where the parameter determines the set of observers that is used to decide the order.

Our second result is to provide complete axiomatizations of finitary $L\pi_{=}$ and $L\pi$ (for processes with no replication). The axiomatizations highlight the differences that arise due to locality and lack of name matching. In addition to laws that are true for asynchronous π -calculus, we obtain laws that are true only in the presence of locality and the absence of name matching. Further, the inference rules for parameterized may testing generalize the ones for the usual may testing. Complete proofs of lemmas and theorems can be found in [13].

2 The Calculus $L\pi_{=}$

We assume an infinite set of names \mathcal{N} , and let u, v, w, x, y, z, \dots range over \mathcal{N} . The set of processes, ranged over by P, Q, R , is defined by the following restricted π -calculus grammar.

$$P := 0 \mid \bar{x}y \mid x(y).P \mid P|P \mid (\nu x)P \mid [x = y]P \mid !x(y).P$$

The name x is said to be the subject of the output $\bar{x}y$ and the input $x(y).P$. The locality property is enforced by requiring that for every subterm of the form $x(y).P$, the bound name y does not occur as the subject of an input in P .

For a tuple \tilde{x} , we denote the set of names occurring in \tilde{x} by $\{\tilde{x}\}$. We write \tilde{x}, \tilde{y} for the result of appending \tilde{y} to \tilde{x} . We let \hat{z} range over $\{\emptyset, \{z\}\}$. The term $(\nu \hat{z})P$ is $(\nu z)P$ if $\hat{z} = \{z\}$, and P otherwise. The functions for free names, bound names and names, $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$, of a process, and alpha equivalence on processes are defined as usual. We use the usual definition and notational convention for name substitutions, and let σ range over them. Name substitution on processes is defined modulo alpha equivalence with the usual renaming of bound names to avoid captures. We write $P\sigma$ and $x\sigma$ to denote the result of applying σ to P and x respectively.

We use an early style labeled transition system for the operational semantics (see table 1). The transition system is defined modulo alpha-equivalence on processes in that alpha-equivalent processes have the same transitions. The

Table 1. An early style labeled transition system for $L\pi_{=}$

$INP: x(y).P \xrightarrow{xz} P\{z/y\}$	$OUT: \bar{x}y \xrightarrow{\bar{x}y} 0$
$PAR: \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 P_2 \xrightarrow{\alpha} P'_1 P_2} \quad bn(\alpha) \cap fn(P_2) = \emptyset$	$COM: \frac{P_1 \xrightarrow{\bar{x}y} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} P'_1 P'_2}$
$RES: \frac{P \xrightarrow{\alpha} P'}{(\nu y)P \xrightarrow{\alpha} (\nu y)P'} \quad y \notin n(\alpha)$	$OPEN: \frac{P \xrightarrow{\bar{x}y} P'}{(\nu y)P \xrightarrow{\bar{x}(y)} P'} \quad x \neq y$
$CLOSE: \frac{P_1 \xrightarrow{\bar{x}(y)} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 P_2 \xrightarrow{\tau} (\nu y)(P'_1 P'_2)} \quad y \notin fn(P_2)$	
$REP: \frac{P !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$	$MATCH: \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'}$

symmetric versions of COM , $CLOSE$, and PAR are not shown. Transition labels, which are also called actions, can be of five forms: τ (a silent action), $\bar{x}y$ (free output of a message with target x and content y), $\bar{x}(y)$ (bound output), xy (free input of a message) and $x(y)$ (bound input). The relation $\xrightarrow{x(y)}$ is defined by the additional rule $P \xrightarrow{x(y)} Q$ if $P \xrightarrow{xy} Q$ and $y \notin fn(P)$. We denote the set of all visible (non- τ) actions by \mathcal{L} , let α range over \mathcal{L} , and let β range over all the actions. The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are defined on \mathcal{L} the usual way. As a uniform notation for free and bound actions we adopt the following convention from [3]: $(\emptyset)\bar{x}y = \bar{x}y$, $(\{y\})\bar{x}y = \bar{x}(y)$, and similarly for input actions. We define a complementation function on \mathcal{L} as $(\hat{y})xy = (\hat{y})\bar{x}y$, $(\hat{y})\bar{x}y = (\hat{y})xy$.

We let s, r, t range over \mathcal{L}^* . The functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are extended to \mathcal{L}^* the obvious way. Complementation on \mathcal{L} is extended to \mathcal{L}^* the obvious way. Alpha equivalence over traces is defined as expected, and alpha-equivalent traces are not distinguished. From now on, only *normal* traces $s \in \mathcal{L}^*$ that satisfy the following hygiene condition are considered: if $s = s_1.\alpha.s_2$, then $(n(s_1) \cup fn(\alpha)) \cap bn(\alpha.s_2) = \emptyset$. For an action α and a set of traces S we define $\alpha.S = \{\alpha.s \mid s \in S\}$.

We use \implies to denote the reflexive transitive closure of $\xrightarrow{\tau}$, and $\xRightarrow{\beta}$ to denote $\implies \xrightarrow{\beta} \implies$. For $s = l.s'$ we use $P \xrightarrow{s} Q$ to denote $P \xrightarrow{l} \xrightarrow{s'} Q$, and similarly $P \xRightarrow{s} Q$ to denote $P \xRightarrow{l} \xRightarrow{s'} Q$. We write $P \xrightarrow{s}$ if $P \xrightarrow{s} Q$ for some Q , and similarly for $P \xrightarrow{\tau}$ and $P \xrightarrow{\tau}$. We say P exhibits the trace s if $P \xRightarrow{s}$.

We now instantiate the testing framework [5] on $L\pi_{=}$. In fact, by extending the notion of locality, we consider a generalized version of may testing that supports encapsulation. We define a parameterized may preorder $\bar{\sim}_{\rho}$, where only observers that do not listen on names in ρ are used to decide the order. The set of names ρ can be interpreted as being “owned” by the process being tested, in that

Table 2. A preorder relation on traces

(L1)	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.s_2$	if $(\hat{y})s_2 \neq \perp$
(L2)	$s_1.(\hat{y})(\alpha.xy.s_2) \prec s_1.(\hat{y})xy.\alpha.s_2$	if $(\hat{y})(\alpha.xy.s_2) \neq \perp$
(L3)	$s_1.(\hat{y})s_2 \prec s_1.(\hat{y})xy.\bar{x}y.s_2$	if $(\hat{y})s_2 \neq \perp$
(L4)	$s_1.\bar{x}w.(s_2\{w/y\}) \prec s_1.\bar{x}(y).s_2$	

any testing context is assumed to have only the capability of sending messages to these names. The reader may note that $\stackrel{\sqsubseteq}{\sim}_\emptyset$ is the usual may preorder.

Definition 1 (may testing). *Observers are processes that can emit a special message $\bar{u}\mu$. We let O range over the set of observers. We say O accepts a trace s if $O \xrightarrow{\bar{s}\bar{u}\mu}$. For P, O , we say P may O if $P|O \xrightarrow{\bar{u}\mu}$. Let $\text{rcp}(P)$ be the set of all free names in P that occur as the subject of an input in P . For any given ρ we say $P \stackrel{\sqsubseteq}{\sim}_\rho Q$ if for every O such that $\text{rcp}(O) \cap \rho = \emptyset$, P may O implies Q may O . We say $P \simeq_\rho Q$ if $P \stackrel{\sqsubseteq}{\sim}_\rho Q$ and $Q \stackrel{\sqsubseteq}{\sim}_\rho P$. Note that $\stackrel{\sqsubseteq}{\sim}_\rho$ is reflexive and transitive, and \simeq_ρ is an equivalence relation. \square*

The larger the parameter of a preorder, the smaller the observer set that is used to decide the order. Hence if $\rho_1 \subset \rho_2$, we have $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$. However, $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ need not imply $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$. For instance, $0 \simeq_{\{x\}} \bar{x}x$, but only $0 \stackrel{\sqsubseteq}{\sim}_\emptyset \bar{x}x$ and $\bar{x}x \not\stackrel{\sqsubseteq}{\sim}_\emptyset 0$. Similarly, $\bar{x}x \simeq_{\{x,y\}} \bar{y}y$, but $\bar{x}x \not\stackrel{\sqsubseteq}{\sim}_\emptyset \bar{y}y$ and $\bar{y}y \not\stackrel{\sqsubseteq}{\sim}_\emptyset \bar{x}x$. However, $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ if $\text{fn}(P) \cup \text{fn}(Q) \subset \rho_1$.

Theorem 1. *Let $\rho_1 \subset \rho_2$. Then $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$. Further, if $\text{fn}(P) \cup \text{fn}(Q) \subset \rho_1$ then $P \stackrel{\sqsubseteq}{\sim}_{\rho_2} Q$ implies $P \stackrel{\sqsubseteq}{\sim}_{\rho_1} Q$. \square*

We now build on the trace-based characterization of may testing for asynchronous π -calculus presented in [3] to obtain a characterization of may testing in $L\pi_{\neq}$. We note that $L\pi_{\neq}$ is a proper subcalculus of the calculus in [3], i.e. every $L\pi_{\neq}$ term is also an asynchronous π -calculus term, and the transition systems of the two calculi match on the common terms. Following is a summary of the alternate characterization of may testing in asynchronous π -calculus. To avoid infinitary branching, a transition system with synchronous inputs instead of asynchronous inputs is used. To account for asynchrony, the trace semantics is modified using a trace preorder \preceq that is defined as the reflexive transitive closure of the laws shown in table 2. The notation $(\hat{y})\cdot$ is extended to traces as follows.

$$(\hat{y})s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } y \notin \text{fn}(s) \\ s_1.x(y).s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & s = s_1.xy.s_2 \text{ and } y \notin \text{fn}(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

The may preorder $\overset{\sqsubset}{\sim}_\emptyset$ in asynchronous π -calculus is then characterized as: $P \overset{\sqsubset}{\sim}_\emptyset Q$ if and only if $P \overset{s}{\Longrightarrow}$ implies $Q \overset{r}{\Longrightarrow}$ for some $r \preceq s$.

The intuition behind the preorder is that if an observer accepts a trace s , then it also accepts any trace $r \preceq s$. Laws $L1$ - $L3$ capture asynchrony, and $L4$ captures the inability to mismatch names. Laws $L1$ and $L2$ state that an observer cannot force inputs on the process being tested. Since outputs are asynchronous, the actions following an output in a trace exhibited by an observer need not be causally dependent on the output. Hence the observer's outputs can be delayed until a causally dependent action ($L2$), or dropped if there are no such actions ($L1$). Law $L3$ states that an observer can consume its own outputs unless there are subsequent actions that depend on the output. Law $L4$ states that without mismatch an observer cannot discriminate bound names from free names, and hence can receive any name in place of a bound name. The intuition behind the trace preorder is formalized in the following lemma. We note that, since $L\pi_=$ is a subcalculus of asynchronous π -calculus, the lemma also holds for $L\pi_=$.

Lemma 1. *If $P \overset{\bar{s}}{\Longrightarrow}$, then $r \preceq s$ implies $P \overset{\bar{r}}{\Longrightarrow}$.* \square

May testing in $L\pi_=$ is weaker than in asynchronous π -calculus. This is because the locality property reduces the number of observers that can be used to test processes. For example, the following two processes are distinguishable in asynchronous π -calculus but equivalent in $L\pi_=$.

$$P = (\nu x)(!x(z).0|\bar{x}x|\bar{y}x) \quad Q = (\nu x)(!x(z).0|\bar{y}x)$$

The observer $O = y(z).z(w).\bar{\mu}\mu$ can distinguish P and Q in asynchronous π -calculus, but is not a valid $L\pi_=$ term as it violates locality. In fact, no $L\pi_=$ term can distinguish P and Q , because the message $\bar{x}x$ is not observable.

To account for locality we need to consider only the traces that correspond to interaction between $L\pi_=$ processes. Note that the transition system does not by itself account for locality. For instance, in case of the example above, we have $P \xrightarrow{\bar{y}x} \xrightarrow{\bar{x}x}$ although the message $\bar{x}x$ is not observable. To counter this deficiency, we define the notion of *well-formed* traces.

Definition 2. *For a set of names ρ and trace s we define $rcp(\rho, s)$ inductively as*

$$rcp(\rho, \epsilon) = \rho \quad rcp(\rho, s.(\hat{y})xy) = rcp(\rho, s) \quad rcp(\rho, s.(\hat{y})\bar{x}y) = rcp(\rho, s) \cup \hat{y}$$

We say s is ρ -well-formed if $s = s_1.(\hat{y})\bar{x}y.s_2$ implies $x \notin rcp(\rho, s_1)$. We say s is well-formed if it is \emptyset -well-formed. \square

Only ρ -well-formed traces correspond to an interaction between a process and an $L\pi_=$ observer O such that $rcp(O) \cap \rho = \emptyset$. We are now ready to give the alternate characterization of $\overset{\sqsubset}{\sim}_\rho$ in $L\pi_=$.

Definition 3. *We say $P \ll_\rho Q$, if for every ρ -well-formed trace s , $P \overset{s}{\Longrightarrow}$ implies there is $r \preceq s$ such that $Q \overset{r}{\Longrightarrow}$.* \square

To prove the characterization, we define an observer $O(s)$ for a well-formed trace s , such that $P \text{ may } O(s)$ implies $P \xrightarrow{r}$ for some $r \preceq s$. This construction is the same as the one used for asynchronous π -calculus [3].

Definition 4 (canonical observer). For a trace s , we define $O(s)$ as follows:

$$\begin{aligned} O(\epsilon) &\stackrel{\text{def}}{=} \bar{\mu}\mu & O((\hat{y})xy.t) &\stackrel{\text{def}}{=} (\nu\hat{y})(\bar{x}y)O(t) \\ O(\bar{x}(y).t) &\stackrel{\text{def}}{=} x(y).O(t) & O(\bar{x}y.s) &\stackrel{\text{def}}{=} x(u).[u=y]O(s) \quad u \text{ fresh} \quad \square \end{aligned}$$

Note that well-formedness of s guarantees that $O(s)$ is an $L\pi_{=}$ term. Furthermore, it is easy to show that if s is ρ -well-formed, then $\text{rcp}(O(s)) \cap \rho = \emptyset$. Since the canonical observer constructions match and $L\pi_{=}$ is a subcalculus of asynchronous π -calculus, the following lemma proved for asynchronous π -calculus [3], also holds in $L\pi_{=}$.

Lemma 2. For a well-formed trace s , $O(s) \xrightarrow{\bar{\tau}.\bar{\mu}\mu}$ implies $r \preceq s$. \square

Theorem 2 proves the equivalence of $\stackrel{\sqsubseteq}{\sim}_{\rho}$ and \ll_{ρ} in $L\pi_{=}$. The proof is similar to that of Theorem 3 in Section 3.

Lemma 3. Let ρ be a set of names where $\text{rcp}(O) \cap \rho = \emptyset$. Then $P|O \xrightarrow{\bar{\mu}\mu}$ can be unzipped into $P \xrightarrow{s}$ and $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$ for some s that is ρ -well-formed. \square

Theorem 2. $P \stackrel{\sqsubseteq}{\sim}_{\rho} Q$ if and only if $P \ll_{\rho} Q$. \square

3 The Calculus $L\pi$

We now investigate the effect of lack of name matching capability. The rules in table 1 except the *MATCH* rule, constitute the transition system for $L\pi$.

The lack of name matching capability further weakens may testing equivalence. For example, the processes $(\nu u)(\bar{x}u|\bar{x}u)$ and $(\nu u, v)(\bar{x}u|\bar{x}v)$ are equivalent in $L\pi$, but not in $L\pi_{=}$. For the alternate characterization of $P \stackrel{\sqsubseteq}{\sim}_{\rho} Q$, it is too stringent to require that for any trace s that P exhibits, Q exhibits a *single* trace r such that any observer accepting s also accepts r . In fact, there exist $L\pi$ processes P and Q such that $P \stackrel{\sqsubseteq}{\sim}_{\rho} Q$, and if P exhibits s , then Q exhibits *different* traces to satisfy different observers that accept s . For instance, let $P = \bar{x}u_1|\bar{y}u_1|u_1(w).\bar{w}w$ which can exhibit $s = \bar{x}u_1.\bar{y}u_1.u_1(w).\bar{w}w$. The following $L\pi$ observers accept s .

$$\begin{aligned} O_1 &= (\nu w)(x(u).y(v).\bar{x}w|w(v).\bar{\mu}\mu) \\ O_2 &= (\nu w)(x(u).y(v).\bar{v}w|w(v).\bar{\mu}\mu) \\ O_3 &= (\nu w)(x(u).y(v).\bar{u}_1w|w(v).\bar{\mu}\mu) \\ O_4 &= (\nu w)(x(u).y(v).(\bar{v}v|\bar{u}u) | u_1(z).u_1(z).\bar{u}_1w | w(v).\bar{\mu}\mu) \end{aligned}$$

Now, the process $Q = (\nu v)(v(z).v(z').(\bar{x}z|\bar{y}z')|\bar{v}u_1|\bar{v}u_2|!u_2(z).\bar{u}_1z | u_1(w).\bar{w}w)$ can satisfy

$$\begin{array}{ll} O_1 \text{ with } r_1 = \bar{x}u_1.\bar{y}u_2.u_1(w).\bar{w}w & O_2 \text{ with } r_2 = \bar{x}u_2.\bar{y}u_1.u_1(w).\bar{w}w \\ O_3 \text{ with } r_1 \text{ or } r_2, \text{ and} & O_4 \text{ with } r_4 = \bar{x}u_1.\bar{y}u_2.u_2u_2.\bar{u}_1u_2.u_1(w).\bar{w}w \end{array}$$

but cannot exhibit a single trace that can satisfy all four observers. In fact, it is the case that $P \stackrel{\sqsubset}{\sim}_{\emptyset} Q$. Intuitively, although unlike P , Q always exports two different names at x and y , for each possible dataflow pattern of the received names inside an observer that P satisfies, Q exhibits a corresponding trace that can lead the observer to a success.

For the alternate characterization, we define *templates* which are a special kind of traces that can be used to represent dataflows in an observer. A template is a trace in which all outputs are bound. The binding relation between arguments of outputs and their subsequent free occurrences, represents the relevant dependencies between the output argument that is received by an observer and its subsequent use in the observer's computation. For a trace s and set of names ρ , we define a set $T(s, \rho)$ that has a template for each possible dataflow in a computation $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$ with $rcp(O) \cap \rho = \emptyset$. Further, if t represents the dataflow in a computation $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$, then it will be the case that $O \xrightarrow{\bar{t}.\bar{\mu}\mu}$. Thus, if an observer accepts a trace s , then it also accepts a template in $T(s, \rho)$. This template construction essentially captures the effect of lack of match operator. We will show that $P \stackrel{\sqsubset}{\sim}_{\rho} Q$ if and only if for every ρ -well-formed trace s that P exhibits and for each $t \in T(s, \rho)$, Q exhibits some $r \preceq t$.

Following is an informal description of how the set $T(s, \rho)$ can be obtained. Due to the lack of name matching capability, an observer cannot fully discriminate between free inputs. Therefore, a process can satisfy an observer O that exhibits $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$, by replacing free input arguments in \bar{s} with any name as long as it is able to account for changes to the subsequent computation steps that depend on the replaced name. Specifically, suppose $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$ abbreviates the following computation:

$$O \xrightarrow{\bar{s}_1} O_0 \xrightarrow{xy} O_1 \xrightarrow{\beta_1} O_2 \xrightarrow{\beta_2} \dots O_n \xrightarrow{\beta_n} \bar{\mu}\mu$$

Because of the locality property, the name y received in the input may be used only in output terms of O_1 . We call such occurrences of y as *dependent* on the input. During subsequent computation, these output terms may appear either as an output action or are consumed internally. In the latter case, y may be the target of the internal communication, or the argument which in turn may generate further output terms with dependent occurrences of y . Therefore, O can do the following computation when y in the input is replaced with an arbitrary name w :

$$O \xrightarrow{\bar{s}_1} O_0 \xrightarrow{(\hat{w})xw} O_1 \xrightarrow{\gamma_1} O_2 \xrightarrow{\gamma_2} \dots O_n \xrightarrow{\gamma_n} \bar{\mu}\mu$$

where γ_i is obtained from β_i as follows. If β_i is an output action, then γ_i is obtained from β_i by substituting dependent occurrences of y with w . If β_i is an

internal delivery of a message $\bar{y}z$ with target y being a dependent occurrence, there are two possibilities. If z is a private name, then $\gamma_i = \bar{w}(z).yz$ and the subsequent bound output β_j ($j > i$) that exports z for the first time (if any), is changed to a free output. If z is not a private name, then $\gamma_i = \bar{w}z'.yz'$, where z' is w when z is a dependent occurrence of y and z otherwise. For all other cases, $\gamma_i = \beta_i$. Note that, if w is fresh, the input of w could be a bound input.

Clearly, any computation obtained by repeated application of the above construction can be performed by O . In particular, if we always replace free inputs with bound inputs, we will eventually obtain a computation in which all inputs are bound and the construction can not be applied any further. Let $O \xrightarrow{\bar{t}, \bar{\mu}\mu}$ abbreviate a computation thus obtained. The trace t is a template that explicitly represents all dependencies between received names (bound input arguments) and subsequent computation steps (subsequent free occurrences of the argument). The set $T(s, \rho)$ consists of all the templates that can be obtained by this construction starting from arbitrary computations of the form $O \xrightarrow{\bar{s}, \bar{\mu}\mu}$ with $rcp(O) \cap \rho = \emptyset$.

We now formalize the ideas presented above, leading to a direct inductive definition of $T(s, \rho)$. Let $O \xrightarrow{\bar{s}_1} xy \xrightarrow{\bar{s}_2} \bar{\mu}\mu$. We first consider the simple case where $y \notin rcp(O_1)$. Due to locality, in the computation following input xy , there cannot be an internal message delivery with y as the target. Therefore, the following computation is possible. $O \xrightarrow{\bar{s}_1} (\hat{w})xw \xrightarrow{\bar{s}'_2} \bar{\mu}\mu$, where \bar{s}'_2 is obtained from \bar{s}_2 by renaming dependent occurrences of y in output actions to w . Specifically, it does not involve exposing internal actions that use dependent occurrences of y . When the computation steps above are not known, all we can say about \bar{s}'_2 is that it is obtained from \bar{s}_2 by renaming some occurrences of y . Similarly, O'_1 is obtained from O_1 by renaming some occurrences of y in output terms. These relations are formalized in Definition 5 and Lemma 4.

Definition 5 (random output substitution). For $\sigma = \{\tilde{u}/\tilde{v}\}$ we define random output substitution (from now on just random substitution) on process P , denoted by $P[\sigma]$, modulo alpha equivalence as follows. We assume $bn(P) \cap \{\tilde{v}\} = fn(P)\sigma \cap bn(P) = \emptyset$. For a name x we define $x[\sigma] = \{x, x\sigma\}$.

$$\begin{aligned} 0[\sigma] &= \{0\} & (x(y).P)[\sigma] &= \{x(y).P' \mid P' \in P[\sigma]\} \\ (\bar{x}y)[\sigma] &= \{\bar{x}'y' \mid x' \in x[\sigma], y' \in y[\sigma]\} & (P|Q)[\sigma] &= \{P'|Q' \mid P' \in P[\sigma], Q' \in Q[\sigma]\} \\ ((\nu x)P)[\sigma] &= \{(\nu x)P' \mid P' \in P[\sigma]\} & (!x(y).P)[\sigma] &= \{!x(y).P' \mid P' \in P[\sigma]\} \end{aligned}$$

Random substitution on traces is defined modulo equivalence as follows. We assume $bn(s) \cap \{\tilde{v}\} = fn(s)\sigma \cap bn(s) = \emptyset$.

$$\begin{aligned} \epsilon[\sigma] &= \{\epsilon\} & ((\hat{y})\bar{x}y.s)[\sigma] &= \{(\hat{y})\bar{x}'y'.s' \mid s' \in s[\sigma]\} \\ (x(y).s)[\sigma] &= \{x'(y).s' \mid x' \in x[\sigma], s' \in s[\sigma]\} \\ (xy.s)[\sigma] &= \{x'y'.s' \mid x' \in x[\sigma], y' \in y[\sigma], s' \in s[\sigma]\} \end{aligned}$$

We will use $[\tilde{u}/\tilde{v}]$ as a short form for $\{\{\tilde{u}/\tilde{v}\}\}$. □

Lemma 4. *If $P \xrightarrow{\bar{s}}$, $P' \in P[w/y]$, and $y \notin \text{rcp}(P)$, then $P' \xrightarrow{\bar{s}'}$ for some $s' \in s[w/y]$. \square*

Now, suppose $y \in \text{rcp}(O_1)$. Then, in the computation $O \xrightarrow{\bar{s}_1} xy \rightarrow O_1 \xrightarrow{\bar{s}_2} \bar{\mu}\mu$ certain internal transitions may involve a message with a dependent occurrence of y as the target. Then, the following computation which exposes such transitions is also possible $O \xrightarrow{\bar{s}_1} (\hat{w})xw \rightarrow O'_1 \xrightarrow{\bar{s}'_2} \bar{\mu}\mu$ where \bar{s}'_2 is obtained from \bar{s}_2 by not only renaming all dependent occurrences of y in output transitions to w , but also exposing each internal message delivery with a dependent occurrence of y as the message target. If the computation steps are not known, we can only say \bar{s}'_2 is obtained from some $r \in s_2[w/y]$ by exposing arbitrary number of internal transitions at any point in \bar{r} . The relation between s_2 and s'_2 is formalized in Definition 6 and Lemma 5. To account for the situation where an exposed pair of actions $(\hat{z})\bar{w}z.yz$ export a private name z , we need the following function on traces.

$$[\hat{y}]s = \begin{cases} s & \text{if } \hat{y} = \emptyset \text{ or } y \notin n(s) \\ s_1.xy.s_2 & \text{if } \hat{y} = \{y\} \text{ and there are } s_1, s_2, x \text{ s.t.} \\ & s = s_1.x(y).s_2 \text{ and } y \notin n(s_1) \cup \{x\} \\ \perp & \text{otherwise} \end{cases}$$

Definition 6. *For a trace s and a pair of names w, y , the set $F(s, w, y)$ is the smallest set closed under the following rules:*

1. $\epsilon \in F(\epsilon, w, y)$
2. $(\hat{v})uv.s' \in F((\hat{v})uv.s, w, y)$ if $s' \in F(s, w, y)$
3. $(\hat{v})\bar{u}v.s' \in F((\hat{v})\bar{u}v.s, w, y)$ if $s' \in F(s, w, y)$
4. $(\hat{z})wz.\bar{y}z.[\hat{z}]s' \in F(s, w, y)$ if $s' \in F(s, w, y)$ and $[\hat{z}]s' \neq \perp$

Note that $s \in F(s, w, y)$. For a set of traces S , we define $F(S, w, y) = \cup_{s \in S} F(s, w, y)$. \square

Lemma 5. *If $P \xrightarrow{\bar{s}}$ and $P' \in P[w/y]$, then $P' \xrightarrow{\bar{s}'}$ for some $s' \in F(s[w/y], w, y)$. \square*

For a trace s and a set of names ρ , we say s is ρ -normal, if s is normal and $\rho \cap \text{bn}(s) = \emptyset$. Now, let O be an arbitrary observer such that $\text{rcp}(O) \cap \rho = \emptyset$. Suppose

$$O \xrightarrow{\bar{s}_1} xy \rightarrow O_1 \xrightarrow{\bar{s}_2} \bar{\mu}\mu$$

where $\bar{s}_1.xy.\bar{s}_2$ is ρ -normal. If $y \in \rho$ or y is the argument of a bound input in \bar{s}_1 , then by locality $y \notin \text{rcp}(O_1)$. Otherwise, since O is arbitrary, it is possible that $y \in \text{rcp}(O_1)$. From this observation, we have that for an arbitrary observer O such that $\text{rcp}(O) \cap \rho = \emptyset$, if O accepts the ρ -normal trace $s_1.\bar{x}y.s_2$, then O also accepts $s_1.(\hat{w})\bar{x}w.s'_2$ where w is an arbitrary name and $s'_2 \in s_2[w/y]$ if $y \in \rho$ or y is the argument of a bound output in s_1 , and $s'_2 \in F(s_2[w/y], w, y)$ otherwise. $T(s, \rho)$ is precisely the set of all traces with no free outputs, that can be obtained by repeated application of this reasoning. $T(s, \rho)$ is formally defined in Definition 7.

Definition 7. For a trace s and a set of names ρ , the set of templates $T(s, \rho)$ is defined modulo alpha equivalence as follows. We assume that s is ρ -normal.

1. $\epsilon \in T(\epsilon, \rho)$.
2. $(\hat{y})xy.s' \in T((\hat{y})xy.s, \rho)$ if $s' \in T(s, \rho)$
3. $\bar{x}(y).s' \in T(\bar{x}(y).s, \rho)$ if $s' \in T(s, \rho \cup \{y\})$
4. $\bar{x}(w).s' \in T(\bar{x}(w).s, \rho)$ if w fresh, $s' \in T(s'', \rho \cup \{w\})$, and

$$s'' \in \begin{cases} s[w/y] & \text{if } y \in \rho \\ F(s[w/y], w, y) & \text{if } y \notin \rho \end{cases}$$

The reader may check that if $t \in T(s, \rho)$, then $s \preceq t$ using only L3 and L4. \square

Lemma 6. If $P \xrightarrow{\bar{s}}$ and $\rho \cap \text{rcp}(P) = \emptyset$, then there is $t \in T(s, \rho)$ such that $P \xrightarrow{\bar{t}}$. \square

Lemma 7 states that template construction in Definition 7 preserves ρ -well-formedness.

Lemma 7. If s is ρ -well-formed then every $t \in T(s, \rho)$ is ρ -well-formed. \square

We are now ready to give the alternate characterization of $\stackrel{\square}{\sim}_{\rho}$ in $L\pi$.

Definition 8. We say $P \ll_{\rho} Q$ if for every ρ -well-formed trace s , $P \xrightarrow{s}$ implies for each $t \in T(s, \rho)$ there is $r \preceq t$ such that $Q \xrightarrow{r}$. \square

For $t \in T(s, \rho)$, where s is a ρ -well-formed trace, let $O(t)$ be the canonical observer as defined in Definition 4. By Lemma 7, since s is ρ -well-formed t is also ρ -well-formed. Hence $O(t)$ satisfies the locality property, and $\text{rcp}(O(t)) \cap \rho = \emptyset$. Further, since t is a template, the case $t = \bar{x}(y).t'$ does not arise in the construction of the observer. Hence $O(t)$ is an $L\pi$ term. Since $L\pi$ is a subcalculus of asynchronous π -calculus, Lemma 1 holds for $L\pi$. Further, since the canonical observer construction is unchanged, the following lemma (which is a weaker version of Lemma 2) holds for $L\pi$.

Lemma 8. For $t \in T(s, \rho)$, where s is a ρ -well-formed trace, $O(t) \xrightarrow{\bar{r}. \bar{\mu}}$ implies $r \preceq t$. \square

Lemma 3 holds for $L\pi$ with formally the same proof. Now, we are ready to prove that \ll_{ρ} is an alternate characterization of $\stackrel{\square}{\sim}_{\rho}$.

Theorem 3. $P \stackrel{\square}{\sim}_{\rho} Q$ if and only if $P \ll_{\rho} Q$.

Proof. (if) Let $P \ll_{\rho} Q$ and $P \underline{\text{may}} O$ for an observer O such that $\text{rcp}(O) \cap \rho = \emptyset$. From $P \underline{\text{may}} O$ we have $P|O \xrightarrow{\bar{\mu}}$. By Lemma 3, this computation can be unzipped into $P \xrightarrow{s}$ and $O \xrightarrow{\bar{s}. \bar{\mu}}$ for some ρ -well-formed trace s . From Lemma 1 and 6 we deduce there is a $t' \in T(s, \mu\mu, \rho)$ such that $r' \preceq t'$ implies $O \xrightarrow{\bar{r}'}$. It is easy to show that $t' \in T(s, \mu\mu, \rho)$ implies $t' = t.\mu\mu$ for some $t \in T(s, \rho)$.

From $P \ll_{\rho} Q$, there is a trace $r \preceq t$ such that $Q \xrightarrow{r}$. Moreover, $r \preceq t$ implies $r.\mu\mu \preceq t.\mu\mu = t'$. Therefore, $O \xrightarrow{\bar{r}.\bar{\mu}\mu}$. We can zip this with $Q \xrightarrow{r}$ to obtain $Q|O \xrightarrow{\bar{\mu}\mu}$, which means $Q \underline{\text{may}} O$.

(only if): Let $P \sqsubset_{\rho} Q$ and $P \xrightarrow{s}$ where s is ρ -well-formed. We have to show for every $t \in T(s, \rho)$ there is a trace $r \preceq t$ such that $Q \xrightarrow{r}$. It is easy to show that if $t \in T(s, \rho)$, then $O(t) \xrightarrow{\bar{s}.\bar{\mu}\mu}$. This can be zipped with $P \xrightarrow{s}$ to get $P|O(t) \xrightarrow{\bar{\mu}\mu}$, that is $P \underline{\text{may}} O(t)$. From $P \sqsubset_{\rho} Q$, we have $Q \underline{\text{may}} O(t)$ and therefore $Q|O(t) \xrightarrow{\bar{\mu}\mu}$. This can be unzipped into $Q \xrightarrow{r}$ and $O(t) \xrightarrow{\bar{r}.\bar{\mu}\mu}$. From Lemma 8, it follows that $r \preceq t$. \square

For finitary processes we can obtain a simpler characterization based on a modified version of Definition 7 as given below.

Definition 9. For a trace s and a set of names ρ , the set $T_f(s, \rho)$ is defined inductively using the first three rules of Definition 7 and the following two.

- 4 $\bar{x}(w).s' \in T_f(\bar{x}y.s, \rho)$ if $y \in \rho, w$ fresh, $s' \in T_f(s'', \rho \cup \{w\})$, and $s'' \in s[w/y]$
- 5 $\bar{x}y.s' \in T_f(\bar{x}y.s, \rho)$ if $y \notin \rho$, and, $s' \in T_f(s, \rho)$ \square

The main difference from Definition 7 is that output arguments y that are not in ρ are not converted to bound arguments. According to rule 4 of Definition 7, such conversions introduce arbitrary number of pairs of input/output actions. But, since the length of traces that a finite process can exhibit is bounded, the only way the process can exhibit a trace $r \preceq t$ for each of the resulting templates, is by emitting the same name y , so that $L4$ and $L3$ can be applied to annihilate some of these input/output pairs. The following lemma helps formalize this observation.

Lemma 9. For a trace s , a set of names ρ , and a prefixed closed set R of traces with bounded length, if for every $t \in T(s, \rho)$ there exists $r \in R$ such that $r \preceq t$, then for every $t_f \in T_f(s, \rho)$ there exists $r \in R$ such that $r \preceq t_f$. \square

Using this lemma, we can show that for finitary processes we can use $T_f(s, \rho)$ in Definition 8 instead of $T(s, \rho)$. The resulting characterization is equivalent to the earlier one for the following reason. Suppose $P \xrightarrow{s}$ implies, for every $t \in T(s, \rho)$, there exists $r \preceq t$ such that $Q \xrightarrow{r}$. Then, let R be the set of all traces that Q exhibits. Note that R is prefix closed. Further, since Q is finite, there is a bound on the length of traces in R . By Lemma 9, for every $t_f \in T_f(s, \rho)$, there exists $r \preceq t_f$ such that $Q \xrightarrow{r}$. Conversely, suppose $P \xrightarrow{s}$ implies that for every $t \in T_f(s, \rho)$ there exists $r \preceq t$ such that $Q \xrightarrow{r}$. It is easy to verify that for every $t \in T(s, \rho)$ there exists a $t_f \in T_f(s, \rho)$ such that $t_f \preceq t$, where the relation can be derived using only $L3$ and $L4$. From transitivity of \preceq , it follows that $P \xrightarrow{s}$ implies for every $t \in T(s, \rho)$ there exists $r \preceq t$ such that $Q \xrightarrow{r}$.

4 An Axiomatization of Finitary $L\pi_{=}$ and $L\pi$

We first give a sound and complete proof system for \sqsubseteq_{ρ} for the finitary fragment of $L\pi$, i.e. for $L\pi$ processes that do not use replication. A simple adaptation of the proof system gives us one for finitary $L\pi_{=}$. The proof system consists of the laws given in table 3 and the rules for reflexivity and transitivity. For a finite index set I , we use the macro $\sum_{i \in I} P_i$ to denote, $(\nu u)((|_{i \in I} u(u).P_i)|\bar{u}u)$ for u fresh if $I \neq \emptyset$, and 0 otherwise. For an index set that is a singleton, we omit I and simply write $\sum P$ instead of $\sum_{i \in I} P$. We let the variable G range over processes of form $\sum_{i \in I} P_i$. We write $\sum_{i \in I} P_i + \sum_{j \in J} P_j$ to denote $\sum_{k \in I \sqcup J} P_k$. We write \sqsubseteq as a shorthand for \sqsubseteq_{\emptyset} , and $=$ for $=_{\emptyset}$. Random input substitution on processes $P[w/y]_i$ is defined similar to random output substitution (Definition 5), except that only the occurrences of y at the subject of input prefixes in P are randomly substituted with w .

While axioms $A1$ to $A19$ all hold in asynchronous π -calculus [3], axioms $A20$ and $A21$ are unique to $L\pi$. $A20$ captures the fact that a message targeted to a name that an environment is prohibited from listening to, cannot escape to the environment. The axiom states that there are only two ways such a message can be handled in the next transition step: it can be consumed internally or delayed for later. The axiom also accounts for delaying the message forever by including dropping of the message as one of the possibilities. As an application of this axiom, if $x \in \rho$, we can prove $\bar{x}y \sqsubseteq_{\rho} 0$ as follows. For w fresh,

$$\begin{aligned}
\bar{x}y &\sqsubseteq_{\rho} \bar{x}y(\nu w)(w(w).0) && (A3, A11, I1) \\
&\sqsubseteq_{\rho} (\nu w)(\bar{x}y|w(w).0) && (A8) \\
&\sqsubseteq_{\rho} (\nu w)(\sum w(w).0 + \sum w(w).\bar{x}y + \sum 0) && (A20, I1) \\
&\sqsubseteq_{\rho} \sum (\nu w)(w(w).0) + \sum (\nu w)w(w).\bar{x}y + \sum (\nu w)0 && (A7) \\
&\sqsubseteq_{\rho} 0 && (A1, A11, A14, I3)
\end{aligned}$$

Axiom $A21$ captures the effect of lack of match operator. It is directly motivated from rule 4 of Definition 9 for template construction.

The inference rules extend the rules for asynchronous π -calculus to handle parameterization of the may preorder. In fact, the rules for asynchronous π -calculus presented in [3] can be obtained by setting $\rho = \emptyset$ in $I1$, $I2$ and $I3$. $I4$ is a new rule that is motivated by Theorem 1. We make a few remarks about $I1$ which is significantly different from its analogue for asynchronous π -calculus. First, using $\bar{x}y \sqsubseteq_{\{x\}} 0$ (proved above) and $I1$, we get $(\nu x)\bar{x}y \sqsubseteq (\nu x)0$, and by axiom $A19$ we have $(\nu x)0 \sqsubseteq 0$. Therefore, $(\nu x)\bar{x}y \sqsubseteq 0$. Note the use of the ability to contract the parameter ρ of the may preorder after applying a restriction. Second, the following example illustrates the necessity of the side condition $rcp(R) \cap \rho = \emptyset$ for composition: $\bar{x}y \sqsubseteq_{\{x\}} 0$ but not $\bar{x}y|x(y).\bar{y}y \sqsubseteq_{\{x\}} x(y).\bar{y}y$, for the LHS can satisfy the observer $y(u).\bar{u}\mu$ and the RHS can not.

The soundness of rules $I1$ - $I4$ can be easily proved directly from Definition 1. We only show the argument for $I1$, which is given in Lemma 10. Soundness of axioms $A1$ - $A21$ is easy to check. For $A1$ - $A19$, whenever $P \sqsubseteq Q$, we have $P \xrightarrow{s} \dots$,

Table 3. Inference rules and axioms for $L\pi$

<i>I1</i>	if $P \sqsubseteq_{\rho} Q$ and $rcp(R) \cap \rho = \emptyset$, then $(\nu x)P \sqsubseteq_{\rho - \{x\}} (\nu x)Q$, $P R \sqsubseteq_{\rho} Q R$.	
<i>I2</i>	if for each $z \in fn(P, Q)$ $P\{z/y\} \sqsubseteq_{\rho} Q\{z/y\}$ then $x(y).P \sqsubseteq_{\rho} x(y).Q$	
<i>I3</i>	if for each $i \in I$ $P_i \sqsubseteq_{\rho} \sum_{j \in J} Q_{ij}$ then $\sum_{i \in I} P_i \sqsubseteq_{\rho} \sum_{i \in I, j \in J} Q_{ij}$	
<i>I4</i>	if $\rho_1 \subset \rho_2$ and $P \sqsubseteq_{\rho_1} Q$ then $P \sqsubseteq_{\rho_2} Q$.	
<i>A1</i>	$G + G = G$	<i>A3</i> $P 0 = P$
<i>A2</i>	$G \sqsubseteq G + G'$	<i>A4</i> $P Q = Q P$
		<i>A5</i> $(P Q) R = P (Q R)$
<i>A6</i>	Let $G = \sum_{i \in I} \alpha_i.P_i$ and $G' = \sum_{j \in J} \alpha'_j.P'_j$ where each α_i (resp. α'_j) does not bind free names of G' (resp. G). Then $G G' = \sum_{i \in I} \alpha_i.(P_i G') + \sum_{j \in J} \alpha'_j.(G P'_j)$	
<i>A7</i>	$(\nu x)(\sum_{i \in I} P_i) = \sum_{i \in I} (\nu x)P_i$	
<i>A8</i>	$(\nu x)(P Q) = P (\nu x)Q$	$x \notin n(P)$
<i>A9</i>	$(\nu x)(\bar{x}y \alpha.P) = \alpha.(\nu x)(\bar{x}y P)$	$x \notin n(\alpha)$
<i>A10</i>	$(\nu x)(\bar{x}y x(z).P) = (\nu x)(P\{y/z\})$	
<i>A11</i>	$(\nu x)(y(z).P) = \begin{cases} y(z).(\nu x)P & \text{if } x \neq y, x \neq z \\ 0 & \text{if } x = y \end{cases}$	
<i>A12</i>	$\bar{x}y \sum_{i \in I} P_i = \sum_{i \in I} (\bar{x}y P_i)$	$I \neq \emptyset$
<i>A13</i>	$\alpha.\sum_{i \in I} P_i = \sum_{i \in I} \alpha.P_i$	$I \neq \emptyset$
<i>A14</i>	$P = \sum P$	
<i>A15</i>	$x(y).(\bar{u}v P) \sqsubseteq \bar{u}v x(y).P$	$y \neq u, y \neq v$
<i>A16</i>	$P\{y/z\} \sqsubseteq \bar{x}y x(z).P$	
<i>A17</i>	$x(u).y(v).P \sqsubseteq y(v).x(u).P$	$u \neq y, u \neq v$
<i>A18</i>	$x(y).(\bar{x}y P) \sqsubseteq P$	$y \notin n(P)$
<i>A19</i>	$(\nu x)P \sqsubseteq P\{y/x\}$	
<i>A20</i>	If $x \in \rho$, $w \neq x$ and $w \neq y$, then $\bar{x}y z(w).P \sqsubseteq_{\rho} \sum z(w).(\bar{x}y P) + \sum z(w).P + \sum Q$, where $Q = \begin{cases} P\{y/w\} & \text{if } x = z \\ 0 & \text{otherwise} \end{cases}$	
<i>A21</i>	$\bar{x}y P \sqsubseteq_{\rho} (\nu w)(\bar{x}w \sum_{P' \in P[w/y]_i} P')$	w fresh, $y \in \rho$.

implies $Q \xrightarrow{r}$ such that $r \preceq s$. For *A20*, both LHS and RHS exhibit the same ρ -well-formed traces. Proof of soundness of axiom *A21* is more involved, and is established in Lemma 10. The reader can verify that *A20* and *A21* would also be sound as equalities. For instance, the converse of *A21* can be shown using *A19*, *A1*, and *I1*.

Lemma 10.

1. If $P \overset{\rho}{\sim} Q$ and $rcp(R) \cap \rho = \emptyset$, then $(\nu x)P \overset{\rho - \{x\}}{\sim} (\nu x)Q$, $P|R \overset{\rho}{\sim} Q|R$.
2. For $y \in \rho$ and w fresh, $\bar{x}y|P \overset{\rho}{\sim} (\nu w)(\bar{x}w|\sum_{P' \in P[w/y]_i} P')$. □

We prove that the laws presented constitute a complete proof system for finite processes, i.e. for finite processes P, Q , $P \sqsubseteq_\rho Q$ if $P \stackrel{\rho}{\approx} Q$. Inspired by the alternate characterization, the proof relies on existence of canonical forms for processes.

Definition 10. *If s is a template, then we call \bar{s} a cotemplate. Thus, a cotemplate is a trace with no free inputs. If s is well-formed, we say \bar{s} is cowell-formed.*

1. For a cowell-formed cotemplate s , the process $e(s)$ is defined inductively as follows.

$$\begin{aligned} e(\epsilon) &\stackrel{def}{=} 0 & e(\bar{x}y.s') &\stackrel{def}{=} \bar{x}y|e(s') \\ e(\bar{x}(y).s') &\stackrel{def}{=} (\nu y)(\bar{x}y|e(s')) & e(x(y).s') &\stackrel{def}{=} x(y).e(s') \end{aligned}$$

Note that cowell-formedness of s implies that $e(s)$ is an $L\pi$ term. From now on we follow the convention that whenever we write $e(s)$ it is implicit that s is a cowell-formed cotemplate.

2. The process $\sum_{s \in S} e(s)$, for a set of traces S , is said to be in canonical form. □

The proof of completeness relies on the following four lemmas. The first lemma states that every process has an equivalent canonical form.

Lemma 11. *For a process P there is a canonical form C such that $P = C$.* □

Lemma 12. *(1) If $e(s) \xrightarrow{r}$, then $e(r) \sqsubseteq e(s)$. (2) If $s \preceq r$ then $e(r) \sqsubseteq e(s)$.* □

The proofs of the two lemmas above are formally the same as the proofs of the corresponding lemmas for asynchronous π -calculus [3]. This is because, the proofs of $P = C$ and $e(r) \sqsubseteq e(s)$ constructed using the proof system of [3], can be transformed into proofs in our proof system. This claim is justified by the following three observations. First, every $L\pi$ term is also an asynchronous π -calculus term. Second, starting from $L\pi$ terms, every term that appears in the proofs of [3] is also an $L\pi$ term. (Note that any summation that appears is finite and can be interpreted as our macro.) Finally, every axiom and inference rule used in their proof is derivable in our proof system.

Lemma 13. *Let R contain all the cowell-formed cotemplates r such that $e(s) \xrightarrow{r}$ and r is ρ -well-formed. Then $e(s) \sqsubseteq_\rho \sum_{r \in R} e(r)$.* □

Lemma 14. $e(s) \sqsubseteq_\rho \sum_{t \in T_f(s, \rho)} e(t)$. □

Note that the summations in the two lemmas above are finite because R and $T_f(s, \rho)$ are finite modulo alpha equivalence. For instance, finiteness of R is a direct consequence of the following two observations. For every $r \in R$, we have $fn(r) \subset fn(e(s))$, and since $e(s)$ is a finite process, the length of traces in R is bounded. We are now ready to establish the completeness of the proof system.

Theorem 4. *For finite $L\pi$ processes P, Q and a set of names ρ , $P \sqsubseteq_\rho Q$ if and only if $P \overset{\rho}{\sim} Q$.*

Proof. The only-if part follows from the soundness of laws in table 3. We prove the if part. By Lemma 11 and soundness of the proof system, without loss of generality, we can assume that both P and Q are in canonical form, i.e. P is of form $\sum_{s \in S_1} e(s)$ and Q is of form $\sum_{s \in S_2} e(s)$. Using Lemma 13, and laws $I3$, $A1$, we get $P \sqsubseteq_\rho \sum_{r \in R} e(r)$, where R is the set of ρ -well-formed cowell-formed cotemplates that P exhibits. Using Lemma 14 and laws $I3$, $A1$, we have $\sum_{r \in R} e(r) \sqsubseteq_\rho \sum_{t \in T} e(t)$, where $T = \cup_{r \in R} T_f(r, \rho)$. Note that since every $r \in R$ is a cotemplate, so is every $t \in T$. Let $t \in T$. Then $t \in T_f(r, \rho)$ for some ρ -well-formed r that P exhibits. Using the characterization of may preorder based on $T_f(r, \rho)$, we have $P \overset{\rho}{\sim} Q$ implies there is $s' \preceq t$ such that $Q \overset{s'}{\implies}$. It follows that for some $s \in S_2$, $e(s) \overset{s'}{\implies}$. Since $Q \overset{s'}{\implies}$, by locality, s' is cowell-formed. From the facts that $s' \preceq t$ and t is a cotemplate, it follows that s' is a cotemplate. Then by Lemma 12.2 and law $I4$, $e(t) \sqsubseteq_\rho e(s')$. Further, by Lemma 12.1 and law $I4$, $e(s') \sqsubseteq_\rho e(s)$. Hence by transitivity of \sqsubseteq_ρ , we have $e(t) \sqsubseteq_\rho e(s)$. Since $t \in T$ is arbitrary, using laws $I3$, $A1$, and $A2$, we deduce $\sum_{t \in T} e(t) \sqsubseteq_\rho \sum_{s \in S_2} e(s)$. The result follows from transitivity of \sqsubseteq_ρ . \square

We obtain a complete proof system for $L\pi_=_$ by dropping axiom $A21$ and adding the following two for the match operator: $[x = x]P = P$, and $[x = y]P = 0$ if $x \neq y$. Completeness of the resulting proof system can be established by simple modifications to the proofs above.

5 Related Work

We have provided an alternate characterization of a parameterized version of may testing for asynchronous variants of π -calculus with locality and no name matching. We have exploited the characterizations to obtain complete axiomatizations of the may preorder for finitary fragments of the calculi. Our results extend the ones obtained by Boreale, De Nicola, and Pugliese for asynchronous π -calculus [3]. We now compare our work with other related research.

Hennessy and Rathke [6] study typed versions of three behavioral equivalences, namely may and must equivalences, and barbed congruence in a typed π -calculus where the type system allows names to be tagged with input/output capabilities. In the typed calculus, one can express processes that selectively distribute different capabilities on names. The locality property is a special case in which only the output capability on names can be passed. A novel labeled transition system is defined over configurations which are process terms with two typed environments, one that constrains the process and the other the environment. It is shown that the standard definitions of trace and acceptance sets [5] defined over the new transition system characterize may and must preorders respectively. In comparison to our work, the typed calculus of Hennessy and Rathke is synchronous and is equipped with name matching, whereas $L\pi_=_$ is asynchronous,

and $L\pi$ is asynchronous with no name matching. Further, $L\pi_{=}$ has no capability types and hence we obtain a simpler characterization of may testing for it, which is based on the usual early style labeled transition system. Finally, we have also given an axiomatization of may testing, which is not pursued by Hennessy and Rathke.

There have been extensive investigations of bisimulation-based behavioral equivalences on $L\pi$ and related variants of π -calculus, which are properly contained in may testing which is trace based. Merro and Sangiorgi [9] investigate barbed congruence in $L\pi$, and show that a variant of asynchronous early bisimulation provides an alternate characterization for the congruence. Boreale and Sangiorgi [4] study typed barbed equivalence for typed (synchronous) π -calculus with capability types and no name matching, and show that the equivalence is characterized by a typed variant of bisimulation. Merro [8] characterizes barbed congruence in the more restricted setting of asynchronous π -calculus with no name matching (no capability types, and no locality in particular). He defines synonymous bisimulation and shows that it characterizes barbed congruence in this setting.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. 223
- [2] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for Asynchronous π -Calculus. In *Proceedings of CONCUR '96*. Springer-Verlag, 1996. LNCS 1119. 223
- [3] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes, 2002. 223, 224, 225, 226, 228, 234, 236, 237
- [4] Michele Boreale and D. Sangiorgi. Bisimulation in Name Passing Calculi without Matching. *Proceedings of LICS*, 1998. 223, 238
- [5] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988. 225, 237
- [6] Matthew Hennessy and Julian Rathke. Typed behavioral equivalences for processes in the presence of subtyping. Technical report, University of Sussex, Computer Science, March 2001. 237
- [7] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Fifth European Conference on Object-Oriented Programming*, July 1991. LNCS 512, 1991. 223
- [8] M. Merro. On equators in asynchronous name-passing calculi without matching. *Electronic Notes in Theoretical Computer Science*, 27, 1999. 238
- [9] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443. 223, 238
- [10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992. 223
- [11] R. De Nicola and M. Hennessy. Testing equivalence for processes. In *Theoretical Computer Science*, volume 34, pages 83–133, 1984. 223
- [12] B. C. Pierce and D. N. Turner. Pict: A programming Language Based on the π -Calculus. Technical Report CSCI-476, Indiana University, March 1997. 223

- [13] Prasanna Thati, Reza Ziaei, and Gul Agha. A theory of may testing for asynchronous calculi with locality and no name matching. Technical Report UIUCDCS-R-2002-2277, University of Illinois at Urbana Champaign, May 2002.
[224](#)

Equational Axioms for Probabilistic Bisimilarity

Luca Aceto¹, Zoltán Ésik^{2*}, and Anna Ingólfssdóttir¹

¹ **BRICS** (Basic Research in Computer Science)
Centre of the Danish National Research Foundation
Department of Computer Science, Aalborg University
Fr. Bajersvej 7E, 9220 Aalborg Ø, Denmark
luca@cs.auc.dk
annai@cs.auc.dk

² Department of Computer Science, University of Szeged
P.O.B. 652, 6701 Szeged, Hungary
esik@inf.u-szeged.hu

Abstract. This paper gives an equational axiomatization of probabilistic bisimulation equivalence for a class of finite-state agents previously studied by Stark and Smolka ((2000) *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pp. 571–595). The axiomatization is obtained by extending the general axioms of iteration theories (or iteration algebras), which characterize, among others, the equational properties of the fixed point operator on (ω -)continuous or monotonic functions, with three axiom schemas that express laws that are specific to probabilistic bisimilarity. Hence probabilistic bisimilarity (over finite-state agents) has an equational axiomatization relative to iteration algebras.

1 Introduction

Probabilistic variations on process algebras have been extensively studied in the literature, and concepts from concurrency theory have been extended to these languages and their underlying probabilistic models—see, e.g., [15] for a survey and many references to the original literature. As part of this research effort to lift process algebraic results to the probabilistic setting, several notions of probabilistic behavioural equivalences and preorders have been proposed in the literature over various models of probabilistic processes, and have been axiomatized over fragments of probabilistic process algebras. Works presenting complete axiomatizations of probabilistic semantic theories for processes are, e.g., [2,4,14,16,17,21,24]. Amongst the aforementioned references, the studies [14,16,17,24] consider languages with finite-state recursive definitions, and offer implicational proof systems for probabilistic bisimulation equivalence. (Indeed, the language TPCCS studied by Hansson in [14] involves a combination of time and probabilities.)

* Supported in part by the National Foundation of Hungary for Scientific Research (grants T30511 and T35163). This work was partly carried out while this author was visiting professor at BRICS, Aalborg University.

In this paper, we contribute to the quest for complete axiomatizations of behavioural equivalences for probabilistic processes by offering a purely equational axiomatization of probabilistic bisimulation equivalence for a class of finite-state agents previously studied by, e.g., Stark and Smolka in [24]. The axiomatization is obtained by extending the general axioms of *iteration theories* (or *iteration algebras*) [6,12], which characterize, among others, the equational properties of the fixed point operator on (ω -)continuous or monotonic functions, with three axiom schemas that express laws that are specific to probabilistic bisimilarity. Hence probabilistic bisimilarity (over finite-state agents) has an equational axiomatization relative to iteration algebras; this axiomatization is finite relative to iteration algebras if we allow for the use of equation schemas.

Historically, an implicational axiom system for probabilistic bisimilarity over finite-state processes was first proposed in [17], where its soundness and completeness were announced for a class of finite-state probabilistic agents with rational probabilities. The unpublished dissertation [16] offered a proof of the soundness and completeness result announced in [17], showing that the assumption of rational probabilities could be dropped. However, according to Stark and Smolka [24], some of the soundness proofs in [16] were flawed, and [24] is apparently the first study which gave a full proof of the soundness and completeness of the axiom system from [17] for a CCS-like language with (possibly unguarded) finite-state recursive definitions. In the meantime, Hansson [14] offered an implicational proof system for bisimilarity over the fragment of his language TPCCS with guarded finite-state recursion. Amongst all these original references, our technical developments in this paper have mostly been influenced by those in [24].

We believe that the results presented in this paper improve upon those previous axiomatizations of probabilistic bisimilarity for the language we consider. First of all, in light of the simplicity and foundational role played by equational logic, it is natural to look for purely equational axiomatizations of algebras of processes—as done, for instance, in the ACP family of process algebras (see, e.g., [13] for a textbook presentation). Moreover, whenever finite-state processes are concerned, implicational axiom systems based on variants on the unique fixed point induction rule for guarded terms, like those presented in the classic paper [22] and the aforementioned references, are somewhat unsatisfactory as they afford very few models. A classic example of this phenomenon is present in the long history of the quest for equational axiomatizations of the algebra of regular languages. Salomaa gave two complete axiomatizations of the algebra of regular languages in [23]. However, one of them contains an infinitary rule, and, as argued by Kozen in [18], the other is not sound in most common interpretations of regular expressions (such as binary relations) because it uses a version of the unique fixed point rule. Implicational axiomatizations for the equational theory of regular languages that are sound over a wealth of important nonstandard interpretations that arise in computer science have been given by Krob in [19] and Kozen in [18]. A purely equational axiomatization of regular languages has been offered by Krob in [19].

Giving a relative axiomatization of probabilistic bisimilarity with respect to iteration algebras has also the benefit of separating the general (embodied by the equations of iteration theories) from the specific (expressed by the equations that describe properties of probabilistic bisimilarity proper). This separation of concerns has at least two benefits. First of all, as a relative axiomatization of probabilistic bisimilarity can be given by adding three axiom schemas to those of iteration algebras, it follows that the non-finite axiomatizability of the equational theory of probabilistic bisimilarity is due to that of iteration algebras (see, e.g., [10]). Secondly, any advance in the equational axiomatization of iteration algebras would yield an improved equational axiomatization for probabilistic bisimilarity.

That standard bisimulation equivalence is finitely axiomatizable relative to iteration algebras was shown in [6, Chapter 13].

2 Preliminaries

In order to make the paper self-contained, we now briefly review the basic notions from [24] and of iteration algebras that will be needed in this study. Moreover, we extend the operational semantics from *op. cit.* and the definition of probabilistic bisimilarity so that they apply to the whole language of probabilistic terms directly.

2.1 Probabilistic Finite-State Terms and Probabilistic Bisimilarity

We begin by presenting the syntax and the operational semantics of the language of finite-state probabilistic terms that will be studied in the remainder of the paper. Our presentation is based on that in [24], to which the reader is referred for more details and background information.

We use \mathbf{Var} to stand for a countably infinite set of *agent variables*, ranged over by x, y, w, z possibly subscripted and/or superscripted, and \mathbf{Act} to denote a nonempty collection of *atomic actions*, ranged over by a . The meta-variable α stands for an element of the set $\mathbf{Act} \cup \mathbf{Var}$.

The syntax of *probabilistic terms* (over \mathbf{Var} and \mathbf{Act}) is defined as follows:

$$t ::= x \mid at \mid t_p + t \mid \mu x.t \text{ ,}$$

where $x \in \mathbf{Var}$, $a \in \mathbf{Act}$ and p is a real number in the open interval $(0, 1)$. The notions of *free* and *bound variables* are defined in the standard way—with $\mu x._$ as a binding construct—, and a variable x is *guarded* in term t if every free occurrence of x in t occurs within a subterm of the form at' . If $\bar{x} = (x_1, \dots, x_n)$ is a vector of distinct variables, we shall sometimes write $t(\bar{x})$ to denote the fact that every free variable of t is in \bar{x} . (As usual, the free variables of t need not contain all of the variables in \bar{x} .) A term is *closed* if it does not contain any free variable. Throughout the paper, we consider two terms as syntactically identical if they are equal up to renaming of their bound variables. If $\bar{x} = (x_1, \dots, x_n)$ is

Table 1. Transition rules for terms ($\alpha \in \text{Act} \cup \text{Var}$)

$\frac{t \xrightarrow{\alpha} t'}{t_p + u \xrightarrow{\alpha} t'}$	$\frac{u \xrightarrow{\alpha} u'}{t_p + u \xrightarrow{\alpha} u'}$	$at \xrightarrow{a} t$	$x \xrightarrow{x} \perp$	$\frac{t[\bar{t}/\bar{x}] \xrightarrow{\alpha} t'}{\mu x.t \xrightarrow{\alpha} t'}$
---	---	------------------------	---------------------------	--

a vector of distinct variables, $\bar{t} = (t_1, \dots, t_n)$ is a vector of terms, and t is a term, then $t[\bar{t}/\bar{x}]$ denotes the term that results by substituting each occurrence of x_i in t with t_i . The definition of substitution in the presence of binders like μ is standard, and is therefore omitted. When writing terms, we assume that the scope of a $\mu x.$ extends to the right as far as possible. In the remainder of the paper, we use \perp as an abbreviation for the closed term $\mu x.x$.

Following the approach adopted by Stark and Smolka in [24], we define the operational semantics for probabilistic terms in two steps. First, we give the transitions of terms using standard structural operational semantics. (Cf. Table 2.1 for the rules. Note that, in our formulation of the operational semantics for terms, the statement $t \xrightarrow{x} \perp$ means that the variable x occurs unguarded in the term t .) Next, we incorporate information about the probability of occurrence of transitions into the operational semantics by associating, with each triple (t, α, u) consisting of terms t, u and $\alpha \in \text{Act} \cup \text{Var}$, a transition probability $\text{prob}(t, \alpha, u) \in [0, 1]$. Following Stark and Smolka, we shall use the more suggestive notation $\text{prob}(t \xrightarrow{\alpha} u)$ in lieu of $\text{prob}(t, \alpha, u)$. For the sake of completeness, we recall that the function prob is defined as the least solution (over the complete partial order of the set of all functions mapping triples (t, α, u) to the real numbers in the interval $[0, 1]$, ordered pointwise) of the recursive equation

$$\text{prob} = \mathcal{P}(\text{prob}) \text{ ,}$$

where \mathcal{P} is given by:

$$\begin{aligned} \mathcal{P}(\text{prob})(at \xrightarrow{\alpha} u) &= \begin{cases} 1 & \text{if } a = \alpha \text{ and } t = u \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{P}(\text{prob})(x \xrightarrow{\alpha} u) &= \begin{cases} 1 & \text{if } x = \alpha \text{ and } \perp = u \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{P}(\text{prob})(t_1_p + t_2 \xrightarrow{\alpha} u) &= p \cdot \text{prob}(t_1 \xrightarrow{\alpha} u) + (1 - p) \cdot \text{prob}(t_2 \xrightarrow{\alpha} u) \\ \mathcal{P}(\text{prob})(\mu x.t \xrightarrow{\alpha} u) &= \text{prob}(t[\mu x.t/x] \xrightarrow{\alpha} u) \text{ .} \end{aligned}$$

For example, we have that, for every $p \in (0, 1)$,

$$\text{prob}(\mu x.a \perp_p + x \xrightarrow{a} \perp) = 1 = \text{prob}(\mu x.x_p + y \xrightarrow{y} \perp) \text{ .}$$

We refer the interested reader to [24] for more information on the definition of the probability assigning function prob , and on its connections with the structural

operational semantics in Table 2.1. Here we limit ourselves to recalling that $\text{prob}(t \xrightarrow{\alpha} u)$ is positive if, and only if, $t \xrightarrow{\alpha} u$ can be inferred from the rules in Table 2.1 (cf. [24, Lem. 2.1]), and that, for every term t , set S of terms and $\alpha \in \text{Act} \cup \text{Var}$, the summation

$$\sum_{u \in S} \text{prob}(t \xrightarrow{\alpha} u)$$

converges to a value between 0 and 1 (cf. [24, Propn. 2.2]). Following Stark and Smolka, we use $\text{prob}(t \xrightarrow{\alpha} S)$ to denote this value.

Remark 1. Unlike Stark and Smolka in the developments in [24], we have presented the operational semantics for terms, possibly containing free variables, in the language we study. In our operational semantics, $\text{prob}(t \xrightarrow{x} \perp)$ measures the “probability of unguardedness” of variable x in term t . Note that $\text{prob}(t \xrightarrow{x} \perp)$ does *not* coincide with $\text{unguard}_t(x)$, a measure of the total probability assigned to unguarded occurrences of variable x in term t defined by Stark and Smolka in [24, Page 587]. For example, if t is the term $\mu x.x_p + y$, with $p \in (0, 1)$, then

$$\text{unguard}_t(y) = (1 - p) \neq 1 = \text{prob}(t \xrightarrow{y} \perp) .$$

The reason for using the definition given here instead of the one by Stark and Smolka is that, unlike the one given in *op. cit.*, the probability of unguardedness of variables it gives is stable under behavioural equivalence.

In what follows, when we refer to the probabilistic labelled transition system determined by a term, we mean the fragment of the transition system generated by the aforementioned operational semantics that can be reached from it.

Notation 1 *In what follows, we shall sometimes use the suggestive notation $t \xrightarrow{p, \alpha} u$ to denote the fact that $\text{prob}(t \xrightarrow{\alpha} u) = p$ and $p > 0$, i.e., that t can perform α with positive probability p , and become u in doing so.*

The notion of behavioural equivalence we shall consider in this paper is probabilistic bisimilarity. This we now proceed to present, extended to the whole set of terms.

Definition 1. *A probabilistic bisimulation is an equivalence relation \mathcal{R} over terms that satisfies the following condition:*

Whenever $t\mathcal{R}u$, then for all $\alpha \in \text{Act} \cup \text{Var}$ and all equivalence classes S of \mathcal{R} we have that

$$\text{prob}(t \xrightarrow{\alpha} S) = \text{prob}(u \xrightarrow{\alpha} S) .$$

Two terms t and u are probabilistically bisimilar, written $t \stackrel{\text{pr}}{\sim} u$, iff there is a probabilistic bisimulation that relates them.

The relation $\stackrel{\text{pr}}{\sim}$ will henceforth be referred to as *probabilistic bisimilarity*.

Example 1. It is not hard to see that the least equivalence relation containing all the pairs of the form $(\mu x.x_p + y, y)$, with $p \in (0, 1)$, is a probabilistic bisimulation. Thus, for every $p \in (0, 1)$, it holds that $\mu x.x_p + y \stackrel{\text{pr}}{\sim} y$.

Remark 2. The definition of probabilistic bisimilarity given above is an extension to the whole set of terms of the original one by Larsen and Skou in [20]. The definitions of this relation given in, e.g., [4,15] are based on an extension of equivalence relations to probability distributions. The two definitions coincide.

The import of the following theorem is that the explicit definition of probabilistic bisimilarity for the whole language of probabilistic terms we have presented coincides with the one given by Stark and Smolka in [24].

Theorem 1. *Let $t(\bar{x})$ and $u(\bar{x})$ be terms. Then $t(\bar{x}) \stackrel{\text{pr}}{\sim} u(\bar{x})$ iff for all vectors of closed terms \bar{t} , it holds that $t[\bar{t}/\bar{x}] \stackrel{\text{pr}}{\sim} u[\bar{t}/\bar{x}]$.*

A proof of the following result, due to Stark and Smolka, may be found in [24, Sect. 4].

Proposition 1. *The relation of probabilistic bisimilarity is a congruence over the language of finite-state probabilistic terms. Thus, whenever t and u are probabilistically bisimilar, so are the terms*

- at and au , for every action a ;
- $t_p + t'$ and $u_p + t'$, for every term t' ;
- $t'_p + t$ and $t'_p + u$, for every term t' ; and
- $\mu x.t$ and $\mu x.u$, for every variable x .

2.2 Axioms of Iteration Algebras

The axioms of *iteration algebras* (or iteration theories) [6] capture the equational properties of the fixed point operation (be it least, unique, initial, etc.). Several (conditional) equational bases of identities for iteration algebras have been studied in the literature (cf., e.g., *op. cit.* and the references [7,11]). In this study, we shall specifically consider an equational axiomatization of iteration algebras obtained by the second author in [12]. This equational basis for iteration algebras consists of the so-called *Conway equations* [6,7]

$$\mu x.t[t'/x] = t[\mu x.t'[t/x]/x] \quad (1)$$

$$\mu x.t[x/y] = \mu x.\mu y.t \quad , \quad (2)$$

and of a set of equations containing one equation for each finite (simple) group. (Group equations for the language of μ -terms were introduced in [12] as a generalization of Conway's group equations for regular languages, cf. [8]. The completeness of the Conway equations and the group equations for iteration algebras extends Krob's result in [19], where he confirmed a long standing conjecture of Conway's [8] about the axiomatization of the equational theory of regular sets.)

In the setting of monotonic and continuous functions, equations (1)–(2) above were established by de Bakker, Bekič, Scott and others (see, e.g., [3,5]), and are sometimes referred to as the *composition identity* (also known as the *rolling identity*) and the *diagonal identity* (also known as the *double-dagger identity*), respectively. Note that the classic *fixed point equation*, viz.

$$\mu x.t = t[\mu x.t/x] \quad , \quad (3)$$

is the instance of the composition identity obtained by taking t' to be the variable x .

To define the group equations, we need to extend the μ -notation to term vectors $\vec{t} = (t_1, \dots, t_n)$. (Henceforth, we shall consider term vectors as ordinary terms.) Let $\vec{x} = (x_1, \dots, x_n)$ be a vector of distinct variables. When $n = 1$, we use $\mu\vec{x}.\vec{t}$ to denote the term vector of dimension one whose unique component is $\mu x_1.t_1$. (We identify any term vector of dimension one with its component.) If $n > 1$, let $\vec{x}' = (x_1, \dots, x_{n-1})$, $\vec{t}' = (t_1, \dots, t_{n-1})$ and $\vec{s} = \vec{t}'[\mu x_n.t_n/x_n]$. (Substitution into a term vector is defined componentwise.) We define

$$\mu\vec{x}.\vec{t} \stackrel{\text{def}}{=} (\mu\vec{x}'.\vec{s}, (\mu x_n.t_n)[\mu\vec{x}'.\vec{s}/\vec{x}']) \quad .$$

The definition is motivated by the Bekič-de Bakker-Scott rule [3,5].

Suppose now that (G, \cdot) is a finite group of order n , whose elements are the integers in the set $[n] = \{1, \dots, n\}$. Given a vector $\vec{x} = (x_1, \dots, x_n)$ of distinct variables and an integer $i \in [n]$, define $i \cdot \vec{x} = (x_{i,1}, \dots, x_{i,n})$. Thus, $i \cdot \vec{x}$ is obtained by permuting the components of \vec{x} according to the i th row of the multiplication table of G . The *group equation associated with G* is

$$(\mu\vec{x}.(t[1 \cdot \vec{x}/\vec{x}], \dots, t[n \cdot \vec{x}/\vec{x}]))_1 = \mu y.t[y/x_1, \dots, y/x_n] \quad , \quad (4)$$

where t is any μ -term, y is a variable, and where

$$(\mu\vec{x}.(t[1 \cdot \vec{x}/\vec{x}], \dots, t[n \cdot \vec{x}/\vec{x}]))_1$$

is the first component of the term vector $\mu\vec{x}.(t[1 \cdot \vec{x}/\vec{x}], \dots, t[n \cdot \vec{x}/\vec{x}])$.

3 An Equational Axiomatization of Probabilistic Bisimilarity

The main result of [24] is a complete implicational axiomatization for probabilistic bisimilarity over probabilistic finite-state terms. The axiomatization offered by Stark and Smolka in *op. cit.* consists of the fixed point equation (3) (axiom R1 in *op. cit.*), the unique fixed point rule for guarded terms and of the equations in Table 3. In equation S2, the condition $C(p, q, r, s)$ holds true whenever $p = rs$, $(1-p)q = (1-r)s$ and $(1-s) = (1-p)(1-q)$. We recall that the *unique fixed point rule* for guarded terms, axiom R3 in [24], states that:

From $t = u[t/x]$, where all occurrences of x in u are guarded, infer that $t = \mu x.u$.

Table 2. Stark and Smolka's equational axioms for probabilistic bisimilarity

S1	$x_p + y = y_{1-p} + x$	
S2	$x_p + (y_q + z) = (x_r + y)_s + z$	if $C(p, q, r, s)$ holds
S3	$x_p + x = x$	
R2	$\mu x.t_p + x = \mu x.t$	

The main aim of this study is to show that the equational laws of probabilistic bisimilarity over finite-state probabilistic terms have a natural axiomatization over the equations of iteration algebras. To this end, we shall consider the purely equational axiom system Ax obtained by extending the equational basis of iteration algebras consisting of (1), (2) and the group equations (4) with axioms S1 and S2 from Table 3, and the following equation

$$\mu x.(x_p + y) = y \quad . \quad (5)$$

The above equation expresses a strengthened form of idempotence of the $_p +$ operation. In fact, equation S3 in Table 3 follows from it and the fixed point equation (3) thus:

$$y = \mu x.x_p + y = (\mu x.x_p + y)_p + y = y_p + y \quad .$$

Moreover, in the presence of axiom S1 and of the diagonal equation, (5) proves equation R2 in Table 3. Indeed:

$$\begin{aligned} \mu x.t_p + x &= \mu x.\mu z.t_p + z && (z \text{ fresh}) \\ &= \mu x.((\mu z.y_p + z)[t/y]) && (y \text{ fresh}) \\ &= \mu x.(y[t/y]) \\ &= \mu x.t \quad . \end{aligned}$$

The remainder of this paper will be devoted to a proof of the following soundness and completeness theorem:

Theorem 2. *The axiom system Ax completely axiomatizes probabilistic bisimilarity over the language of finite-state probabilistic terms, i.e., for all terms t, u , the equivalence $t \approx u$ holds iff the equality $t = u$ is provable using the equations in Ax .*

4 Soundness

Our first step towards a proof of Theorem 2 will be to show the following theorem, to the effect that the axiom system Ax is sound with respect to probabilistic bisimilarity.

Theorem 3 (Soundness). *For all terms t and u , if Ax proves that $t = u$ then $t \stackrel{\text{pr}}{\approx} u$.*

Since Stark and Smolka have proven in [24, Sect. 4] that their axiom system is sound with respect to $\stackrel{\text{pr}}{\approx}$, the above theorem follows from the following result to the effect that the implicational axiom system due to Stark and Smolka entails Ax :

Theorem 4. *Every equation in Ax can be proven from the axiom system for probabilistic bisimilarity due to Stark and Smolka.*

Remark 3. Another, possibly more standard, approach to showing the soundness of Ax with respect to probabilistic bisimilarity is to identify the probabilistic transition systems associated with the left- and right-hand sides of all of the equations in Ax , and to exhibit appropriate probabilistic bisimulations between them—as we did for axiom (5) in Example 1. We have, however, plumped for an equational approach to such a proof because it can be better formalized for complex equations like the Conway and group equations. Moreover, Theorem 4 gives more information than the mere soundness of Ax with respect to probabilistic bisimilarity. For example, it entails that the models of the axiom system by Stark and Smolka are also models of Ax .

It is clear that axiom (5) is derivable from the axiom system by Stark and Smolka by using axioms S1 and R2 in Table 3, and the fixed point equation. It is much less clear that so are the Conway and group equations. The interested reader may find the details of the non-trivial equational arguments used in the proofs of these equations from the axiom system by Stark and Smolka in [1].

5 Normal Forms

The next step in the proof of our main result is the isolation of a suitable notion of normal form for finite-state probabilistic terms. Normal forms have a direct interpretation as finite-state probabilistic transition systems, and this will be crucial in establishing the completeness of our axiom system. We prove that, modulo Ax , every term is provably equal to one in normal form (Theorem 5).

We begin by introducing a useful notation that will help clarify the connection between terms in normal form, and the probabilistic transition systems they denote. In this notation, we use the notion of *stochastic vector*, which is a vector of real numbers in the interval $[0, 1]$ that sum up to 1.

Notation 2 *Let $\{(p_i, t_i) \mid i \in [k]\}$ be a nonempty set of pairs, where each t_i is a term, and (p_1, \dots, p_k) is a stochastic vector. The notation*

$$\sum_{i \in [k]} p_i \cdot t_i$$

is defined recursively as follows:

$$\sum_{i \in [k]} p_i \cdot t_i \stackrel{\text{def}}{=} \begin{cases} t_k & \text{if } p_k = 1 \\ \sum_{i \in [k-1]} p_i \cdot t_i & \text{if } p_k = 0 \\ (\sum_{i \in [k-1]} (p_i / (1 - p_k)) \cdot t_i) \cdot_{1-p_k} t_k & \text{if } p_k \in (0, 1) \end{cases} .$$

In what follows, we shall often write $p_1 \cdot t_1 + \dots + p_k \cdot t_k$ in lieu of $\sum_{i \in [k]} p_i \cdot t_i$, and t instead of $1 \cdot t$.

For example, we write

$$\frac{1}{4} \cdot a \perp + \frac{1}{2} \cdot x + \frac{1}{4} \cdot \perp$$

for the term

$$(a \perp \cdot_{1/3} + x) \cdot_{3/4} \perp .$$

Definition 2. A simple term in the variables $\bar{x} = (x_1, \dots, x_n)$ and parameters $\bar{y} = (y_1, \dots, y_m)$ is a term of the form

$$p_1 \cdot t_1 + \dots + p_k \cdot t_k + q_1 \cdot y_1 + \dots + q_m \cdot y_m + q \cdot \perp , \quad (6)$$

where:

- $k, m \geq 0$,
- $(p_1, \dots, p_k, q_1, \dots, q_m, q)$ is a stochastic vector,
- for every $i \in [k]$, the term t_i is of the form $a x_\ell$ for some action a and variable $x_\ell \in \{x_1, \dots, x_n\}$, and
- for every $i_1, i_2 \in [k]$, if $i_1 \neq i_2$ then $t_{i_1} \neq t_{i_2}$.

(If k or m are 0, then the corresponding part of a simple term is missing.)

A summand of a simple term of the form (6) is a subterm of the form $p \cdot t$, where p is positive.

Note that, in light of the last condition above on simple terms, modulo S1 and S2, axiom S3 in Table 3 cannot be applied to a simple term when used as a rewrite rule from left to right. For example, the term

$$\frac{2}{3} \cdot ax + \frac{1}{3} \cdot ax$$

is not simple, but using S3 from left to right it can be proven equal to the simple term $1 \cdot ax$, that is to ax .

Definition 3 (Normal Form Terms). A normal form term in the parameters $\bar{y} = (y_1, \dots, y_m)$ is a term

$$\mu \bar{x} . \bar{t} = \mu(x_1, \dots, x_n) \cdot (t_1, \dots, t_n) ,$$

where each t_i ($i \in [n]$) is a simple term in the variables $\bar{x} = (x_1, \dots, x_n)$ and parameters $\bar{y} = (y_1, \dots, y_m)$.

Intuitively, the i th component of a term vector $\mu(x_1, \dots, x_n).(t_1, \dots, t_n)$ is the i th component of a distinguished solution of the list of equations

$$\begin{aligned} x_1 &= t_1 \\ &\vdots \\ x_n &= t_n \end{aligned} .$$

We shall sometimes identify a normal form term with its corresponding list of equations. The main reason for doing so is that the list of equations associated with a normal form term can be naturally viewed as a kind of probabilistic transition system. Indeed, the set of states of such a probabilistic transition system may be taken to be the integers in the set $[n]$ —here, integer i stands for the i th component of the term vector determined by the list of equations—plus a distinguished \perp state. If the simple term t_i has the form (6), the set of transitions out of state $i \in [n]$ is defined as follows:

- for every $j \in [n]$, there is a transition $i \xrightarrow{p, a} j$ if, and only if, $p \cdot ax_j$ is a summand of the simple term t_i —that is, when $t_i \xrightarrow{p, a} x_j$ holds; and
- for every variable x , there is a transition $i \xrightarrow{p, x} \perp$ if, and only if, $p \cdot x$ is a summand of the simple term t_i —that is, when $t_i \xrightarrow{p, x} \perp$ holds.

We use $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ to denote this probabilistic transition system, and say that a normal form $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ is *accessible* if every state of $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ is reachable from state 1.

Proposition 2. *The transition system $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ is probabilistically bisimilar to $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$, for every normal form $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$.*

Remark 4. Actually, the transition systems $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ and $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ are not just probabilistically bisimilar, but also strongly equivalent—in the sense that the two transition systems “unfold to the same tree” (cf. [9]).

The following normal form theorem is a version of Milner’s equational characterization of regular CCS process, cf. [22]. A version of Milner’s equational characterization theorem for the finite-state probabilistic terms we consider has been given by Stark and Smolka in [24, Thm. 2].

Theorem 5. *For every term $t(\bar{y})$, there is an accessible normal form term $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ such that the equality $t(\bar{y}) = (\mu\bar{x}.\bar{t}(\bar{x}, \bar{y}))_1$ is provable from the axioms in Ax.*

From now on, we shall assume that terms in normal form are accessible. Moreover, we equip the transition system $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ with the initial state 1. Probabilistic bisimulations between two such transition systems will relate their initial states.

6 Completeness

In Sect. 4, we established the soundness of our axiom system Ax with respect to probabilistic bisimilarity by showing that the axiom system proposed by Stark and Smolka in [24] entails it. We now aim at proving that, like the one by Stark and Smolka, our axiom system is complete with respect to probabilistic bisimilarity. This is the import of the following theorem:

Theorem 6 (Completeness). *For all terms t and u , if $t \stackrel{\text{pr}}{\approx} u$ then Ax proves that $t = u$.*

The remainder of this section will be devoted to a proof of the above result. Apart from the normal form theorem (Theorem 5), our proof of the completeness theorem consists of two main ingredients.

- First we shall show that, in a suitable technical sense, two probabilistically bisimilar normal forms are structurally related.
- Next, we use the structural relationship between probabilistically bisimilar normal forms to show that two equivalent normal forms are provably equal using Ax . It is this final step of the proof that relies upon the group equations and the full power of the axioms of iteration theories.

We now proceed to study the structural relation that exists between probabilistically bisimilar normal forms. In the remainder of this section, equality of terms is modulo the axioms S1–S3 in Table 3.

Definition 4. *Let $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$ be two vectors, each consisting of distinct variables. Let, furthermore, ρ be a function mapping $[n]$ to $[k]$. For every term $t(\bar{x}, \bar{y})$, we define the term $t \circ \rho$ thus:*

$$t \circ \rho \stackrel{\text{def}}{=} t[z_{\rho(1)}/x_1, \dots, z_{\rho(n)}/x_n] .$$

If $\bar{t} = (t_1, \dots, t_n)$ is a vector of terms over variables $\bar{x} = (x_1, \dots, x_n)$, then we write $\bar{t} \circ \rho$ for the vector of terms $(t_1 \circ \rho, \dots, t_n \circ \rho)$.

If $\bar{u} = (u_1, \dots, u_k)$ is a vector of terms, then we write $\rho \circ \bar{u}$ for the vector of terms $(u_{\rho(1)}, \dots, u_{\rho(n)})$.

Note that when the components of \bar{t} are simple, then, modulo S1–S3, so are the components of $\bar{t} \circ \rho$.

Example 2. Consider the vectors of simple terms $\bar{t} = (t_1, t_2, t_3)$ and $\bar{u} = (u_1, u_2)$ over variables $\bar{x} = (x_1, x_2, x_3)$, where

$$\begin{aligned} t_1 &= \frac{1}{3} \cdot ax_2 + \frac{2}{3} \cdot ax_3 \\ t_2 &= ax_2 \\ t_3 &= \frac{1}{2} \cdot ax_2 + \frac{1}{2} \cdot ax_3 \\ u_1 &= ax_1 \quad \text{and} \\ u_2 &= \text{an arbitrary simple term} . \end{aligned}$$

Let ρ map each $i \in [3]$ to 1. Then, modulo the axioms S1–S3 in Table 3,

$$\bar{t} \circ \rho = (ax_1, ax_1, ax_1) = \rho \circ \bar{u} .$$

Whenever $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ are two terms in normal form over variables $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$, respectively, we say that a function $\rho : [n] \rightarrow [k]$ determines a probabilistic bisimulation from the probabilistic transition system $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ to $\text{ts}(\bar{u}(\bar{z}, \bar{y}))$ if, and only if, the least equivalence relation over the disjoint union of $[n]$ and $[k]$ containing the graph of ρ is a probabilistic bisimulation.

Proposition 3. *Let $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ be two terms in normal form over variables $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$, respectively. A function $\rho : [n] \rightarrow [k]$ determines a probabilistic bisimulation from $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ to $\text{ts}(\bar{u}(\bar{z}, \bar{y}))$ if, and only if, the following two conditions hold:*

1. $\rho(1) = 1$, and
2. $\bar{t} \circ \rho = \rho \circ \bar{u}$ modulo axioms S1–S3 in Table 3.

Notation 3 *In what follows, we shall write $\bar{t}(\bar{x}, \bar{y}) \xrightarrow[\rho]{} \bar{u}(\bar{z}, \bar{y})$ when ρ determines a probabilistic bisimulation from $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ to $\text{ts}(\bar{u}(\bar{z}, \bar{y}))$.*

Proposition 4. *Let $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ be two terms in normal form over variables $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$ respectively. Then the probabilistic transition system $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ is probabilistically bisimilar to $\text{ts}(\bar{u}(\bar{x}, \bar{y}))$ if, and only if, there are a normal form $\mu\bar{w}.\bar{r}(\bar{w}, \bar{y})$ (over variables $\bar{w} = (w_1, \dots, w_\ell)$), and functions $\rho : [\ell] \rightarrow [n]$ and $\tau : [\ell] \rightarrow [k]$ such that*

$$\bar{t}(\bar{x}, \bar{y}) \xleftarrow[\rho]{} \bar{r}(\bar{w}, \bar{y}) \xrightarrow[\tau]{} \bar{u}(\bar{z}, \bar{y}) .$$

In light of Propositions 2–4, in order to prove the completeness of our axiom system with respect to probabilistic bisimilarity it would be sufficient to show that:

Proposition 5. *Whenever $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ are two terms in normal form over variables $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$, respectively, and $\rho : [n] \rightarrow [k]$ is a function meeting the constraints in the statement of Proposition 3, then the axiom system Ax proves that $(\mu\bar{x}.\bar{t})_1 = (\mu\bar{z}.\bar{u})_1$.*

Indeed, using the above statement, we can prove Theorem 6 thus:

Proof of Theorem 6: Let t and u be two probabilistically bisimilar terms. By Theorem 5, there are normal forms $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ such that Ax proves that $t = (\mu\bar{x}.\bar{t}(\bar{x}, \bar{y}))_1$ and $u = (\mu\bar{z}.\bar{u}(\bar{z}, \bar{y}))_1$. By Proposition 2 and the soundness of the axiom system Ax with respect to probabilistic bisimilarity, we have that the probabilistic transition system $\text{ts}(\bar{t}(\bar{x}, \bar{y}))$ is probabilistically bisimilar to $\text{ts}(\bar{u}(\bar{z}, \bar{y}))$. By Proposition 4, it follows that there are a normal form $\mu\bar{w}.\bar{r}(\bar{w}, \bar{y})$, and functions ρ and τ such that

$$\bar{t}(\bar{x}, \bar{y}) \xleftarrow[\rho]{} \bar{r}(\bar{w}, \bar{y}) \xrightarrow[\tau]{} \bar{u}(\bar{z}, \bar{y}) .$$

By Proposition 5, Ax proves that

$$(\mu\bar{x}.\bar{t}(\bar{x}, \bar{y}))_1 = (\mu\bar{w}.\bar{r}(\bar{w}, \bar{y}))_1 = (\mu\bar{z}.\bar{u})_1 .$$

By transitivity, we thus obtain $t = u$, which was to be shown. \square

Our order of business is now to prove Proposition 5. In fact, we establish the following strengthening of that statement:

Proposition 6. *Let $\mu\bar{x}.\bar{t}(\bar{x}, \bar{y})$ and $\mu\bar{z}.\bar{u}(\bar{z}, \bar{y})$ be two terms in normal form over variables $\bar{x} = (x_1, \dots, x_n)$ and $\bar{z} = (z_1, \dots, z_k)$, respectively. Assume that $\rho : [n] \rightarrow [k]$ is a function meeting the constraints in the statement of Proposition 3. Then Ax proves that*

$$\mu\bar{x}.\bar{t} = \rho \circ (\mu\bar{z}.\bar{u}) . \quad (7)$$

To prove the above result, we rely on the fact that (7) is implied by the identities of iteration algebras if the term vectors $\bar{t}(\bar{x}, \bar{y})$ and $\bar{u}(\bar{z}, \bar{y})$ satisfy the following condition: There exist a vector $\bar{w} = (w_1, \dots, w_\ell)$ of variables, term vectors $r_j(\bar{w}, \bar{y})$, $j \in \rho([n])$, and functions $\rho_i : [\ell] \rightarrow [n]$, $i \in [n]$, such that for all $i \in [n]$ and $j \in [k]$ with $\rho(i) = j$ it holds that

$$r_j \circ \rho_i = t_i$$

modulo axioms S1–S3 in Table 3, where r_j and t_i denote the j th component of \bar{r} and the i th component of \bar{t} , respectively. (See the *generalized commutative identity* on p. 138 in [6].) We establish this condition by using:

Lemma 1. *Let c be a positive real number. Suppose furthermore that $\bar{c}_i = (c_{i1}, \dots, c_{ik_i})$ ($i \in [n]$) are nonempty sequences of positive real numbers all summing up to c . Then there is a sequence (d_1, \dots, d_ℓ) of positive real numbers such that, for every $i \in [n]$, there are positive integers $\ell_1 < \ell_2 < \dots < \ell_{k_i-1} < \ell$ with*

$$\begin{aligned} c_{i1} &= d_1 + \dots + d_{\ell_1} \\ c_{i2} &= d_{\ell_1+1} + \dots + d_{\ell_2} \\ &\vdots \\ c_{ik_i} &= d_{(\ell_{k_i-1}+1)} + \dots + d_\ell . \end{aligned}$$

References

1. L. ACETO, Z. ÉSIK, AND A. INGÓLFSÓTTIR, *Equational axioms for probabilistic bisimilarity (preliminary report)*, Report RS-02-6, BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation), February 2002. Available at the URL <http://www.brics.dk/RS/02/6/index.html>. 247
2. J. C. M. BAETEN, J. A. BERGSTRA, AND S. A. SMOLKA, *Axiomatizing probabilistic processes: ACP with generative probabilities*, Inform. and Comput., 121 (1995), pp. 234–255. 239

3. J. W. DE BAKKER AND D. SCOTT, *A theory of programs*, Technical Report, IBM Laboratory, Vienna, 1969. 245
4. E. BANDINI AND R. SEGALA, *Axiomatizations for probabilistic bisimulation*, in Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP) 2001, vol. 2076 of Lecture Notes in Computer Science, Springer-Verlag, July 2001, pp. 370–381. 239, 244
5. H. BEKIČ, *Definable operations in general algebras, and the theory of automata and flowcharts*, Technical Report, IBM Laboratory, Vienna, 1969. 245
6. S. L. BLOOM AND Z. ÉSIK, *Iteration theories*, Springer-Verlag, Berlin, 1993. 240, 241, 244, 252
7. ———, *The equational logic of fixed points*, Theoret. Comput. Sci., 179 (1997), pp. 1–60. 244
8. J. H. CONWAY, *Regular Algebra and Finite Machines*, Mathematics Series (R. Brown and J. De Wet eds.), Chapman and Hall, London, United Kingdom, 1971. 244
9. C. C. ELGOT, S. L. BLOOM, AND R. TINDELL, *On the algebraic structure of rooted trees*, J. Comput. System Sci., 16 (1978), pp. 362–399. 249
10. Z. ÉSIK, *Independence of the equational axioms for iteration theories*, J. Comput. System Sci., 36 (1988), pp. 66–76. 241
11. ———, *Completeness of Park induction*, Theoret. Comput. Sci., 177 (1997), pp. 217–283. Mathematical foundations of programming semantics (Manhattan, KS, 1994). 244
12. ———, *Group axioms for iteration*, Inform. and Comput., 148 (1999), pp. 131–180. 240, 244
13. W. J. FOKKINK, *Introduction to Process Algebra*, Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, 2000. 240
14. H. HANSSON, *Time and Probability in Formal Design of Distributed Systems*, vol. 1 of Real-Time Safety Critical Systems, Elsevier, 1994. 239, 240
15. B. JONSSON, W. YI, AND K. G. LARSEN, *Probabilistic extensions of process algebras*, in Handbook of Process Algebra, North-Holland, Amsterdam, 2001, pp. 685–710. 239, 244
16. C.-C. JOU, *Aspects of Probabilistic Process Algebra*, PhD thesis, SUNY at Stony Brook, Stony Brook, New York, 1991. 239, 240
17. C.-C. JOU AND S. A. SMOLKA, *Equivalences, congruences, and complete axiomatizations for probabilistic processes*, in Proceedings CONCUR 90, Amsterdam, J. Baeten and J. Klop, eds., vol. 458 of Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 367–383. 239, 240
18. D. KOZEN, *A completeness theorem for Kleene algebras and the algebra of regular events*, Inform. and Comput., 110 (1994), pp. 366–390. 240
19. D. KROB, *Complete systems of B-rational identities*, Theoretical Comput. Sci., 89 (1991), pp. 207–343. 240, 244
20. K. G. LARSEN AND A. SKOU, *Bisimulation through probabilistic testing*, Information and Computation, 94 (1991), pp. 1–28. 244
21. ———, *Compositional verification of probabilistic processes*, in Proceedings CONCUR 92, Stony Brook, NY, USA, W. R. Cleaveland, ed., vol. 630 of Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 456–471. 239
22. R. MILNER, *A complete inference system for a class of regular behaviours*, J. Comput. System Sci., 28 (1984), pp. 439–466. 240, 249
23. A. SALOMAA, *Two complete axiom systems for the algebra of regular events*, J. Assoc. Comput. Mach., 13 (1966), pp. 158–169. 240

24. E. W. STARK AND S. A. SMOLKA, *A complete axiom system for finite-state probabilistic processes*, in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, MIT Press, Cambridge, MA, 2000, pp. 571–595. [239](#), [240](#), [241](#), [242](#), [243](#), [244](#), [245](#), [247](#), [249](#), [250](#)

Bisimulation by Unification*

Paolo Baldan¹, Andrea Bracciali², and Roberto Bruni²

¹ Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italia

² Dipartimento di Informatica, Università di Pisa, Italia

baldan@dsi.unive.it

braccia@di.unipi.it bruni@di.unipi.it

Abstract. We propose a methodology for the analysis of open systems based on process calculi and bisimilarity. Open systems are seen as coordinators (i.e. terms with place-holders), that evolve when suitable components (i.e. closed terms) fill in their place-holders. The distinguishing feature of our approach is the definition of a symbolic operational semantics for coordinators that exploits spatial/modal formulae as labels of transitions and avoids the universal closure of coordinators w.r.t. all components. Two kinds of bisimilarities are then defined, called *strict* and *large*, which differ in the way formulae are compared. Strict bisimilarity implies large bisimilarity which, in turn, implies the one based on universal closure. Moreover, for process calculi in suitable formats, we show how the symbolic semantics can be defined constructively, using unification. Our approach is illustrated on a toy process calculus with CCS-like communication within ambients.

1 Introduction

The ever increasing usage and development of mobile devices raise the need of formal models for open systems, where components can be dynamically connected to interact with network services. Process calculi (PC) are often instrumental in focusing on certain aspects like communications, distribution and causal dependencies. However, PC techniques are mostly devised for the study of *components* (i.e. closed terms of the calculus) rather than *coordinators* (i.e. contexts with holes marked by process variables).

In particular, while the operational semantics and several equivalences have been often defined for components (e.g., based on either bisimulation or traces or testing), their extensions to coordinators usually require additional efforts. Roughly, an equivalence \approx defined on components can be lifted to coordinators by letting $C[X_1, \dots, X_n] \approx D[X_1, \dots, X_n]$ when $C[p_1, \dots, p_n] \approx D[p_1, \dots, p_n]$ for all components p_1, \dots, p_n . In the case of bisimulation, this means that the coalgebraic techniques applicable to components fall short for coordinators, since the definition involves universal quantification on components. Instead, a symbolic

* Research supported by the IST programme on FET-GC Projects AGILE, MYTHS and SOCS.

technique for allowing contexts to “bisimulate without instantiation” would ease the analysis and verification of coordinators’ properties.

This issue finds its dual formulation in the contextual closure needed when the bisimilarity on components \sim is not a congruence and one defines the largest congruence \simeq contained in \sim (by letting $p \simeq q$ if for all contexts $C[\cdot]$, identity included, $C[p] \sim C[q]$ holds). Note that in general \simeq is not a bisimulation. The largest congruence which is also a bisimulation is called *dynamic bisimilarity* and it is defined by allowing context closure at each bisimulation step [22].

To avoid universal quantification on contexts, several authors—e.g., Sewell in [25], Leifer and Milner in [20]—propose a symbolic transition system for components whose labels are the “minimal” contexts needed to the component for evolving. A transition

$$p \xrightarrow{C[\cdot, X_1, \dots, X_n]} D[X_1, \dots, X_n]$$

means that $C[p, p_1, \dots, p_n]$ can reduce in one step to $D[p_1, \dots, p_n]$, and that C is strictly necessary to perform the step. However, in their symbolic systems, though transitions always depart from components, they may lead also to contexts (like D above) and therefore bisimulation must be defined on contexts via universal quantification over all possible closed instantiations. Thus, the problem of universal quantification is shifted from contexts to components. Finding a sound and efficient way to face this problem is the goal of our contribution.

Symbolic Bisimulation. It is nowadays commonly accepted that the operational semantics of most process calculi can be conveniently expressed by exploiting two basic ingredients, according to Plotkin’s SOS recipe [23]: the structure of the components and the behaviour of their subcomponents. Thus, a PC definition usually involves a process signature Σ , a structural equivalence \equiv on process terms, and a labelled transition system (LTS) specified through a set of inductive (structural) proof rules.¹

It follows that the behaviour of a coordinator can depend: (1) on the spatial structure of the components that are inserted/substituted/connected in/with it; (2) on their behaviour, i.e. on the actions that can be observed.

The first attempt could be to define a transition system whose states are coordinators and whose arcs are labelled with the components that allow coordinators to evolve. But this would result in a too large transition system, making verification difficult.

To attack this problem, reducing the size of the transition system, we propose to borrow formulae from a suitable logic for expressing the most general class of processes with whom each coordinator can react. This leads us to the notion of *symbolic transition system* (STS), whose states are coordinators and whose transitions have the shape

$$C[X_1, \dots, X_n] \xrightarrow{\varphi_1, \dots, \varphi_n} D[Y_1, \dots, Y_m]$$

¹ Reduction semantics can be obviously recasted in LTS’s as the special case with a unique label.

meaning that the step can be performed by $C[p_1, \dots, p_n]$ whenever $p_i \models \varphi_i$, for $i \in [1, n]$.

The logic where the formulae φ_i 's live and the notion of satisfaction \models must be of course targeted to the PC under study. In general, the logic may involve both spatial and temporal aspects of components, i.e. it can be a *spatial logic* [8,11].

Fixed an STS, two kinds of bisimilarities \sim_{strict} and \sim_{large} , referred to, respectively, as *strict* and *large*, can be defined on coordinators, differing for the way labels (i.e. formulae) are compared, with $\sim_{\text{strict}} \Rightarrow \sim_{\text{large}}$, and \sim_{strict} being an equivalence relation. We show that, whenever the STS satisfies suitable properties of correspondence w.r.t. the operational semantics of components, called *correctness* and *completeness*, \sim_{large} (and thus \sim_{strict}) implies the equivalence induced by the universal closure.

For PC whose rules are in a quite general format, called *algebraic format* [16], we provide a constructive way of defining a spatio-temporal logic and we give an algorithm for building a correct and complete STS over such a logic. The algorithm, expressed as a Prolog program, builds labels by computing recursively the most general unifiers between coordinators and left-hand sides of the operational rules.

Synopsis. In § 2 we fix the notation and we recall some basic definitions. In § 3, we overview the general ideas on which our approach relies, introducing the notion of (correct and complete) STS, defining large and strict symbolic bisimilarities and showing that both relations imply bisimilarity via universal closure. In § 4, first we illustrate the algorithmic construction of correct and complete STS's for process calculi with no structural axioms and operational rules in the *algebraic format* of [16], and then we show how to deal with the common AC1 axioms for the parallel composition operator. In § 5, we test our approach against a simple case study consisting of a fragment of the ambient calculus with CCS-like communication within ambients.

Related work. The aforementioned papers by Sewell, Leifer and Milner have motivated and inspired our quest for a set of labels powerful enough to model the maximal classes of components with whom coordinators can react. The papers by Caires, Cardelli and Gordon on spatial logics have suggested us an elegant mathematical tool for expressing both structural and temporal constraints in the labels. It is worth mentioning that spatial formulae have many analogies with the topological modalities introduced separately by Fiadeiro *et al.* in [15] when proposing a verification logic for rewriting logic.

A symbolic approach to bisimulation in the case of value passing calculi, where actions are parametrised over a possibly infinite set, has been explored in [17].

Among other frameworks where the semantics of components and coordinators is defined uniformly, let us mention *tile logic* (TL) [16,6], *conditional transition systems* (CTS) [24] and *context systems* (CS) [19], which come also equipped with different formats for guaranteeing that bisimilarity is a congruence. While

$$\begin{array}{c}
\frac{P \equiv Q \in E, \sigma(P), \sigma(Q) \in \mathbb{T}_\Sigma}{\sigma(P) \equiv \sigma(Q)} \text{ (subs)} \quad \frac{p_1 \equiv q_1, \dots, p_n \equiv q_n, f \in \Sigma_n}{f(p_1, \dots, p_n) \equiv f(q_1, \dots, q_n)} \text{ (context)} \\
\frac{p \in \mathbb{T}_\Sigma}{p \equiv p} \text{ (refl)} \quad \frac{p \equiv q}{q \equiv p} \text{ (symm)} \quad \frac{p_1 \equiv p_2, p_2 \equiv p_3}{p_1 \equiv p_3} \text{ (trans)}
\end{array}$$

Fig. 1. Closure of structural axioms

models based on TL, CTS and CS can be easily translated in our framework, the use of spatial formulae makes our approach applicable to a wider class of calculi.

The idea of using unification for building formulae comes from Logic Programming and more precisely by its view as an interactive system presented in [7].

2 Notation

To ease the presentation we consider one-sorted signatures, but our results easily extend to the many-sorted case. A *signature* is a set of *operators* Σ together with an arity function $ar : \Sigma \rightarrow \mathbb{N}$. For $n \in \mathbb{N}$, we let $\Sigma_n = \{f \in \Sigma \mid ar(f) = n\}$. We denote by $\mathbb{T}_\Sigma(\mathcal{X})$ the term algebra over Σ and variables in the set \mathcal{X} (disjoint from Σ), with $\mathbb{T}_\Sigma = \mathbb{T}_\Sigma(\emptyset)$. For $P \in \mathbb{T}_\Sigma(\mathcal{X})$ we denote by $var(P)$ the set of variables $X \in \mathcal{X}$ that appear in P . If $var(P) = \emptyset$ then P is called *closed*, otherwise *open*. When signatures are used for presenting the syntax of PC, closed terms define the set \mathcal{P} of *components* p of the calculus, while the general, possibly open, terms form the set \mathcal{C} of *coordinators* C . Often we shall write $C[X_1, \dots, X_n]$ to mean that C is a coordinator such that $var(C) \subseteq \{X_1, \dots, X_n\}$.

A *structural axiom* is a sentence $P \equiv Q$ for $P, Q \in \mathbb{T}_\Sigma(\mathcal{X})$. Given a set $E = \{P_i \equiv Q_i \mid i \in I\}$ of structural axioms, we say that a Σ -algebra \mathbf{A} *satisfies* E if for any assignment $\sigma : \mathcal{X} \rightarrow \mathbf{A}$ of values to the variables in \mathcal{X} , we have that $\sigma(P_i) =_{\mathbf{A}} \sigma(Q_i)$, for all $i \in I$. The initial algebra $\mathbb{T}_{\Sigma, E}$ is the quotient of \mathbb{T}_Σ modulo the equivalence \equiv defined in Fig. 1, where axioms in E are closed w.r.t. substitution, contextualization, reflexivity, symmetry, and transitivity.

Process calculi come often equipped with LTS operational semantics, where states are components over the process signature Σ , labels range over a suitable alphabet A , and transitions model the activities of components. Commonly such LTS is specified by a collection of inductive (transition) proof rules. In the presence of structural axioms, states are equivalence classes of components (modulo \equiv), as if proof rules included:

$$\frac{p' \equiv p \quad p \xrightarrow{a} q \quad q' \equiv q}{p' \xrightarrow{a} q'} \text{ (equiv)}$$

A *bisimulation* is a symmetric, reflexive relation \approx over components such that if $p \approx q$, then for any transition $p \xrightarrow{a} p'$ there exists a component q' and

a transition $q \xrightarrow{a} q'$ with $p' \approx q'$. We denote by \sim the largest bisimulation and call it *bisimilarity*. Note that the rule (*equiv*) makes $p \sim q$ hold whenever $p \equiv q$. We call *universal bisimilarity* the usual lifting of \sim to coordinators obtained by closing for all possible substitutions:

$$C[X_1, \dots, X_n] \sim D[X_1, \dots, X_n] \stackrel{\text{def}}{\iff} \forall p_1, \dots, p_n \in \mathcal{P}, \quad C[p_1, \dots, p_n] \sim D[p_1, \dots, p_n]$$

3 Formulae as Labels

The definition of bisimilarity on coordinators based on the closure with respect to any possible substitution presents obvious drawbacks. In fact, to verify the bisimilarity of two coordinators, one is typically led to check the bisimilarity of infinitely many processes (all the possible closed instances of the coordinators). Furthermore bisimilarity of coordinators is not defined in a coinductive way and thus the coalgebraic techniques applicable to components fall short for coordinators. In trying to prove the equivalence of two coordinators it is thus convenient to perform a kind of symbolic calculation:

1. without instantiating components which do not play an active role in a step and instantiating the active components as little as possible;
2. making assumptions not only on the structure, but (as in TL, CTS, CS) also on the behaviour of the active components.

The above strategy is formalised by introducing a symbolic transition system whose states are coordinators and whose labels encode the structural and/or behavioural conditions (see points 1–2 above) that components should fulfill for enabling the move.

In the following, we assume that a process calculus PC is fixed with signature Σ and structural axioms E , whose semantics is given by the LTS \mathcal{L} over $\mathbb{T}_{\Sigma, E}$ and label alphabet Λ . We also assume that a logic \mathbb{L} over components is given, which may have modal operators and whose atomic formulae include the process variables in \mathcal{X} and the components in \mathcal{P} (with $p \models X$ for any $p \in \mathcal{P}$ and $X \in \mathcal{X}$, while $p \models q$ iff $p \equiv q$ for any $p, q \in \mathcal{P}$, where \models is satisfaction).

Definition 1 (Symbolic Transition System). A symbolic transition system (STS) \mathcal{S} over \mathbb{L} for the process calculus PC is a set of transitions

$$C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D[Y_1, \dots, Y_m]$$

where $C[X_1, \dots, X_n]$ and $D[Y_1, \dots, Y_m]$ are coordinators, $a \in \Lambda$ and φ_i are formulae in \mathbb{L} containing only variables from $\{Y_1, \dots, Y_m\}$.

The variable names in the states of \mathcal{S} are not relevant: they are just indexed placeholders, whose number can vary along the computation. The correspondence between variables in the source (e.g. X_i) and their residuals (e.g. Y_j) in the target is expressed by the formulae (e.g. φ_i), in which the residuals may

occur. For example, the modal formula $\varphi_i = \diamond a.Y_j$ is satisfied by any process performing a (and its residual replaces Y_j in D).

For \mathcal{S} to be an abstract view of PC we must of course require some additional properties enforcing the correspondence with the concrete LTS \mathcal{L} . Consider a transition system where coordinators have just one hole. Intuitively, whenever $C[X] \xrightarrow{\varphi}_a D[Y]$ the idea is that the coordinator C , when instantiated with any component satisfying φ , can perform action a becoming an instance of D . The process variable Y , which typically occur in φ , is intended to represent the residual of what substituted for X , after it has exhibited the capabilities required by φ . More precisely, for any component q such that $p \models \varphi[q/Y]$ (where $\varphi[q/Y]$ denotes the formula obtained from φ by replacing all the occurrences of Y by q) the component $C[p]$ can perform an action a becoming $D[q]$. On the other hand, any concrete transition on components should have symbolic counterparts. These two properties are formalised as *correctness* and *completeness*, respectively.

Definition 2 (Correctness). *An STS \mathcal{S} for the process calculus PC is correct if for any symbolic transition*

$$C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D[Y_1, \dots, Y_m]$$

in \mathcal{S} , for any q_1, \dots, q_m and for any $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ for $i \in [1, n]$, there exists a transition $C[p_1, \dots, p_n] \xrightarrow{a} D[q_1, \dots, q_m]$ in \mathcal{L} .

Definition 3 (Completeness). *An STS \mathcal{S} for the process calculus PC is complete if for any coordinator $C[X_1, \dots, X_n]$, for all components p_1, \dots, p_n and for any transition*

$$C[p_1, \dots, p_n] \xrightarrow{a} q$$

in \mathcal{L} there exists a transition $C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D[Y_1, \dots, Y_m]$ in \mathcal{S} and q_1, \dots, q_m such that $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ for $i \in [1, n]$, and $q \equiv D[q_1, \dots, q_m]$.

Over any STS we can straightforwardly define a bisimulation-like equivalence.

Definition 4 (Strict Symbolic Bisimulation). *A symmetric relation \approx over the set of coordinators \mathcal{C} is a strict symbolic bisimulation if for any two coordinators $C[X_1, \dots, X_n]$ and $D[X_1, \dots, X_n]$ such that $C[X_1, \dots, X_n] \approx D[X_1, \dots, X_n]$, for any transition*

$$C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a C'[Y_1, \dots, Y_m]$$

there exists a transition $D[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D'[Y_1, \dots, Y_m]$ such that $C'[Y_1, \dots, Y_m] \approx D'[Y_1, \dots, Y_m]$. The largest strict symbolic bisimulation is an equivalence relation called strict symbolic bisimilarity and denoted by \sim_{strict} .

Our first result states that the strict symbolic bisimilarity distinguishes as much as universal (closure) bisimilarity \sim , as defined by the end of Section 2.

Theorem 1 ($\sim_{\text{strict}} \Rightarrow \sim$). *If \mathcal{S} is a correct and complete STS, then*

$$C[X_1, \dots, X_n] \sim_{\text{strict}} D[X_1, \dots, X_n] \Rightarrow C[X_1, \dots, X_n] \sim D[X_1, \dots, X_n]$$

Proof. Suppose $C[X_1, \dots, X_n] \sim_{\text{strict}} D[X_1, \dots, X_n]$. We want to show that for any p_1, \dots, p_n , we have $C[p_1, \dots, p_n] \sim D[p_1, \dots, p_n]$. Let $\mathcal{R}_{\text{strict}}$ be the relation defined by

$$C[p_1, \dots, p_n] \mathcal{R}_{\text{strict}} D[p_1, \dots, p_n] \stackrel{\text{def}}{\iff} C[X_1, \dots, X_n] \sim_{\text{strict}} D[X_1, \dots, X_n].$$

We first show that $\mathcal{R}_{\text{strict}}$ is a bisimulation for \mathcal{L} .

For any transition $C[p_1, \dots, p_n] \xrightarrow{a} q$ in \mathcal{L} , by completeness of \mathcal{S} , a symbolic transition $C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)_a} C'[Y_1, \dots, Y_m]$ and m components q_1, \dots, q_m exist such that $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ and $q \equiv C'[q_1, \dots, q_m]$. Since $C[X_1, \dots, X_n] \sim_{\text{strict}} D[X_1, \dots, X_n]$ by hypothesis, we have that $D[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)_a} D'[Y_1, \dots, Y_m]$ with $C'[Y_1, \dots, Y_m] \sim_{\text{strict}} D'[Y_1, \dots, Y_m]$. By correctness of \mathcal{S} , and by $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ for all $i \in [1, n]$, it holds that $D[p_1, \dots, p_n] \xrightarrow{a} D'[q_1, \dots, q_m]$. Since $C'[Y_1, \dots, Y_m] \sim_{\text{strict}} D'[Y_1, \dots, Y_m]$, we have that $C'[q_1, \dots, q_m] \mathcal{R}_{\text{strict}} D'[q_1, \dots, q_m]$. The relation $\mathcal{R}_{\text{strict}}$ is obviously symmetric and hence it is a bisimulation. Since bisimilarity \sim is the largest bisimulation, it contains $\mathcal{R}_{\text{strict}}$ and therefore $C[p_1, \dots, p_n] \sim D[p_1, \dots, p_n]$, concluding the proof. \square

3.1 Large Symbolic Bisimulation

The requirement of exact matching between formulae in the definition of strict symbolic bisimulation can be too strong, especially in the presence of spatial formulae and structural congruences. Hence, we propose a way to relax this condition.

To this aim, we assume that the logic \mathbf{L} is a spatial logic whose operators include a subset $\Sigma_{\mathbf{L}}$ of Σ , with satisfaction defined by (for any $f \in \Sigma_{\mathbf{L}}$ with arity n):

$$p \models f(\varphi_1, \dots, \varphi_n) \quad \text{iff} \quad \exists p_1, \dots, p_n. p \equiv f(p_1, \dots, p_n) \wedge \forall i. p_i \models \varphi_i.$$

We call φ a *spatial formula* if it is built by using just variables $X \in \mathcal{X}$ and spatial operators $f \in \Sigma_{\mathbf{L}}$. Abusing the notation, a spatial formula can be either seen as a component/coordinator or as a logic formula, depending on the setting where it is used.

Definition 5 (Large Symbolic Bisimulation). *A symmetric relation \approx over the set of coordinators \mathcal{C} is a large symbolic bisimulation if for any pair of coordinators $C[X_1, \dots, X_n]$ and $D[X_1, \dots, X_n]$ such that $C[X_1, \dots, X_n] \approx D[X_1, \dots, X_n]$, for any transition*

$$C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)_a} C'[Y_1, \dots, Y_m]$$

a transition $D[X_1, \dots, X_n] \xrightarrow{(\psi_1, \dots, \psi_n)_a} D'[Z_1, \dots, Z_k]$ and k spatial formulae ψ'_1, \dots, ψ'_k exist such that $\varphi_i = \psi_i[\psi'_1/Z_1, \dots, \psi'_k/Z_k]$ and $C'[Y_1, \dots, Y_m] \approx D'[\psi'_1, \dots, \psi'_k]$. The greatest large bisimulation is called large symbolic bisimilarity and denoted \sim_{large} .

Large symbolic bisimulation allows a transition to be simulated by another transition where the spatial constraints on the Y 's are relaxed, so that “more general” components can be used for the X 's. It follows that transitions in \mathcal{S} that are dominated by transitions with a less (spatially) specified label can be abstracted away from the system.

Example 1. Let $\Sigma = \{a, f(\cdot), g(\cdot)\}$ and let the logic L include all the three corresponding spatial operators. Let \mathcal{S} be the STS with transitions $f(X) \xrightarrow{X}_\tau X$, $g(X) \xrightarrow{X}_\tau X$, and $g(X) \xrightarrow{a}_\tau a$. Then it is obvious that $f(X) \not\sim_{\text{strict}} g(X)$, because the last transition of $g(X)$ cannot be matched by $f(X)$. However, the formula X is “more general” than the formula a , and therefore $f(X) \sim_{\text{large}} g(X)$. \square

Remark 1. While \sim_{strict} is an equivalence relation, we only proved that, if necessary, \sim_{large} can be guaranteed to be an equivalence by suitably saturating the STS with redundant transitions. We also point out that the obvious way of relaxing the requirements of \sim_{strict} by allowing a step $C[X] \xrightarrow{\varphi}_a C'[Y]$ to be simulated by $D[X] \xrightarrow{\psi}_a D'[Y]$ with $\varphi \Rightarrow \psi$, would not yield a consistent formulation, as it can be seen that, contrary to spatial operators, modal operators in φ cannot be safely abstracted away in ψ .

Proposition 1 ($\sim_{\text{strict}} \Rightarrow \sim_{\text{large}}$). *For any symbolic transition system \mathcal{S}*

$$C[X_1, \dots, X_n] \sim_{\text{strict}} D[X_1, \dots, X_n] \Rightarrow C[X_1, \dots, X_n] \sim_{\text{large}} D[X_1, \dots, X_n].$$

Proof. It follows directly from the definition of the two bisimulations, since the spatial formulae ψ'_i 's used in \sim_{large} when simulating the step can of course be identities. \square

Theorem 2 ($\sim_{\text{large}} \Rightarrow \sim$). *If \mathcal{S} is correct and complete w.r.t. \mathcal{L} , then*

$$C[X_1, \dots, X_n] \sim_{\text{large}} D[X_1, \dots, X_n] \Rightarrow C[X_1, \dots, X_n] \sim D[X_1, \dots, X_n].$$

Proof. The proof is similar to, but slightly more involved than, that of Theorem 1. Suppose $C[X_1, \dots, X_n] \sim_{\text{large}} D[X_1, \dots, X_n]$. We want to show that for any p_1, \dots, p_n , we have $C[p_1, \dots, p_n] \sim D[p_1, \dots, p_n]$. Let $\mathcal{R}_{\text{large}}$ be the relation defined by

$$C[p_1, \dots, p_n] \mathcal{R}_{\text{large}} D[p_1, \dots, p_n] \stackrel{\text{def}}{\iff} C[X_1, \dots, X_n] \sim_{\text{large}} D[X_1, \dots, X_n].$$

We first show that $\mathcal{R}_{\text{large}}$ is a bisimulation for \mathcal{L} . For any transition $C[p_1, \dots, p_n] \xrightarrow{a} q$ in \mathcal{L} , by completeness of \mathcal{S} , a symbolic transition $C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)_a}$

$C'[Y_1, \dots, Y_m]$ and m components q_1, \dots, q_m exist with $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ and $q \equiv C'[q_1, \dots, q_m]$. Since $C[X_1, \dots, X_n] \sim_{\text{large}} D[X_1, \dots, X_n]$ by hypothesis, we have

$$D[X_1, \dots, X_n] \xrightarrow{(\psi_1, \dots, \psi_n)}_a D'[Z_1, \dots, Z_k]$$

and k spatial formulae ψ'_1, \dots, ψ'_k exist such that $C'[Y_1, \dots, Y_m] \sim_{\text{large}} D'[\psi'_1, \dots, \psi'_k]$ and $\varphi_i = \psi_i[\psi'_1/Z_1, \dots, \psi'_k/Z_k]$ for all $i \in [1, n]$. Since $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ (for all $i \in [1, n]$), letting $q'_i \equiv \psi'_i[q_1/Y_1, \dots, q_m/Y_m]$, it follows that $p_i \models \psi_i[q'_1/Z_1, \dots, q'_k/Z_k]$. Therefore, by correctness of \mathcal{S} , it follows that $D[p_1, \dots, p_n] \xrightarrow{a} D'[q'_1, \dots, q'_k]$. Moreover, since $C'[Y_1, \dots, Y_m] \sim_{\text{large}} D'[\psi'_1, \dots, \psi'_k]$, we have that $C'[q_1, \dots, q_m] \mathcal{R}_{\text{large}} D'[q'_1, \dots, q'_k]$. The relation $\mathcal{R}_{\text{large}}$ is clearly symmetric and hence it is a bisimulation for \mathcal{L} . Since bisimilarity \sim is the largest bisimulation, it contains $\mathcal{R}_{\text{large}}$ and thus $C[p_1, \dots, p_m] \sim D[q_1, \dots, q_m]$. \square

Note that Theorem 1 now follows as a corollary of Proposition 1 and Theorem 2.

4 Bisimulation by Unification

In this section we outline a methodology for deriving a correct and complete STS for algebraic PC, i.e. PC whose operational proof rules are in a quite general format, called *algebraic format* [16], recalled below. More specifically, given a PC, a logic L_{PC} with spatial and modal operators in the style of [8,11] can be systematically derived. Then the proof rules of the calculus are used to construct a Prolog program (finite if the set of proof rules of the PC is finite) which represents an STS over L_{PC} for the PC, in the sense that given any coordinator, the program allows to compute the set of its symbolic transitions. Such STS can be proved to be correct and complete for the given PC.

Definition 6 (Algebraic Format). *A proof rule is in algebraic format if it has the form*

$$\frac{\{X_i \xrightarrow{a_i} Y_i\}_{i \in I}}{C[X_1, \dots, X_n] \xrightarrow{a} D[Z_1, \dots, Z_n]}$$

with $I \subseteq [1, n]$, and where $Z_i = Y_i$ if $i \in I$ and $Z_i = X_i$ otherwise. An algebraic process calculus is a PC whose proof rules are in algebraic format.

The algebraic format generalises De Simone format [14] by allowing a generic context C , possibly involving more than one operator, (to appear) as left-hand side of the conclusion of the rule. However, it is worth recalling that while De Simone format guarantees that bisimilarity is a congruence, for algebraic PC's this is not necessarily the case.

4.1 A Spatio-temporal Logic for Symbolic Transition Systems

Given a process calculus PC over a signature Σ we define the logic whose formulae will be used as labels in the STS. The logic must be powerful enough

$$\begin{array}{ll}
p \models X & \\
p \models q & \text{iff } p \equiv q \\
p \models \diamond a. \varphi & \text{iff } \exists p'. p \xrightarrow{a} p' \wedge p' \models \varphi \\
p \models f(\varphi_1, \dots, \varphi_n) & \text{iff } \exists p_1, \dots, p_n. p \equiv f(p_1, \dots, p_n) \wedge p_i \models \varphi_i
\end{array}$$

Fig. 2. Satisfaction of formulae in the STS logic L_{PC}

to be able to express, for any coordinator, the (more general) structural and behavioural properties which should be fulfilled by unspecified components to allow transitions to happen.

Definition 7 (sts Logic). Let Σ_s be the set of operators in Σ which appear in the left-hand side of the conclusion of a proof rule of PC (e.g. the operators in $C[X_1, \dots, X_n]$ for the rule of Definition 6). The STS logic L_{PC} associated to PC has as formulae

$$\varphi ::= X \mid p \mid \diamond a. \varphi \mid f(\varphi_1, \dots, \varphi_n)$$

where $X \in \mathcal{X}$, $p \in \mathcal{P}$, $a \in \Lambda$, $f \in \Sigma_s$.

A formula $f(\varphi_1, \dots, \varphi_n)$ is satisfied by any component of the shape $f(p_1, \dots, p_n)$ where each p_i satisfies φ_i . A formula $\diamond a. \varphi$ is satisfied by any component which is able to perform an a -labelled transition, evolving in a component satisfying φ . Since the logic will be used to label the transitions of an STS, according to the general assumptions in Section 3, process variables and (closed) components are included as atomic formulae. Observe that, if $\Sigma_s = \Sigma$ then all components can be inductively constructed as formulae of the kind $f(\varphi_1, \dots, \varphi_n)$ with $f \in \Sigma$ and thus there is no need to add them explicitly (but in most PC no rule is given for the nil component 0, which is thus not in Σ_s). Satisfaction is formally defined in Fig. 2 (for any $p \in \mathcal{P}$ and for any formula φ in L_{PC}).

To understand the definition of the STS logic L_{PC} note that an instance $C[p_1, \dots, p_n]$ of a given coordinator $C[X_1, \dots, X_n]$, in order to perform a transition, must match the left-hand side of the conclusion of a rule. This might impose the components p_i 's to have a certain structure, hence the need of inserting the spatial operators $f \in \Sigma_s$ in the logic. Furthermore, the premises of the matched rule must be satisfiable. Such premises usually require the components p_i 's to be able to exhibit some behaviour, i.e. to perform a certain transition. Hence the logic includes also modal operators $\diamond a.(_)$.

4.2 Algebraic PC without Structural Axioms

We next illustrate a constructive procedure for defining a correct and complete STS over the logic L_{PC} for a given process calculus PC whose proof rules are in algebraic format. Here we concentrate on process calculi *without* structural axioms. In Section 4.3 will discuss the refinements needed in the presence of structural axioms.

The STS over L_{PC} is specified by means of a Prolog program which can be used to compute the possible symbolic transitions of every coordinator.

Definition 8 (Prolog Program). *The Prolog program $Prog(PC)$ associated to the process calculus PC contains as the first clause*

$$\text{trs}(\text{box}(A, X), A, X) \text{ :- } !.$$

where box is a new operator, not in Σ , and A is a variable that stands for any action. For any proof rule in PC of the shape outlined in Definition 6 also a clause

$$\text{trs}(C[X_1, \dots, X_n], a, D[Z_1, \dots, Z_n]) \text{ :- } \text{trs}(X_{i_1}, a_{i_1}, Y_{i_1}), \dots, \text{trs}(X_{i_k}, a_{i_k}, Y_{i_k}).$$

is included, where $\{i_1, \dots, i_k\}$ is the set of indexes I of the corresponding rule and Z_i can be either Y_i (when $i \in I$) or X_i (otherwise).

The program $Prog(PC)$ defines the predicate $\text{trs}(X, A, Y)$ whose intended meaning is “any component satisfying X can perform a transition labelled by A and become a component satisfying Y ”. Given a coordinator $C[X_1, \dots, X_n]$, if the query

$$?- \text{trs}(C[X_1, \dots, X_n], A, Z)$$

is successful, then the corresponding computed answer substitution can be seen as a symbolic step for the coordinator $C[X_1, \dots, X_n]$: the computed answer substitutions for the variables X_1, \dots, X_n will represent the formulae in L_{PC} labelling the transition, A the action label and Z the target coordinator.

The first clause in $Prog(PC)$ can be unified only with a goal $\text{trs}(X, A, _)$ whose first argument is a variable (since box is not an operator in PC). In this case there is no need of imposing structural requirements on X , since the only requirement for any component X for doing a and becoming Y is exactly $\text{box}(a, Y)$. Thus the goal is refuted just imposing a behavioural constraint on the component corresponding to X , i.e. by asking that X can perform an A action. The cut operator in the body of the clause avoids that subsequent refutations are tried, using different rules that could be otherwise matched by the goal $\text{trs}(X, A, _)$. To this aim, it is important that modal rules be listed first than all the other rules.

The second class of clauses in $Prog(PC)$ just represents a Prolog translation of the operational proof rules of the calculus. Each such clause imposes (by unification) the more general structural (spatial) constraints that the unspecified components of a coordinator should satisfy to allow the corresponding step. The requirements on the behaviour of the subcomponents, as expressed by the premises of the corresponding proof rule, are represented by the subgoals in the body of the clause.

The backtracking mechanism of Prolog and the use of meta-logic operators (like `bagof`) allow one to determine all the symbolic transitions for each coordinator C (finitely many under the assumption that the rules of the calculus and thus the program are finite). Hence the Prolog program $Prog(PC)$ can be seen as the specification of an STS for the process calculus PC over logic L_{PC} . The main result of this section states that such STS is correct and complete for the considered process calculus.

Theorem 3. *The STS specified by $Prog(PC)$ is correct and complete.*

Proof (Sketch). To prove correctness observe that if $C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D[Y_1, \dots, Y_m]$ then there exists a refutation of the query

$$?- \text{trs}(C[X_1, \dots, X_n], a, Z)$$

with computed answer substitution $X_i = \varphi_i$ and $Z = D[Y_1, \dots, Y_m]$. An inductive reasoning on the height of the refutation allows us to prove that for any q_1, \dots, q_m and p_1, \dots, p_n such that each $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$ there exists a derivation of $C[p_1, \dots, p_n] \xrightarrow{a} D[q_1, \dots, q_m]$.

As for completeness, if $C[p_1, \dots, p_n] \xrightarrow{a} q$ then the corresponding derivation in the proof system of PC can be turned into a refutation witnessing that $C[X_1, \dots, X_n] \xrightarrow{(\varphi_1, \dots, \varphi_n)}_a D[Y_1, \dots, Y_m]$. Furthermore $q \equiv D[q_1, \dots, q_m]$ and each $p_i \models \varphi_i[q_1/Y_1, \dots, q_m/Y_m]$. \square

4.3 Algebraic PC with AC1 Parallel Composition Operator

To understand why the proposed approach must be extended to deal with structural axioms, we focus on a very common case, i.e., an algebraic PC with a *parallel composition operator* “|”, subject to AC1 axioms (associativity, commutativity and identity)

$$(X | Y) | Z \equiv X | (Y | Z) \quad X | Y \equiv Y | X \quad X | \mathbf{0} \equiv X$$

where $\mathbf{0}$ is the inactive component. Furthermore we suppose that parallel composition allows a single component to move autonomously, performing an action that is reflected at topmost level, i.e., we assume that the proof rules for parallel composition include

$$\frac{X \xrightarrow{a} X'}{X | Y \xrightarrow{a} X' | Y} \text{ (par)}$$

For the construction of the Prolog program $Prog(PC)$ we first need to extend the set of proof rules of the calculus. Due to the presence of the associativity axiom, for any proof rule r of the calculus where “|” occurs in the left-hand side of the conclusion as topmost operator, we have to insert a new rule r' . The new rule is obtained from r by adding in parallel a generic idle component, i.e. for any rule of the kind

$$\frac{\{X_i \xrightarrow{a_i} Y_i\}_{i \in I}}{C_1[X_1, \dots, X_n] | C_2[X_{n+1}, \dots, X_{n+m}] \xrightarrow{a} D[Z_1, \dots, Z_{n+m}]}$$

we add a new rule (analogous to the completion in rewriting systems modulo AC1)

$$\frac{\{X_i \xrightarrow{a_i} Y_i\}_{i \in I}}{C_1[X_1, \dots, X_n] | C_2[X_{n+1}, \dots, X_{n+m}] | X_{n+m+1} \xrightarrow{a} D[Z_1, \dots, Z_{n+m}] | X_{n+m+1}}$$


```

trs( box(tau,X) , tau , X ) :- !.
trs( a.X|'a      , tau , X ).
trs( X|Y         , tau , X|Z ) :- trs(Y, tau, Z).

```

Fig. 3. The Prolog program relative to the simple CCS-like calculus

Then $Prog(PC)$ is defined exactly as before. Of course unification must be considered up to AC1 structural axioms (see algorithms and further references in [18,4]).

Example 2. Consider a simple CCS-like calculus, with AC1 parallel composition and only one rule for asynchronous communication

$$\frac{}{a.X \mid \bar{a} \xrightarrow{\tau} X}$$

The Prolog program induced by the original proof rule is shown in Fig. 3, where 'a is the program representation for action \bar{a} . The following query

```
?- trs(a.0|a.0|X, A, Z)
```

would return the substitutions $X = 'a$ for X and $Z = a.0$ for Z ; but $X = 'a$ is not the more general substitution for X that allows the context to perform the step. In fact, the coordinator $a.0|a.0|X$, instantiated with a component p satisfying the formula $\varphi = \bar{a}$ returned by the Prolog program (namely with $p = \bar{a}$), could perform only one step, but, obviously, X could also be instantiated with the component $\bar{a} \mid \bar{a}$, allowing the coordinator to perform two steps. Actually, $X = 'a \mid Y$ results to be the more general substitution which, thanks to the identity axiom, “comprises” the previous one. In order to obtain such a computed answer from the program, it is enough to extend the proof system with the rule r'

$$\frac{}{a.X \mid \bar{a} \mid Y \xrightarrow{\tau} X \mid Y}$$

□

It is easy to show that the new proof rules r' are valid in the original proof system, hence the extension of the proof system does not change the semantics of the PC. Due to the presence of the identity axiom, for any r' we can also remove the original rule r without affecting the semantics of the calculus. The result expressed by Theorem 3 extends also to this case, i.e., the STS specified by $Prog(PC)$ is correct and complete.

An analogous approach can be followed to deal with a *replication* operator “!”, subject to the structural axiom $!X \equiv !X \mid X$.

5 Case Study: A Basic Calculus for Mobility

We consider a basic calculus for mobility (BCM) which can be seen as an asynchronous version of CCS [21], enriched with ambients, or, alternatively, as (a restriction-free version of) the ambient calculus [12] with asynchronous CCS-like communication.

$$\begin{array}{c}
\frac{}{n[P] \mid \text{open } n.Q \rightarrow P|Q} \text{ (open)} \qquad \frac{}{n[P] \mid m[\text{in } n.Q|R] \rightarrow n[P|m[Q|R]]} \text{ (in)} \\
\frac{}{n[P|m[\text{out } n.Q|R]] \rightarrow n[P] \mid m[Q|R]} \text{ (out)} \qquad \frac{}{n[a.P|\bar{a}|Q] \rightarrow n[P|Q]} \text{ (comm)} \\
\frac{P \rightarrow Q}{n[P] \rightarrow n[Q]} \text{ (amb)} \qquad \frac{P \rightarrow Q}{P|R \rightarrow Q|R} \text{ (par)}
\end{array}$$

Fig. 4. Operational semantics of BCM

Definition 9 (bcm). Let \mathbf{A} be a set of channels and let \mathcal{N} be a set of ambient names. The set of BCM processes \mathcal{P} is defined by the grammar:

$$P ::= 0 \mid \bar{a} \mid a.P \mid \text{open } n.P \mid \text{in } n.P \mid \text{out } n.P \mid n[P] \mid P|P$$

with $a \in \mathbf{A}$, $n \in \mathcal{N}$, and where the parallel operator is AC1:

$$P|(Q|R) \equiv (P|Q)|R \quad P|Q \equiv Q|P \quad P|0 \equiv P$$

The operational semantics of BCM is defined by the SOS operational rules in Fig. 4. The rules *open*, *in*, and *out* are the classical rules of ambient calculus; communication (rule *com*) is allowed only inside the same ambient; reductions can happen under any ambient and in any parallel process (but not under prefixes), as stated by rules *amb* and *par*, respectively. Since the semantics is presented as a reduction system, transitions have no label (or equivalently they can be thought of as having all the same label τ).

The logic \mathbf{L}_{BCM} over the set of components is defined as explained in Section 4.1. The set of spatial operators of the logic includes all the operators of the signature, i.e., $\Sigma_s = \Sigma$, so as to characterise all the possible transitions of the semantics of BCM (strictly speaking, $0 \notin \Sigma_s$, but its presence is harmless and makes the notation simpler). All axioms in Fig. 4 introduce the need of spatial formulae (the lefthand side of the reduction requires a specific structure of the component). The rules *amb* and *par*, instead, calls for modal formulae, since their premises refer to observable behaviours and not to the structure of the components. The formulae φ of the logic \mathbf{L}_{BCM} are:

$$\varphi ::= X \mid \diamond . \varphi \mid 0 \mid \alpha.\varphi \mid n[\varphi] \mid \varphi_1|\varphi_2,$$

where $X \in \mathcal{X}$, $n \in \mathcal{N}$ and $\alpha \in \{a, \bar{a}, \text{open } n, \text{in } n, \text{out } n\}$. Since transitions are not labelled, the modal operator does not refer to any action. The notion of satisfaction for \mathbf{L}_{BCM} ($P \models \varphi$) is defined like in the general case (see Fig. 2). Then we can consider the correct and complete STS for BCM specified by the Prolog program *Prog*(BCM).

To have a grasp of the properties of the calculus, let us consider two ambients with different names $n[a.0 \mid \bar{a}.0]$ and $m[b.0 \mid \bar{b}.0]$. Both processes are able to perform an internal communication according to rule *comm*, evolving to a

(deadlocked) ambient containing the nil component 0. Straightforwardly,

$$n[a.0 \mid \bar{a}.0] \sim m[b.0 \mid \bar{b}.0],$$

i.e. internal actions do not distinguish ambients. It is easy to show that bisimilarity is not a congruence for this calculus, since the above bisimilar processes are distinguished when put in parallel with *open* $n.0$ (it interacts with $n[a.0 \mid \bar{a}.0]$ but not with $m[b.0 \mid \bar{b}.0]$).

Processes $n[a.0 \mid \bar{a}.0]$ and $m[b.0 \mid \bar{b}.0]$ are (bisimilar) instances of the coordinators $n[X]$ and $m[X]$. It is easy to verify $n[X] \not\sim_{\text{strict}} m[X]$, in fact, due to rule *out*:

$$n[X] \xrightarrow{Y \mid m[\text{out } n. Z \mid W]} n[Y] \mid m[Z \mid W],$$

while $m[X]$ has an analogous transition but with a different label and conclusion:

$$m[X] \xrightarrow{Y \mid n[\text{out } m. Z \mid W]} m[Y] \mid n[Z \mid W].$$

Actually, $n[X] \not\sim m[X]$, since they are distinguished by $X = k[\text{out } n.0]$, and hence, by Theorem 2, $n[X] \not\sim_{\text{large}} m[X]$. An example of coordinators related by \sim_{strict} , and hence, using Theorems 1 and 2, also by \sim_{large} and \sim , is: $n[m[\text{out } n.X]] \sim_{\text{strict}} n[0 \mid m[a \mid \bar{a}.X]]$. In fact, the two coordinators have the only symbolic transitions below, which lead to obviously bisimilar coordinators:

$$n[m[\text{out } n.X]] \xrightarrow{Y} n[0 \mid m[Y]] \quad \text{and} \quad n[0 \mid m[a \mid \bar{a}.X]] \xrightarrow{Y} n[0 \mid m[\bar{Y}]].$$

6 Conclusions

We have illustrated a general methodology for reasoning about open systems, viewed as coordinators in suitable process calculi, with special interest in bisimilarity. For a PC and a process logic which characterises the structural and behavioural properties of interest, we have introduced a notion of (correct and complete) symbolic transition system, where states are coordinators and transitions are labelled by logic formulae expressing the requirements which uninstantiated components should satisfy for the transition to happen. Over an STS two symbolic bisimilarities can be defined, the *strict bisimilarity* and the “coarser” *large bisimilarity*, both refining the universal bisimilarity on coordinators which takes all possible closed instantiations. For algebraic PC, whose rules are in a quite general format, we have also provided a constructive way of deriving a spatio-temporal logic and a (correct and complete) symbolic transition system over such logic. The applicability of the proposed methodology has been finally illustrated by means of a toy process calculus with CCS-like communication within ambients.

An interesting issue which has not been faced here is the treatment of names and name restriction, which plays a basic role in the specification of systems with fresh or secret resources. While the notions of (correct and complete) STS and the results about symbolic bisimulation are parametric w.r.t. the chosen

process logic, the constructive definition of the correct and complete STS for a given process calculus, presented in Section 4, and especially the definition of the underlying process logic, should be extended to deal with a logical notion of freshness. A source of inspiration could be the work of Cardelli and Caires [9,10].

We already mentioned that symbolic transitions have been studied by several authors, e.g. Sewell [25], and Leifer and Milner [20] in order to avoid universal quantification over contexts. These approaches, where steps can perform contextual closures, can be seen as the dual of our approach, where steps can instantiate contexts. It would be interesting to give a formal account of this duality, and, in particular, to see if the categorical approach of [20], based on relative pushouts, can be dualised in our case resorting to a notion of relative pullback.

Recently, the symbolic approach to the verification of infinite state cryptographic protocols has attracted a lot of interest. Some authors use logic abstractions to characterise symbolic states [1,13], others exploit, in particular, the generality of unification to devise minimal assumptions over symbolic states [5]. Pursuing further the similarities of our symbolic approach to bisimulation with these approaches, so as to apply our methodology to the field, appears to be a stimulating line of future research.

Regarding the automatic construction of STS, we plan to generalise it to *meta* and *abductive* Logic Programming. The first one should allow for the programmable definition of proofs, and hence for more specific reasoning over the structure of a PC. The second one should provide the means for hypothetical (assumption-based) reasoning about the properties labelling STS, allowing to answer questions like “under which circumstances (assumptions) the process $P \mid X$ can evolve so as to satisfy a given property?”, typically relevant in open and dynamic system engineering [3,2].

Acknowledgements

We would like to thank Narciso Martí-Oliet, Sabina Rossi and the anonymous referees for their helpful comments and suggestions.

References

1. M. Abadi and M. P. Fiore. Computing symbolic models for verifying cryptographic protocols. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pp. 160–173. IEEE, 2001. 269
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connectors. *ACM TOSEM*, 6(3), pp. 213–249, 1997. 269
3. L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos and M. Wermelinger. Coordination for Orchestration. In *Coordination Models and Languages*, 5th Int. Conference COORDINATION. *Lect. Notes in Comput. Sci.* 2315 pp. 5–13. Springer 2002. 269
4. F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*. Elsevier Science, 2000. 266

5. M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proc. ICALP'01, Lect. Notes in Comput. Sci.* 2076, pp. 667–681. Springer, 2001. [269](#)
6. R. Bruni, D. de Frutos-Escrig, N. Martí-Oliet, and U. Montanari. Bisimilarity congruences for open terms and term graphs via tile logic. In *Proc. CONCUR 2000, Lect. Notes in Comput. Sci.* 1877, pp. 259–274. Springer, 2000. [256](#)
7. R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(6):647–690, 2001. [257](#)
8. L. Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, Departamento de Informática, Universidade Nova de Lisboa, 1999. [256](#), [262](#)
9. L. Caires and L. Cardelli. A spatial logic for concurrency (part I). In *Proc. TACS 2001, Lect. Notes in Comput. Sci.* 2215, pp. 1–37. Springer, 2001. [269](#)
10. L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *Proc. CONCUR 2002, Lect. Notes in Comput. Sci.*, Springer, 2002. To appear. [269](#)
11. L. Cardelli and A. D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proc. POPL 2000*, pp. 365–377. ACM, 2000. [256](#), [262](#)
12. L. Cardelli and A. D. Gordon. Mobile ambients. In *Proc. FoSSaCS'98, Lect. Notes in Comput. Sci.* 1378, pp. 140–155. Springer, 1998. [266](#)
13. E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. PROCOMET'98*. Chapman & Hall, 1998. [269](#)
14. R. De Simone. Higher level synchronizing devices in MEIJE-SCCS. *TCS*, 37:245–267, 1985. [262](#)
15. J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In *Proc. WADT'99, LNCS 1827*, pp. 438–458. Springer, 2000. [256](#)
16. F. Gadducci and U. Montanari. The tile model. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. [256](#), [262](#)
17. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoret. Comp. Sci.*, 138:353–389, 1995. [256](#)
18. A. Herold and J. Siekmann. Unification in abelian semi-groups. *Journal of Automated Reasoning*, 3(3):247–283, 1987. [266](#)
19. K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. In *Proc. ICALP'90, Lect. Notes in Comput. Sci.* 443, pp. 526–539. Springer, 1990. [256](#)
20. J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *Proc. CONCUR 2000, Lect. Notes in Comput. Sci.* 1877, pp. 243–258. Springer, 2000. [255](#), [269](#)
21. R. Milner. *A Calculus of Communicating Systems, LNCS 92*. Springer, 1980. [266](#)
22. U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16:171–196, 1992. [255](#)
23. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981. [255](#)
24. A. Rensink. Bisimilarity of open terms. *Inform. and Comput.*, 156(1-2):345–385, 2000. [256](#)
25. P. Sewell. From rewrite rules to bisimulation congruences. In *Proc. CONCUR'98, Lect. Notes in Comput. Sci.* 1466, pp. 269–284. Springer, 1998. [255](#), [269](#)

Transforming Processes to Check and Ensure Information Flow Security^{*}

Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari di Venezia
{bossi,focardi,piazza,srossi}@dsi.unive.it

Abstract. *Persistent_BNDC* (P_BNDC for short) is an information-flow security property for processes in dynamic contexts, i.e., contexts that can be reconfigured at runtime. We propose a method for transforming an arbitrary process into a process satisfying P_BNDC and show that the transformation preserves the “low level” observational semantics for a large class of processes. We also study how to efficiently verify P_BNDC by exploiting a characterization of it through a suitable notion of weak bisimulation *up to high level actions*. We define a second transformation over processes which allows us to reduce the problem of checking P_BNDC to the problem of testing a weak bisimulation between two processes. This approach is particularly appealing as it allows us to perform the P_BNDC check using already existing tools at a low time complexity.

1 Introduction

Systems are becoming more and more complex, and the security community has to face this by taking into account new threats and potentially dangerous situations. A significant example is the introduction of *process mobility* among different architectures and systems. A mobile process moving on the network collects information about the environments it crosses, and such information can influence it. A system or an application executing in a “secure way” inside one environment could find itself in a “insecure state” when moving to a different environment. In this setting, one can abstractly think that the environment is dynamically reconfigured at run-time, changing in unpredictable ways.

A number of formal definitions of security properties (see, for instance, [1, 3, 5, 7, 13, 14, 15, 17, 20, 21, 22]) has been proposed in the literature. *Persistent_BNDC* (P_BNDC , for short), proposed in [10], is a security property which is suitable to analyze processes in completely dynamic hostile environments, i.e., environments which can be dynamically reconfigured at run-time, changing in unpredictable ways. The notion of P_BNDC is based on the idea of Non-Interference [11, 19, 22] (formalized as $BNDC$ [7]) and requires that every state which is reachable by the system still satisfies a basic Non-Interference property.

^{*} This work has been partially supported by the MURST project “Modelli formali per la sicurezza” and the EU Contract IST-2001-32617 “Models and Types for Security in Mobile Distributed Systems” (MyThS).

If this holds, one is assured that even if the environment changes during the execution no malicious attacker will be able to compromise the system, as every possible reachable state is guaranteed to be secure. In [10] it is proved that the P_BNDC property is equivalent to an already proposed security property called $SBSNNI$ and studied in [7]. In particular, the property $SBSNNI$ is compared with different properties in the taxonomy of Non-Interference properties [11]. From the analysis presented in [7] two important problems emerge: how to verify the P_BNDC property and how to construct P_BNDC processes. The first problem has been considered in [10] where it has been shown to be decidable, and in [9] where efficiency issues have also been tackled. To the best of our knowledge the second problem has not been analyzed yet. In [7] there are many examples of processes which are not P_BNDC but can be modified in order to obtain a P_BNDC process. However each single example is treated in a different way by applying each time an “ad hoc” re-definition.

Our purpose here is to find a general method for rectifying non P_BNDC processes. It turns out that the method we suggest can be used both to rectify and to efficiently verify the P_BNDC property. We automatically transform a process E into a P_BNDC process E^τ and identify a large class of processes for which the transformation preserves the low level observational semantics, i.e., for the low level user E and E^τ are not distinguishable. This transformation can be used to construct “secure” processes from a first possibly “insecure” definition. Moreover, this also allows us to give an alternative characterization of P_BNDC through a suitable notion of weak bisimulation *up to high level actions* [10]. More precisely, we obtain that a process E is P_BNDC if and only if E and E^τ are weak bisimilar up to high level actions. The problem of verifying whether a process is P_BNDC is then reduced to the problem of checking whether E and E^τ are weak bisimilar up to high level actions. We show that this problem can be further simplified reducing it to the problem of checking the more usual notion of weak bisimulation between two processes. In particular we define a second transformation over processes such that the problem of checking whether a process is P_BNDC is reduced to the problem of testing a weak bisimulation relation. This approach seems to be particularly appealing as it allows us to perform the P_BNDC check using already existing tools at a low time complexity.

The paper is organized as follows. In Section 2 we present some basic notions on the SPA language, we introduce the P_BNDC property and we recall its characterization in terms of weak bisimulation up to high level actions. In Section 3 we define our first transformation and prove its main properties. In Section 4 we introduce a second transformation and show how to use both our transformations to check P_BNDC . In Section 5 we illustrate the usefulness of our transformations on a simple example. Finally, in Section 6 we draw some conclusions.

2 Basic Notions

In this section we report from [7] the syntax and semantics of the *Security Process Algebra* together with the definition of the Non-Interference property called *BNDC*. We then report from [10] the definition of *Persistent-BNDC* together with the main result that we will exploit for the verification of such a property.

The SPA Language. The *Security Process Algebra* (SPA, for short) [7] is a variation of Milner's CCS [16], where the set of visible actions is partitioned into high level actions and low level ones in order to specify multilevel systems. SPA syntax is based on the same elements as CCS that is: a set \mathcal{L} of *visible* actions such that $\mathcal{L} = I \cup O$ where $I = \{a, b, \dots\}$ is a set of *input* actions and $O = \{\bar{a}, \bar{b}, \dots\}$ is a set of *output* actions; a special action τ which models internal computations, i.e., not visible outside the system; a complementation function $\bar{\cdot} : \mathcal{L} \rightarrow \mathcal{L}$, such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$, and $\bar{\tau} = \tau$; $Act = \mathcal{L} \cup \{\tau\}$ is the set of all *actions*. The set of visible actions is partitioned into two sets, H and L , of high and low actions such that $\bar{H} = H$ and $\bar{L} = L$. The syntax of SPA *terms* (or *processes*) is defined as follows:

$$E ::= \mathbf{0} \mid a.E \mid E + E \mid E|E \mid E \setminus v \mid E[f] \mid Z$$

where $a \in Act$, $v \subseteq \mathcal{L}$, $f : Act \rightarrow Act$ is such that $f(\bar{a}) = \overline{f(a)}$ and $f(\tau) = \tau$, and Z is a constant that must be associated with a definition $Z \stackrel{\text{def}}{=} E$.

Intuitively, $\mathbf{0}$ is the empty process that does nothing; $a.E$ is a process that can perform an action a and then behaves as E ; $E_1 + E_2$ represents the non-deterministic choice between the two processes E_1 and E_2 ; $E_1|E_2$ is the parallel composition of E_1 and E_2 , where executions are interleaved, possibly synchronized on complementary input/output actions, producing an internal action τ ; $E \setminus v$ is a process E prevented from performing actions in v ; $E[f]$ is the process E whose actions are renamed *via* the relabelling function f . For the definition of security properties it is also useful the *hiding* operator, $/$, of CSP which can be defined as a relabelling as follows: for a given set $v \subseteq \mathcal{L}$, $E/v \stackrel{\text{def}}{=} E[f_v]$ where $f_v(x) = x$ if $x \notin v$ and $f_v(x) = \tau$ if $x \in v$. In practice, E/v turns all actions in v into internal τ 's.

Given a fixed language \mathcal{L} we denote by \mathcal{E} the set of all SPA processes and by \mathcal{E}_H the set of all high level processes, i.e., those constructed on $H \cup \{\tau\}$.

The operational semantics of SPA agents is given in terms of *Labelled Transition Systems*. A *Labelled Transition System* (LTS) is a triple (S, A, \rightarrow) where S is a set of states, A is a set of labels (actions), $\rightarrow \subseteq S \times A \times S$ is a set of labelled transitions. The notation $(S_1, a, S_2) \in \rightarrow$ (or equivalently $S_1 \xrightarrow{a} S_2$) means that the system can move from the state S_1 to the state S_2 through the action a . The operational semantics of SPA is the LTS $(\mathcal{E}, Act, \rightarrow)$, where the states are the terms of the algebra and the transition relation $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is defined by structural induction as the least relation generated by the inference rules reported in Fig. 1. The operational semantics for an agent E is the

$$\begin{array}{l}
\text{Prefix} \frac{}{a.E \xrightarrow{\alpha} E} \\
\text{Sum} \frac{E_1 \xrightarrow{\alpha} E'_1 \quad E_2 \xrightarrow{\alpha} E'_2}{E_1 + E_2 \xrightarrow{\alpha} E'_1 + E'_2} \\
\text{Parallel} \frac{E_1 \xrightarrow{\alpha} E'_1 \quad E_2 \xrightarrow{\alpha} E'_2 \quad E_1 \xrightarrow{\alpha} E'_1 \quad E_2 \xrightarrow{\bar{\alpha}} E'_2}{E_1|E_2 \xrightarrow{\alpha} E'_1|E_2 \quad E_1|E_2 \xrightarrow{\alpha} E_1|E'_2 \quad E_1|E_2 \xrightarrow{\tau} E'_1|E'_2} \quad a \in \mathcal{L} \\
\text{Restriction} \frac{E \xrightarrow{\alpha} E'}{E \setminus v \xrightarrow{\alpha} E' \setminus v} \quad \text{if } a \notin v \\
\text{Relabelling} \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]} \\
\text{Constant} \frac{E \xrightarrow{\alpha} E'}{A \xrightarrow{\alpha} E'} \quad \text{if } A \stackrel{\text{def}}{=} E
\end{array}$$

Fig. 1. The operational rules for SPA

subpart of the SPA LTS reachable from the initial state and we refer to it as $LTS(E) = (S_E, Act, \rightarrow)$. A process E is said to be *finite-state* if S_E is finite.

In [16] it is shown that a finite-state process E can always be defined through a system S of equations of the form

$$E_j = a_1.E_1 + \dots + a_n.E_n,$$

such that $E_1, \dots, E_n \in S_E$ and there is one equation in S for each $E_j \in S_E$.

The concept of *observation equivalence* between two processes is based on the idea that two systems have the same semantics if and only if they cannot be distinguished by an external observer. This is obtained by defining an equivalence relation over \mathcal{E} . In the following, we report the definition of an observation equivalence called *weak bisimulation* [16]. Intuitively, weak bisimulation equates two processes if they are able to mutually simulate their behavior step by step. Weak bisimulation does not care about internal τ actions. So, when F simulates an action of E , it can also execute some τ actions before or after that action.

We will use the following auxiliary notations. If $t = a_1 \dots a_n \in Act^*$ and $E \xrightarrow{a_1} \dots \xrightarrow{a_n} E'$, then we write $E \xrightarrow{t} E'$. We also write $E \xRightarrow{t} E'$ if $E \xrightarrow{(\tau)^*} \xrightarrow{a_1} \xrightarrow{(\tau)^*} \dots \xrightarrow{(\tau)^*} \xrightarrow{a_n} \xrightarrow{(\tau)^*} E'$ where $(\tau)^*$ denotes a (possibly empty) sequence of τ labelled transitions. If $t \in Act^*$, then $\hat{t} \in \mathcal{L}^*$ is the sequence gained by deleting all occurrences of τ from t . As a consequence, $E \xRightarrow{\hat{t}} E'$ stands for $E \xrightarrow{t} E'$ if

$a \in \mathcal{L}$, and for $E(\xrightarrow{\tau})^* E'$ if $a = \tau$ (note that $\xrightarrow{\tau}$ requires at least one τ labelled transition while $\xrightarrow{\hat{\tau}}$ means zero or more τ labelled transitions).

Definition 1 (Weak Bisimulation). A binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over agents is a weak bisimulation if $(E, F) \in \mathcal{R}$ implies, for all $a \in \text{Act}$,

- if $E \xrightarrow{a} E'$, then there exists F' such that $F \xrightarrow{\hat{a}} F'$ and $(E', F') \in \mathcal{R}$;
- if $F \xrightarrow{a} F'$, then there exists E' such that $E \xrightarrow{\hat{a}} E'$ and $(E', F') \in \mathcal{R}$.

Two agents $E, F \in \mathcal{E}$ are weakly bisimilar, denoted by $E \approx F$, if there exists a weak bisimulation \mathcal{R} containing the pair (E, F) .

\approx is the largest weak bisimulation and an equivalence relation (see [16]).

Security Properties. In this section, we recall from [10] the *Persistent_BNDC* (*P_BNDC*, for short) security property and its characterization in terms of weak bisimulation up to high level actions. We start by recalling the definition of *Bisimulation-based Non Deducibility on Compositions* (*BNDC*, for short) [7]. The *BNDC* security property aims at guaranteeing that no information flow from the high to the low level is possible, even in the presence of malicious processes. The main motivation is to protect a system also from internal attacks, which could be performed by the so called *Trojan Horse* programs, i.e., programs that pretend/appear to be honest but incorporate some malicious code.

Property *BNDC* is based on the idea of checking the system against all high level potential interactions, representing every possible high level malicious program. In particular, a system E is *BNDC* if for every high level process Π a low level user cannot distinguish E from $(E|\Pi)$, i.e., if Π cannot interfere [11] with the low level execution of the system E .

Definition 2 (BNDC). Let $E \in \mathcal{E}$.

$$E \in \text{BNDC} \text{ iff } \forall \Pi \in \mathcal{E}_H, E \setminus H \approx (E|\Pi) \setminus H.$$

In [10] it is shown that the *BNDC* property is not strong enough to analyse systems in dynamic execution environments. To deal with these situations, in [10] it has been introduced the security property named *P_BNDC*. Intuitively, a system E is *P_BNDC* if it never reaches insecure states.

Definition 3 (Persistent_BNDC). Let $E \in \mathcal{E}$.

$$E \in \text{P_BNDC} \text{ iff } \forall E' \text{ reachable from } E, E' \in \text{BNDC}.$$

We give a simple example of a *P_BNDC* process. A more expressive example can be found in [10].

Example 1. Consider the process $E_1 = l.h.j.\mathbf{0} + l.(\tau.j.\mathbf{0} + \tau.\mathbf{0})$ where $l, j \in L$ and $h \in H$. E_1 can be proved to be *BNDC*. Indeed, the causality between h and j in the first branch of the process is “hidden” by the second branch $l.(\tau.j.\mathbf{0} + \tau.\mathbf{0})$, which may simulate all the possible interactions with a high level process.

Suppose now that E_1 is moved in the middle of a computation. This might happen when it find itself in the state $h.j.\mathbf{0}$ (after the first l is executed). Now it is clear that this process is not secure, as a direct causality between h and j is present. In particular $h.j.\mathbf{0}$ is not $BNDC$ and this gives evidence that E_1 is not P_BNDC . The process may be “repaired” as follows: $E_2 = l.(h.j.\mathbf{0} + \tau.j.\mathbf{0} + \tau.\mathbf{0}) + l.(\tau.j.\mathbf{0} + \tau.\mathbf{0})$. It may be proved that E_2 is P_BNDC . Note that, from this example it follows that $P_BNDC \subset BNDC$.

In [10] it has been proven that the property P_BNDC is equivalent to the security property $SBSNNI$ [6, 7], which is automatically checkable over finite-state processes.

However, this property still requires a universal quantification over all the possible states that are reachable from the initial process E . In [10] it has been shown that this can be avoided, by including the requirement of “being secure in every state” directly inside the bisimulation equivalence notion.

In particular, an observation equivalence, named *weak bisimulation up to H* , is defined in such a way that actions from H may be ignored, i.e., they may be matched with zero or more τ actions. This bisimulation notion is based on a suitable transition relation $\xrightarrow{\hat{a}}_{\setminus H}$ which does not take care of both internal actions and actions from H .

Definition 4. Let $E, E' \in \mathcal{E}$ and $a \in Act$. We define the transition relation $\xrightarrow{\hat{a}}_{\setminus H}$ as follows:

$$E \xrightarrow{\hat{a}}_{\setminus H} E' = \begin{cases} E \xrightarrow{\hat{a}} E' & \text{if } a \notin H \\ E \xrightarrow{a} E' \text{ or } E \xrightarrow{\hat{\tau}} E' & \text{if } a \in H \end{cases}$$

Note that the relation $\xrightarrow{\hat{a}}_{\setminus H}$ is a generalization of the relation $\xrightarrow{\hat{a}}$ used in the definition of weak bisimulation [16]. In fact, if $H = \emptyset$, then for all $a \in Act$, $E \xrightarrow{\hat{a}}_{\setminus H} E'$ coincides with $E \xrightarrow{\hat{a}} E'$.

The concept of weak bisimulation up to H is defined as follows.

Definition 5 (Weak Bisimulation up to H). A binary relation $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ over agents is a weak bisimulation up to H if $(E, F) \in \mathcal{R}$ implies, for all $a \in Act$,

- (1) if $E \xrightarrow{a} E'$, then there exists F' such that $F \xrightarrow{\hat{a}}_{\setminus H} F'$ and $(E', F') \in \mathcal{R}$;
- (2) if $F \xrightarrow{a} F'$, then there exists E' such that $E \xrightarrow{\hat{a}}_{\setminus H} E'$ and $(E', F') \in \mathcal{R}$.

Two agents $E, F \in \mathcal{E}$ are weakly bisimilar up to H , written $E \approx_{\setminus H} F$, if $(E, F) \in \mathcal{R}$ for some weak bisimulation \mathcal{R} up to H .

The relation $\approx_{\setminus H}$ is the largest weak bisimulation up to H and it is an equivalence relation.

In [10] it has been proved that P_BNDC can be characterized in terms of $\approx_{\setminus H}$ as follows.

Theorem 1. Let $E \in \mathcal{E}$. Then, $E \in P_BNDC$ iff $E \approx_{\setminus H} E \setminus H$.

3 Defining P_BNDC Processes

In this section we define a transformation on processes which maps an arbitrary process into a P_BNDC one. Moreover, we show that a process is P_BNDC iff it is weak bisimilar up to H to its transformed version. In order to prove this second result we exploit some basic properties of weak bisimulation up to H which are introduced in Section 3.1.

3.1 Basic Properties of $\approx_{\setminus H}$

We start by proving some properties of the relation $\approx_{\setminus H}$. Actually the relation $\approx_{\setminus H}$ enjoys the majority of the properties of the standard weak bisimulation.

First of all $\approx_{\setminus H}$ is coarser than \approx .

Lemma 1. *If $E \approx F$, then $E \approx_{\setminus H} F$.*

Proof. Let $\mathcal{S} = \{(E, F) \mid E \approx F\}$. The binary relation \mathcal{S} is a weak bisimulation up to H since for all processes E it holds that if $E \xrightarrow{\hat{a}} E'$, then $E \xrightarrow{\hat{a}}_{\setminus H} E'$. \square

The relation $\approx_{\setminus H}$ is compositional with respect to the restriction on high level actions, as stated by the following lemma.

Lemma 2. *If $E \approx_{\setminus H} F$, then $E \setminus H \approx_{\setminus H} F \setminus H$.*

Proof. Let $\mathcal{S} = \{(E \setminus H, F \setminus H) \mid E \approx_{\setminus H} F\}$. It is easy to prove that \mathcal{S} is a weak bisimulation up to H . \square

The P_BNDC class of processes is closed with respect to the equivalence relation of $\approx_{\setminus H}$.

Lemma 3. *Let $E, F \in \mathcal{E}$. If $E \approx_{\setminus H} F$ and $F \in P_BNDC$, then $E \in P_BNDC$.*

Proof. If $E \approx_{\setminus H} F$, then we obtain

$$\begin{aligned} E &\approx_{\setminus H} F \\ &\approx_{\setminus H} F \setminus H \text{ by Theorem 1} \\ &\approx_{\setminus H} E \setminus H \text{ by Lemma 2} \end{aligned}$$

Hence, since $E \approx_{\setminus H} E \setminus H$, by Theorem 1 we obtain that E is P_BNDC . \square

By Lemma 1 and Lemma 3 it immediately follows that if $E \approx F$ and $F \in P_BNDC$, then $E \in P_BNDC$.

Another useful property of P_BNDC processes is that restriction and hiding with respect to high level actions yield weakly bisimilar processes.

Lemma 4. *If $E \in P_BNDC$, then $E \setminus H \approx E/H$.*

Proof. This is a consequence of the fact that P_BNDC is equivalent to the $SBSNNI$ property [10] and $SBSNNI$ implies that $E \setminus H \approx E/H$ [6, 7]. \square

3.2 The τ Completion of a Process

Now we are ready to define our first transformation over finite-state processes which maps a process E into a P -BNDC process E^τ .

Definition 6 (τ completion of \mathbf{E}). Let $E \in \mathcal{E}$ be one of the processes defined by a system of equations S . The τ completion of E is the process E^τ defined by the system S^τ , where:

- if $F = \mathbf{0}$ is in S , then $F^\tau = \mathbf{0}$ is in S^τ ;
- if $F = \sum_{i \in I} l_i.F_i + \sum_{j \in J} h_j.F_j$ is in S , with $l_i \in L \cup \{\tau\}$ and $h_j \in H$, then $F^\tau = \sum_{i \in I} l_i.F_i^\tau + \sum_{j \in J} h_j.F_j^\tau + \sum_{j \in J} \tau.F_j^\tau$ is in S^τ .

In practice, the LTS associated to E^τ can be obtained from the LTS of E by simply adding a τ edge whenever there is a transition with a label in H .

Example 2. Consider the process E defined by the following system S :

$$\begin{cases} E = h.F + l_1.\mathbf{0} \\ F = l_2.E \end{cases}$$

Using the above definition we obtain the process E^τ defined by

$$\begin{cases} E^\tau = h.F^\tau + \tau.F^\tau + l_1.\mathbf{0} \\ F^\tau = l_2.E^\tau \end{cases}$$

The two LTS's for E and E^τ are depicted in Fig. 2. Different edges between the same nodes are represented in a compact way with a single arc labeled by a sequence of actions separated by commas.

The following lemma formalizes the relations between E and E^τ , which are at the basis of all the results in this section.

Lemma 5. Let $E \in \mathcal{E}$.

1. if $E^\tau \xrightarrow{a} E'$ with $a \neq \tau$, then there exists E_1 such that $E' = E_1^\tau$ and $E \xrightarrow{a} E_1$;
2. if $E^\tau \xrightarrow{\tau} E'$, then there exists E_1 such that $E' = E_1^\tau$ and $E \xrightarrow{k} E_1$ with $k \in H \cup \{\tau\}$;
3. if $E \xrightarrow{a} E_1$ with $a \in H \cup L \cup \{\tau\}$, then $E^\tau \xrightarrow{a} E_1^\tau$;

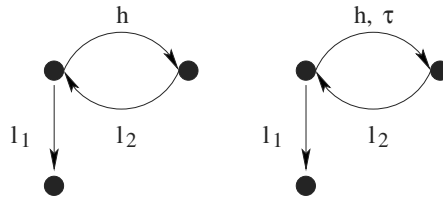


Fig. 2. The LTS's representing E and E^τ

4. if $E^\tau \xrightarrow{h} E'$ with $h \in H$, then $E^\tau \xrightarrow{\tau} E'$.

Proof. It immediately follows by Definition 6. \square

The difference between E and E^τ is that whenever E can perform a high action, E^τ can silently simulate the same reduction, thus hiding the high level actions to the low level user. We prove that E^τ so defined is P_BNDC .

Theorem 2. For any process $E \in \mathcal{E}$, $E^\tau \in P_BNDC$.

Proof. Let $\mathcal{S} = \{(E^\tau, E^\tau \setminus H) \mid E \in \mathcal{E}\}$. It is easy to prove that \mathcal{S} is a weak bisimulation up to H . The result follows by Theorem 1. \square

The following lemma shows that E and E^τ behave in the same way if we hide the high level actions.

Lemma 6. For any process $E \in \mathcal{E}$ it holds that $E/H \approx E^\tau/H$.

Proof. Let $\mathcal{S} = \{(E/H, E^\tau/H) \mid E \in \mathcal{E}\}$. By Lemma 5, it is easy to prove that \mathcal{S} is a weak bisimulation. \square

The previous result is not sufficient to guarantee that the transformation preserves the semantics of the process, at least from the low level user point of view. In fact, what the low level user can observe are the restrictions $E \setminus H$ and $E^\tau \setminus H$. The following theorem identifies the class of processes for which the transformation preserves the low level semantics.

Theorem 3. Let $E \in \mathcal{E}$. $E \setminus H \approx E^\tau \setminus H$ iff $E \setminus H \approx E/H$.

Proof. If $E \setminus H \approx E^\tau \setminus H$, then

$$\begin{aligned} E \setminus H &\approx E^\tau \setminus H && \text{by hypothesis} \\ &\approx E^\tau/H && \text{by Theorem 2 and Lemma 4} \\ &\approx E/H && \text{by Lemma 6.} \end{aligned}$$

If $E \setminus H \approx E/H$, then

$$\begin{aligned} E \setminus H &\approx E/H && \text{by hypothesis} \\ &\approx E^\tau/H && \text{by Lemma 6} \\ &\approx E^\tau \setminus H && \text{by Theorem 2 and Lemma 4.} \end{aligned}$$

\square

The processes satisfying $E \setminus H \approx E/H$ are studied in [7] and form the class of $BSNNI$ processes. In [7] it is also shown that the class of P_BNDC processes (there called $SBSNNI$) is properly included in the class of $BSNNI$ processes.

In a certain sense we have the feeling that E^τ is a straight completion of E in order to obtain a P_BNDC process, and that if E is P_BNDC then E must be *not too far from* (in strong connection with) E^τ . In the rest of this section we study which is this connection. First, we show that P_BNDC properly includes the class of processes which are weakly bisimilar to their τ completion and it is properly included in the class of processes whose H restriction is weak bisimilar to the restriction of their completion.

Proposition 1. *Let $E \in \mathcal{E}$. The following properties hold:*

- (1) *if $E \approx E^\tau$ then $E \in P_BNDC$;*
- (2) *if $E \in P_BNDC$ then $E \setminus H \approx E^\tau \setminus H$.*

Proof. (1) By Theorem 2 we have that E^τ is P_BNDC , hence by Lemma 3 we have the thesis. (2) This is a corollary of Theorem 3 and of Lemma 4. \square

Note that $E \in P_BNDC$ does not imply $E \approx E^\tau$. As an example consider the process $E = h.\mathbf{0}$ which is P_BNDC but it is not weak bisimilar to $E^\tau = h.\mathbf{0} + \tau.\mathbf{0}$. Moreover, $E \setminus H \approx E^\tau \setminus H$ does not imply $E \in P_BNDC$. This is a consequence of the fact that, by Theorem 3, $E \setminus H \approx E^\tau \setminus H$ is equivalent to the $BSNNI$ property which, as already said, is weaker than P_BNDC (see [7]).

The following theorem shows that if we use the relation $\approx_{\setminus H}$ instead of \approx we obtain the desired characterization of P_BNDC processes.

Theorem 4. *Let $E \in \mathcal{E}$. Then, $E \in P_BNDC$ iff $E \approx_{\setminus H} E^\tau$.*

Proof. (\Rightarrow)

$$\begin{aligned}
 E &\approx_{\setminus H} E \setminus H && \text{by Theorem 1} \\
 &\approx_{\setminus H} E/H && \text{by Lemma 4} \\
 &\approx_{\setminus H} E^\tau/H && \text{by Lemma 6} \\
 &\approx_{\setminus H} E^\tau \setminus H && \text{by Lemma 4} \\
 &\approx_{\setminus H} E^\tau && \text{by Theorem 1.}
 \end{aligned}$$

(\Leftarrow) By Theorem 2, we have that $E^\tau \in P_BNDC$, hence, by Lemma 3, we obtain the thesis. \square

4 Checking P_BNDC

By Theorem 4, it follows that in order to check whether a process E is P_BNDC we can equivalently check whether $E \approx_{\setminus H} E^\tau$. The first question should be whether this test is decidable or not; then it is necessary to study the complexity of a decision algorithm. Instead of defining an ad hoc algorithm we prefer here to prove that it is possible to reduce the test of weak bisimilarity up to H to a test of weak bisimilarity¹. Hence, we define a second transformation over processes which maps a process E into a process E^H in such a way that $E \approx_{\setminus H} F$ iff $E^H \approx F^H$. Since the transformation can be performed in linear time we obtain that the test of weak bisimilarity up to H is in the same complexity class of the test of weak bisimilarity.

Definition 7. *Let $E \in \mathcal{E}$ be one of the processes defined by a system of equations S . E^H is the process defined by the system S^H , where:*

- *if $F = \mathbf{0}$ is in S , then $F^H = \sum_{h \in H} h.F^H$ is in S^H ;*

¹ Note that weak bisimilarity is usually tested through strong bisimulation on transformed processes (see [4]).

– if $F = \sum_{i \in I} a_i.F_i + \sum_{j \in J} \tau.F_j$ is in S , with $a_i \neq \tau$, then $F^H = \sum_{i \in I} a_i.F_i^H + \sum_{j \in J} \tau.F_j^H + \sum_{j \in J, h \in H} h.F_j^H + \sum_{h \in H} h.F^H$ is in S^H .

The LTS associated to E^H can be obtained from the LTS of E by adding all the possible H transitions to any τ transition and all the possible H self-loops.

Example 3. Let $H = \{h_1, h_2\}$ and consider the process E defined by the following system

$$\begin{cases} E = \tau.F \\ F = h_1.E \end{cases}$$

We have that E^H is process defined by the following system

$$\begin{cases} E^H = \tau.F^H + h_1.F^H + h_2.F^H + h_1.E^H + h_2.E^H \\ F^H = h_1.E^H + h_1.F^H + h_2.F^H \end{cases}$$

Similarly to Lemma 5 in the previous section, the following lemma formalizes the relations between E and E^H . Its proof follows by Definitions 4 and 7.

Lemma 7. *Let $E \in \mathcal{E}$.*

- (1) if $E^H \xrightarrow{a} E'$ with $a \notin H$, then there exists E_1 such that $E' = E_1^H$ and $E \xrightarrow{a} E_1$;
- (2) if $E^H \xrightarrow{h} E'$ with $h \in H$ then there exists E_1 such that $E' = E_1^H$ and $E \xrightarrow{k} E_1$ with $k \in \{h\} \cup \{\tau\}$;
- (3) if $E^H \xrightarrow{\tau} E'$ then $E^H \xrightarrow{h} E'$ for all $h \in H$;
- (4) if $E \xrightarrow{a} E_1$ then $E^H \xrightarrow{a} E_1^H$ for any action a ;
- (5) if $E \xrightarrow{\tau} E_1$ then $E^H \xrightarrow{h} E_1^H$ for all $h \in H$;
- (6) if $E \xrightarrow{\hat{a}}_{\setminus H} E_1$ then $E^H \xrightarrow{\hat{a}} E_1^H$ for any action a ;
- (7) if $E \xrightarrow{\hat{\tau}}_{\setminus H} E_1$ then $E^H \xrightarrow{\hat{h}} E_1^H$ for all $h \in H$;
- (8) if $E^H \xrightarrow{\hat{a}} E_1^H$ then $E \xrightarrow{\hat{a}}_{\setminus H} E_1$ for any action a .

We are now ready to prove the main result of this section which shows that we can reduce the test of $\approx_{\setminus H}$ to a test of \approx .

Theorem 5. *Let $E, F \in \mathcal{E}$. Then, $E \approx_{\setminus H} F$ iff $E^H \approx F^H$.*

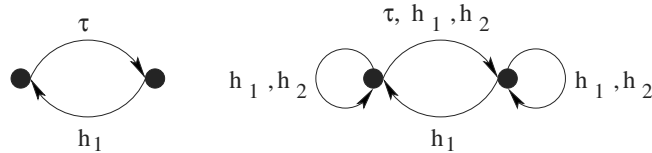


Fig. 3. The LTS's associated to E and E^H

Proof. (\Rightarrow) Let $\mathcal{S} = \{(E^H, F^H) \mid E \approx_{\setminus H} F\}$. By Lemma 7, it is easy to prove that \mathcal{S} is a weak bisimulation. (\Leftarrow) Let $\mathcal{S} = \{(E, F) \mid E^H \approx F^H\}$. By Lemma 7, it is easy to prove that \mathcal{S} is a weak bisimulation up to H . \square

The results presented so far show that the τ completion of a process E is a P_BNDC process which can be used both to check whether E is P_BNDC and in case it is not to *rectify* it. Moreover, both the construction of E^τ and the P_BNDC test performed using E^τ have a low time complexity, as stated by the following result.

Theorem 6. *Let $T(n_1, n_2, m_1, m_2)$ be the time complexity of an algorithm to test $F_1 \approx F_2$ where $n_{1,2}$ is the number of nodes in $LTS(F_{1,2})$ and $m_{1,2}$ is the number of edges in $LTS(F_{1,2})$. It is possible to check if $E \in P_BNDC$ in time $T(n, n, m^H, m^{\tau H}) + O(n + m^\tau)$, where n is the number of nodes in $LTS(E)$, m^H is the number of edges in $LTS(E^H)$, m^τ is the number of edges in $LTS(E^\tau)$, and $m^{\tau H}$ is the number of edges in $LTS((E^\tau)^H)$.*

Proof. This is an immediate consequence of the following facts:

- $LTS(E^H)$ and $LTS(E^\tau)$ have the same number of nodes of $LTS(E)$;
- $LTS(E^H)$ can be computed through a visit of $LTS(E)$;
- $LTS(E^\tau)$ can be computed through a visit of $LTS(E)$;
- $LTS((E^\tau)^H)$ can be computed through a visit of $LTS(E^\tau)$;
- the number of edges in $LTS(E^\tau)$ is greater than the number of edges in $LTS(E)$, hence $O(n + m) + O(n + m^\tau) = O(n + m^\tau)$, where m is the number of edges in $LTS(E)$.

\square

Notice that if m is the number of edges in $LTS(E)$, then $m^\tau \leq m + m_H$, where m_H is the number of edges in $LTS(E)$ labelled with a high action. Moreover, $m^H \leq m + H * m_\tau$, where m_τ is the number of edges in $LTS(E)$ labelled with a τ action. Hence, $m^{\tau H} \leq m + H * (m_\tau + m_H)$. However, in order to check $E^H \approx (E^\tau)^H$ it is not really necessary to explicitly compute E^H and $(E^\tau)^H$, since it is sufficient to build a simple interface for the bisimulation algorithm which reinterprets the labels of the transitions. More precisely, for instance, the set of processes which can evolve into E^H with a h action is equal to the union of the set of processes which can evolve into E with a τ or a h action.

5 An Example

In this section we illustrate through an example how the τ completion can be used to rectify a process which is neither P_BNDC nor $BSNNI$.

Note that, as the system is not $BSNNI$, then it is neither $BNDC$, i.e., it is insecure even with respect to non-dynamic environments. Moreover, the fact that the system is not $BSNNI$ implies (by Theorem 3) that the low level semantics is not preserved by the τ completion. However, we will see that the way low level

semantics is changed is very reasonable and only affects some deadlock states caused by high level activity.

Consider the process C described through a value-passing extension of SPA by

$$C = in(x).\overline{out}(x).C$$

C is a channel which may accept a value x at the left-hand port, labelled in . When it holds a value, it may deliver it at the right-hand port, labelled \overline{out} . If the domain of x is $\{0,1\}$, then the channel C can be translated into SPA in a standard way by following [16] as:

$$C = in_0.\overline{out}_0.C + in_1.\overline{out}_1.C$$

Let us assume that C is used as communication channel from low to high level. This can be expressed as $in_0, in_1 \in L$ and $\overline{out}_0, \overline{out}_1 \in H$.

Note that such a channel should be secure as it provides a “legal” information flow from low to high. However, we show that this is not the case. If we compute C^τ we obtain

$$C^\tau = in_0.(\overline{out}_0.C^\tau + \tau.C^\tau) + in_1.(\overline{out}_1.C^\tau + \tau.C^\tau)$$

Moreover, C^H and $(C^\tau)^H$ are respectively

$$\begin{aligned} C^H &= in_0.\overline{out}_0.C^H + in_1.\overline{out}_1.C^H + \overline{out}_0.C^H + \overline{out}_1.C^H \\ (C^\tau)^H &= in_0.(\overline{out}_0.(C^\tau)^H + \tau.(C^\tau)^H + \overline{out}_1.(C^\tau)^H) + \\ &\quad in_1.(\overline{out}_1.(C^\tau)^H + \tau.(C^\tau)^H + \overline{out}_0.(C^\tau)^H) + \\ &\quad \overline{out}_0.(C^\tau)^H + \overline{out}_1.(C^\tau)^H \end{aligned}$$

It is immediate to see that they are not weak bisimilar, since $(C^\tau)^H$ can silently reset itself after every input action, while C^H must always execute the corresponding output. Hence C is not P_BNDC . Intuitively, a high level user can indefinitely block the process C after each low level input by just refusing to accept the corresponding output (remind that communication is synchronous). The potential high level deadlocks could be exploited to transmit information as shown, e.g., in [7].

Now, by Theorem 2 we can replace C by C^τ thus obtaining a P_BNDC process. Intuitively C^τ is P_BNDC as the presence of “resetting” τ transitions avoids the high level deadlocks mentioned above.

These τ 's basically makes the channel a lossy one, as high level outputs may be non-deterministically lost. However, note that non-determinism is used to abstract away implementation details. For example, such τ 's could correspond, at implementation time, to time-outs for the high output actions, i.e., events that empty the channel and allow a new low level input, whenever high outputs are not accepted within a certain amount of time. In this respect, it would be quite interesting to rephrase our theory to models enriched with time or probability as [2, 12, 8], in order to study how the τ completion instantiate to more concrete settings. Even if the resulting process behaves differently from the low level point

of view (C is not *BSNNI*), we think that C^τ can be reasonably proposed as a secure rectifying of C .

Indeed, note that the only difference, from a low level perspective, is the absence in $C^\tau \setminus H$ of deadlock states after the low level input actions. Such states of $C \setminus H$ are exactly the cause of potential information flows in process C , as they provide a causality between high level activity (i.e., accepting or not high outputs) and low level one.

In general when we define E^τ from a given process E and we add a τ transition relative to a high level output, this can always be seen as the insertion of a time-out. While in the case we add a τ transition relative to a high level input this corresponds to generating a non-deterministic high input. This latter case is clearly less reasonable. Hence, our transformation seems to be appropriate for fixing flows related to high level outputs. A rectifying strategy could be to add only the τ transitions relative to the outputs and check whether this is sufficient to have a *P_BNDC* process.

6 Conclusions

In the recent years, issues concerning security have received an increasing attention due to the augmented possibilities of interconnections and information exchanges. A number of formal definitions of security properties has been proposed in the literature.

In this paper we consider the security property *P_BNDC* based on the idea of Non-Interference [11, 19, 22] which has been deeply studied in [10] and showed to be suitable to guarantee security in a dynamically reconfigurable context. We present a method to automatically construct a *P_BNDC* process by a transformational approach. We show that the transformation preserves the low level observational semantics of *BSNNI* processes. Moreover we illustrate on an example how the transformation produces reasonable corrections also for non *BSNNI* processes where a modification of the low level semantics is necessary in order to ensure security. We show that our transformation can be used also to efficiently check the *P_BNDC* property, exploiting existing tools for bisimulation.

We are presently trying to apply our techniques to more significant examples in order to establish their effectiveness in producing secure systems from insecure ones. Moreover, it would be interesting to have a measure of what security damage would be in case *P_BNDC* does not hold. In [18], it is proposed a way of classifying information flow properties depending on which kind of channels from high to low level are implementable from systems that do not satisfy such properties. For example, obtaining a “perfect” channel represents a damage worse than, e.g., obtaining a noisy one. It could be interesting to measure the strengtness of *P_BNDC* with respect to this kind of classification.

References

- [1] M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM*, 46(5):749–786, 1999. 271
- [2] A. Aldini. Probabilistic information flow in a process algebra. In *Proc. of the 12th International Conference on Concurrency Theory (CONCUR'01)*, pages 152–168. Springer LNCS 2154, August 2001. 283
- [3] C. Bodei, P. Degano, R. Focardi, and C. Priami. Primitives for Authentication in Process Algebras. *Theoretical Computer Science*, 2001. To appear. 271
- [4] T. Bolognesi and S. A. Smolka. Fundamental results for the verification of observational equivalence: A survey. In H. Rudin and C. H. West, editors, *Proc. of Int'l Conference on Protocol Specification, Testing and Verification (PSTV'87)*, pages 165–179. North-Holland, 1987. 280
- [5] N. A. Durgin, J. C. Mitchell, and D. Pavlovic. A Compositional Logic for Protocol Correctness. In *Proc. of of the 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001. 271
- [6] R. Focardi and R. Gorrieri. The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997. 276, 277
- [7] R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of LNCS. Springer, 2001. 271, 272, 273, 275, 276, 277, 279, 280, 283
- [8] R. Focardi, R. Gorrieri, and F. Martinelli. Information-flow analysis in a discrete-time process algebra. In *Proc. of of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001. 283
- [9] R. Focardi, C. Piazza, and S. Rossi. Proof methods for bisimulation based information flow security. In A. Cortesi, editor, *Proc. of Int. Workshop on Verification, Model Checking and Abstract Interpretation*, LNCS. Springer, 2002. 272
- [10] R. Focardi and S. Rossi. Information flow security in dynamic contexts. In *Proc. of of the 15th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2002. 271, 272, 273, 275, 276, 277, 284
- [11] J. A. Goguen and J. Meseguer. Security Policy and Security Models. In *Proc. of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982. 271, 272, 275, 284
- [12] Jan Jürjens. Secure information flow for concurrent processes. In *Proc. of International Conference on Concurrency Theory (Concur 2000)*. LNCS 1877, Springer-Verlag, August 2000. 283
- [13] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. of of the 14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001. 271
- [14] D. McCullough. “A Hookup Theorem for Multilevel Security”. *IEEE Transactions on Software Engineering*, pages 563–568, June 1990. 271
- [15] J. McLean. “A General Theory of Composition for Trace Sets Closed Under Selective Interleaving Functions”. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1994. 271
- [16] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 273, 274, 275, 276, 283
- [17] L. C. Paulson. Proving Properties of Security Protocols by Induction. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997. 271

- [18] Roberto Gorrieri Riccardo Focardi and Roberto Segala. A new definition of multilevel security. In *Proc. of Workshop on Issues in the Theory of Security (WITS '00)*, (P. Degano, ed.), July 2000. 284
- [19] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 200–215. IEEE Computer Society Press, 2000. 271, 284
- [20] S. Schneider. Verifying Authentication Protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9), 1998. 271
- [21] V. Shmatikov and J. C. Mitchell. Analysis of a Fair Exchange Protocol. In *Proc. of 7th Annual Symposium on Network and Distributed System Security (NDSS 2000)*, pages 119–128. Internet Society, 2000. 271
- [22] G. Smith and D. M. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proc. of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98)*, pages 355–364. ACM Press, 1998. 271, 284

On Bisimulations for the Spi Calculus^{*}

Johannes Borgström¹ and Uwe Nestmann²

¹ KTH, Sweden

² EPFL, Switzerland

Abstract. The spi calculus is an extension of the pi calculus with cryptographic primitives, designed for the verification of cryptographic protocols. Due to the extension, the naive adaptation of labeled bisimulations for the pi calculus is too strong to be useful for the purpose of verification. Instead, as a viable alternative, several “environment-sensitive” bisimulations have been proposed. In this paper we formally study the differences between these bisimulations.

1 Introduction

The spi calculus, proposed by Martín Abadi and Andrew Gordon in [AG99] as an extension of the pi calculus [Mil99], is a process calculus designed for the description and formal verification of cryptographic protocols.

According to [AG99], many correctness properties for cryptographic protocols are naturally expressed through may-testing equivalences between certain process terms, but proofs of such properties are notoriously hard due to the requirement of infinitary quantifications (usually quantifications over infinitely many process contexts). In contrast, the standard notion of bisimulation as an equivalence on process terms provides a coinductive proof technique, usually avoiding infinitary quantifications. The interest in bisimulation notions for the spi calculus follows from the fact that bisimilarity is a sound (but not complete) approximation to may-testing equivalence.

Bisimulation is based on the idea of an environment observing a pair of processes to see whether it may distinguish the two processes. The observations are derived from the operational semantics of processes giving rise to labeled transition systems. In usual process calculi, the two points of view of an observing environment and of an observed process are symmetric: any transition that a process can do according to its semantics is also observable by the environment. This symmetry is no longer valid in the case of the spi calculus.

In Figure 1, we highlight some meaningful and meaningless transitions in the spi calculus compared to the pi calculus. The various labels represent the input $a b$ of a name b along channel a , the output $\bar{a} b$ of a message b along channel a , or the bound counterpart $(\nu b)\bar{a} b$ for a fresh name b , and the (bound) output $(\nu b k)\bar{a}\langle E_k(b)\rangle$ along channel a of a message $E_k(b)$ representing the encryption of cleartext b using key k . The displayed transitions represent the possible observations about a process from the point of view of an environment interacting with

^{*} Supported by the Swiss National Science Foundation, grant No. 21-65180.01.

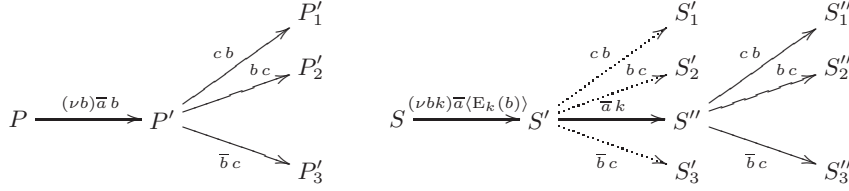


Fig. 1. Environment transitions in pi and spi calculi

the process. For example, an environment observing P might see the bound output $(\nu b)\bar{a}b$, which at the same time means that the environment itself performs a respective input operation. Afterwards, note that the process has now evolved into P' , the fresh name b received by the environment may itself be used to interact with P' in the various ways displayed. Essentially, once the environment receives a name, it may freely use it in interactions. The situation is different in the spi calculus, where exchanged messages may be encrypted, as it happens in the transition from S to S' . Note that both the key k and the datum b are bound, but when S passes on the encrypted message, it does not also give away the encryption key as an accessible datum. Therefore, none of the dotted transitions is possible from within S' : since the environment does not know the key k itself, it cannot interact with S' using the cleartext (the name b) hidden inside the ciphertext $E_k(b)$, neither to communicate *on* b nor sending back b to the process. However, this becomes possible in S'' after S' has exposed the key k explicitly to the environment as shown in the transition from S' to S'' .

Summing up the previous examples, a proper treatment of transitions with respect to bisimulation in the spi calculus must, in contrast to the pi calculus, explicitly take into account the *knowledge* of an environment about a process. As a means to capture this environment knowledge, the notion of *environment-sensitive* bisimulation has been developed for the spi calculus, in various styles:

- Abadi and Gordon introduced *framed* bisimulation [AG98] by imposing on every bisimulation pair a common *frame-theory* pair that represents the knowledge of the environment about the pair. The frame is the set of names (channels, keys) that the environment has learned so far, while the theory is the set of pairs of non-name data items received from the pair of processes during the bisimulation game that the environment must believe to be “the same”, because it has no means (i.e., decryption keys) to distinguish them.
- Boreale et al. introduced another notion under the generic name of environment-sensitive bisimulation itself [BDP99]. Here, each of the processes in a bisimulation pair is accompanied by a separate environment (roughly) listing the messages that a process and an environment have exchanged in the past. In this paper, we call their variant *alley* bisimulation, pictorially reminding of this separation. To express the identification of non-distinguishable data items, an explicit condition of *equivalent environments* is imposed.

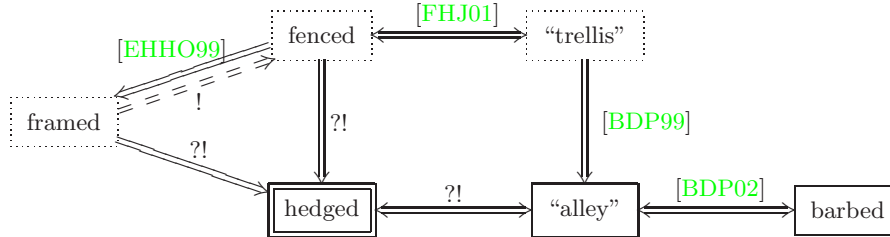


Fig. 2. Comparing bisimulations

- Elkjær et al. introduced *fenced* bisimulation, an approximation to framed bisimulation by getting rid of one of its infinitary quantifications [EHHO99].

All of the above notions of bisimulation are, assuming that observing environments know all free names, sound approximations of may-testing equivalence.

Comparing Bisimulations The immediate question on the various competing notions of bisimulation is (1) how they relate to each other, and (2) how each of them relates to *barbed equivalence*, which is a uniformly defined contextual notion of bisimulation that is usually considered prime among all bisimilarities [MS92]. So far, these questions have only been treated in parts. Boreale et al. proved that alley bisimilarity, for a wide range of processes, is a sound and complete approximation of barbed equivalence [BDP99]. In contrast, Abadi and Gordon were already aware that their notion of framed bisimulation is strictly stronger than barbed equivalence, at least when the spi calculus contains a pairing construct [AG98]; without pairing, no result has been published. Elkjær et al. proposed that fenced bisimulation would coincide with framed bisimulation [EHHO99], but their proof is flawed (see below): framed bisimilarity is not contained in fenced bisimilarity. Finally, Frendrup et al. recently showed that fenced bisimulation and a strengthened form of alley bisimulation (also discussed in [BDP99]; here, we call it “trellis” bisimulation) coincide [FHJ01].

In Figure 2, we pictorially summarize the various relations, which lets us also clearly discuss the scientific contribution of the current paper.

Contributions In this paper, we formally highlight the differences between the above-mentioned environment-sensitive bisimulations by introducing an improved form of framed bisimulation, called *hedged* bisimulation. In summary, all of the question marks in Figure 2 are supported by proofs for positive results and counterexamples for negative results. In particular, we prove that hedged bisimulation coincides with alley bisimulation, and thus also with barbed equivalence. However, being defined in the style of framed and fenced bisimulation, hedged bisimulation allows us to formally and much more intuitively assess the differences of the former notions, as indicated in Figure 2. We also exhibit, by means of a counterexample, that fenced bisimulation is not complete w.r.t. framed bisimulation, in contrast to the results of [EHHO99]. (Moreover, we developed a semantic framework for the comparison of environment-sensitive bisimulations

and rephrase the results in terms of simple category theoretical notions by means of equivalent categories, but due to space limitations, we leave the exposition to the long version of the paper [BN02].)

Conclusions As an interpretation of the results, we may underline two different deficiencies in the original definition of framed bisimulation. In a sense, it is at the same time too weak and too strong with respect to barbed equivalence, for orthogonal reasons. (1) The definition is too weak in the sense that its authors did not impose a minimality requirement on the environment and argue that this “results in simpler definitions, and does not compromise soundness (w.r.t. testing equivalence)”. However, when adding the minimality requirement, as done in fenced bisimulation, then the relation becomes strictly stronger. According to our respective example in Section 4.1, we may even be inclined to state that framed bisimulation allows to “cheat”. Thus, fenced bisimulation could be regarded as the better framed bisimulation. However, fenced (like framed) bisimilarity suffers from yet another problem: (2) The definition is too rigid, because it requires the syntactic coincidence of names received by the environment from the two processes under observation: whereas framed bisimulation requires identity, alley bisimulation allows the environment to simply record that the names respectively received from the processes in a bisimulation step correspond (c.f. Section 4.2). The lack of this distinction captures exactly why fenced bisimulation coincides with “trellis” bisimulation.

When proving concrete example processes equivalent, we find that hedged bisimulation is easier to work with compared to alley bisimulation, because the knowledge available to the environment in each reachable configuration is arguably easier to understand. We are also currently developing a symbolic semantics for the spi calculus as the basis of a notion of symbolic bisimulation. To this aim, the use of hedged bisimulation seems again favorable to the use of alley bisimulation, because the involved data structures are easier to deal with. Furthermore, an ultimate implementation of a bisimulation-checker is likely to profit from the minimality requirements on hedges.

Related Work All of the work on bisimulations for the spi calculus that we are aware of has been mentioned in this Introduction.

Overview The definition of the spi calculus used in this paper is presented in Section 2. Definitions of the environment-sensitive bisimulations can be found in Section 3. In Section 4 we exhibit some examples showing the differences between framed, fenced and hedged bisimulation. In Section 5 we summarize the positive results concerning the comparison of the various bisimulation of this paper.

2 Language

The spi calculus used in this paper is the one used by Boreale et al. in [BDP02], with some changes in notation. Furthermore, we build on the same assumptions on the underlying system of shared-key cryptography, which we do not repeat here. We also assume the reader to have some familiarity with the pi calculus.

Table 1. Syntax of the spi calculus

$a, b, c \dots, k, l, m, n \dots, x, y, z$	names \mathcal{N}
$\zeta, \eta ::= a \mid E_\zeta(\zeta) \mid D_\zeta(\zeta)$	expressions \mathcal{E}
$\delta ::= a \mid E_\delta(\delta)$	decryption-free expressions \mathcal{D}
$M, N ::= a \mid E_k(M)$	messages \mathcal{M}
$\phi, \psi ::= tt \mid \phi \wedge \phi \mid \neg\phi$	guards \mathcal{G}
$\mid \text{let } z = \zeta \text{ in } \phi$	(decryption)
$\mid \text{is_name}(\delta)$	(is a name)
$\mid [\delta = \delta]$	(equality)
$P, Q ::= \mathbf{0}$	processes \mathcal{P}
$\mid \delta(x).P$	(input prefix)
$\mid \bar{\delta}(\delta).P$	(output prefix)
$\mid P + P$	(choice)
$\mid P \mid P$	(parallel)
$\mid (\nu a)P$	(restriction)
$\mid !P$	(replication)
$\mid \phi P$	(boolean guard)
$\mid \text{let } x = \zeta \text{ in } P$	(decryption)

Syntax We assume a countably infinite set \mathcal{N} of names. Names are untyped, meaning that the same name can be used as a channel, a key, a variable or the clear-text of a message. The lower case letters $a, b, c, k, l, m, n, x, y, z$ are used to range over names. In examples, a, b, c are used for channels, k, l for keys, m, n for messages and x, y, z for variables when their subsequent usage is not explicit.

The syntax of expressions, guards, and processes, is given in Table 1.

In contrast to the pi calculus, the spi calculus offers next to mere names another kind of transmissible messages, namely *ciphertexts*, which are provided by the addition of primitive constructs to encrypt ($E(\cdot)$) and decrypt ($D(\cdot)$) data using a shared-key cryptographic system. Encryptions can be arbitrarily nested, but (in contrast to [AG99]) only proper names can be used as encryption keys. While expressions ζ are formed arbitrarily using the encryption and decryption operators, messages M represent proper decryption-free ciphertexts where the encryption keys are names. The role of decryption-free expressions δ is to formalize that decryption constructors can only occur within let-constructs, as indicated by the occurrences of ζ . This property will be preserved by the operational semantics later on.

Logical formulae ϕ generalize the usual equality operator of the pi calculus by conjunction and negation. Moreover, the predicate $\text{is_name}(\delta)$ tests for the format of δ , i.e., whether it is a plain name or not. The formula $\text{let } z = \zeta \text{ in } \phi$ binds the value of expression ζ , produced by evaluation as defined in Subsection 2, to the name z within formula ϕ .

Processes are formed as in the pi calculus, except for the following aspects: Input and output forms have to take into account that the channel and message positions might be (decryption-free) expressions; however, only names in channel position make sense, otherwise the process will be stuck. Guarded processes represent the generalized matching construct. Like in formulae, we also have a

Table 2. Evaluation (decryption) in the spi calculus

$\llbracket a \rrbracket$	$= a$
$\llbracket E_\zeta(\eta) \rrbracket$	$= \begin{cases} E_k(M) & \text{if } \llbracket \eta \rrbracket = M \in \mathcal{M} \text{ and } \llbracket \zeta \rrbracket = k \in \mathcal{N} \\ \perp & \text{otherwise} \end{cases}$
$\llbracket D_\zeta(\eta) \rrbracket$	$= \begin{cases} M & \text{if } \llbracket \eta \rrbracket = E_k(M) \in \mathcal{M} \text{ and } \llbracket \zeta \rrbracket = k \in \mathcal{N} \\ \perp & \text{otherwise} \end{cases}$
$\llbracket tt \rrbracket$	$= tt$
$\llbracket \phi \wedge \psi \rrbracket$	$= \llbracket \phi \rrbracket \wedge \llbracket \psi \rrbracket$
$\llbracket \neg\psi \rrbracket$	$= \neg\llbracket \psi \rrbracket$
$\llbracket \text{let } z = \zeta \text{ in } \phi \rrbracket$	$= \begin{cases} \llbracket \phi\{^M/z\} \rrbracket & \text{if } \llbracket \zeta \rrbracket = M \in \mathcal{M} \\ ff & \text{otherwise} \end{cases}$
$\llbracket \text{is_name}(\zeta) \rrbracket$	$= \begin{cases} tt & \text{if } \zeta \in \mathcal{N} \\ ff & \text{otherwise} \end{cases}$
$\llbracket \llbracket \zeta = \eta \rrbracket \rrbracket$	$= \begin{cases} tt & \text{if } \zeta = \eta \in \mathcal{M} \\ ff & \text{otherwise} \end{cases}$
$(\text{GUARD}) \frac{P \xrightarrow{\mu} P'}{\phi P \xrightarrow{\mu} P'} \text{ if } \llbracket \phi \rrbracket = tt \quad (\text{LET}) \frac{P\{\llbracket \zeta \rrbracket / z\} \xrightarrow{\mu} P'}{\text{let } z = \zeta \text{ in } P \xrightarrow{\mu} P'} \text{ if } \llbracket \zeta \rrbracket \neq \perp$	

let-construct on processes. In this paper we leave out the treatment of tuples in messages; see the long version for details on this extension.

As usual abbreviation, in single-letter outputs forms we sometimes omit the brackets, as in $\bar{a}M.0$. Free and bound names of terms are inductively defined as expected: a is bound in “ $(\nu a)P$ ”, x is bound in “ $\delta(x).P$ ”, in “let $x = \zeta$ in P ”, and in “let $x = \zeta$ in ϕ ”. Two processes are α -equivalent if they can be made equal by conflict-free renaming of bound names. Substitutions σ are mappings $\{^M/x\}$ from names x to messages M , obeying the usual assumption that name-capture is avoided through implicit α -conversion, if necessary. Substitutions are applied to processes, expressions and guards in the straightforward way: for example, $P\{^M/x\}$ replaces all free occurrences of x in P by M , possibly renaming bound names in P to avoid name capture.

Semantics To define operational semantics we need to be able to evaluate both expressions and boolean guards. The evaluation function for expressions $\llbracket \cdot \rrbracket : \mathcal{E} \rightarrow \mathcal{M} \cup \{\perp\}$ (where $\perp \notin \mathcal{M}$) and guards $\llbracket \cdot \rrbracket : \mathcal{G} \rightarrow \{tt, ff\}$ is defined recursively (see Table 2).

Note that let- and guard-constructs are the only constructs that perform evaluation of expressions, so all decryption must take place there.

The operational semantics (see Table 2) follows [BDP02]. It uses an early input style semantics, which is standard from the pi calculus, except for the rules (GUARD) and (LET) which provide the only attempts to decrypt messages. A process ϕP behaves like P provided that ϕ evaluates to true; otherwise, ϕP is stuck. A process let $z = \zeta$ in P behaves like $P\{\llbracket \zeta \rrbracket / z\}$ provided that the evaluation of ζ succeeds; otherwise, let $z = \zeta$ in P is stuck. Symmetric variants of (PAR), (COM) and (SUM) are not explicated. By (ALPHA) we have that α -equivalent processes

have the same transitions, so all relations on processes based on transitions will also be defined up to α -equivalence.

3 Environment-Sensitive Bisimulations

Ordinary bisimulations relate process pairs (P, Q) , but in environment-sensitive bisimulations \mathcal{S} we relate triples (e, P, Q) , also written $e \vdash P \mathcal{S} Q$, where e is called *environment*. Such bisimulations relations are described intuitively as follows: Q simulates P under e if every labeled transition $P \rightarrow P'$ that can be detected by e can be matched by a similarly labeled transition $Q \rightarrow Q'$ such that Q' simulates P' under an updated environment e' . To relate two processes P and Q , one usually wants to find a bisimulation \mathcal{S} such that $e \vdash P \mathcal{S} Q$ and $\text{fn}(P, Q)$ is “contained” in e , i.e., known by e .

In order to capture meaningful interactions between an attacker and the process pair, e must accurately model the knowledge of an attacker. Usually, e is a data structure that represents a set of messages that the attacker has been able to accumulate, plus some information about which messages received from P must be considered indistinguishable from messages correspondingly received from Q . The knowledge of e is the set of all pairs of messages (related by non-distinguishability) that the attacker can synthesize from e . Thus, the notion of *synthesis of e* plays a crucial role in the definitions below for testing whether an attacker can use a certain channel or produce a certain message. To properly calculate the synthesis it is also necessary to check whether some of the encrypted messages of e might be decrypted via other available messages in e . This is captured in the notion of *analysis*, which essentially reduces e to the set of its *irreducibles*, i.e., the messages that the attacker cannot currently decrypt further. Finally, the notion of *consistency* guarantees a variety of well-formedness criteria on the data structures representing e .

Depending on the type of the data structure e introduced further below (frame-theory pairs, hedges, or substitution pairs) the notions of analysis, synthesis, and consistency appear in different forms or only implicitly, which makes it tricky to compare them. Irreducibility may be enforced on the data structure. Synthesis may be expressed explicitly by means of substitution on expressions.

Whenever $\mathcal{R} \subseteq \mathbf{E} \times \mathcal{P} \times \mathcal{P}$ is an environment-sensitive relation for some kind of environments \mathbf{E} , we define $\mathcal{R}^{-1} := \{(e^{-1}, Q, P) \mid (e, P, Q) \in \mathcal{R}\}$ for some suitably defined inversed environment e^{-1} . We write that $e \vdash P \mathcal{R} Q$ if $(e, P, Q) \in \mathcal{R}$, otherwise $e \not\vdash P \mathcal{R} Q$. \mathcal{R} is *symmetric* if $\mathcal{R} = \mathcal{R}^{-1}$.

3.1 Framed and Fenced Bisimulations

Framed bisimulation, as introduced by Abadi and Gordon [AG98], is the first environment-sensitive bisimulation that was proposed for the spi calculus. The original definition was for a strong and late bisimulation. Here, we study a weak and early variant in order to sharpen the comparison with the definition of bisimulation in [BDP99, BDP02]. Abadi and Gordon also used a different calculus, with a complex set of messages containing integers, pairing and general encryption keys but without general guards, choice and general “let”.

In framed bisimulation the environment consists of a frame and a theory. A frame is a set of names known to the environment. A theory is a set of pairs of messages considered equivalent by the environment.

Definition 1. A frame is a finite subset of \mathcal{N} . A theory is a finite subset of $\mathcal{M} \times \mathcal{M}$. \mathbf{FT} is the set of all frame-theory pairs. If $B \subset \mathcal{M}$ is finite then we define $\text{Id}_B := \{(b, b) \mid b \in B\}$. If th is a theory, we define $\text{th}^{-1} := \{(N, M) \mid (M, N) \in \text{th}\}$, $\pi_1(\text{th}) := \{M \mid (M, N) \in \text{th}\}$ and $\pi_2(\text{th}) := \{N \mid (M, N) \in \text{th}\}$. The names of a theory is defined as $\text{n}(\text{th}) := \text{n}(\pi_1(\text{th}) \cup \pi_2(\text{th}))$.

A frame-theory pair is *consistent* if the theory only contains pairs of encrypted messages that the environment can not decrypt. The environment can not consider a given message equivalent to two different messages.

Definition 2. A frame-theory pair (fr, th) is consistent iff for all messages M and N such that $(M, N) \in \text{th}$ we have that

1. $M, N \notin \mathcal{N}$
2. If $(M', N') \in \text{th}$ then $M = M' \iff N = N'$
3. If $M = E_a(M')$ and $N = E_b(N')$ then $\text{fr} \cap \{a, b\} = \emptyset$.

The *synthesis* $\mathcal{S}(\cdot)$ of a frame-theory pair is the set of message pairs constructed by encrypting message pairs from the theory with keys from the frame. The environment considers equivalent any message pair in the synthesis.

Definition 3. If (fr, th) is a frame-theory pair, we let $\mathcal{S}(\text{fr}, \text{th})$ be the smallest subset of $\mathcal{M} \times \mathcal{M}$ containing $\text{th} \cup \text{Id}_{\text{fr}}$ and satisfying

$$\text{(SYN-ENC)} \quad \frac{(M, N) \in \mathcal{S}(\text{fr}, \text{th}) \quad (a, b) \in \mathcal{S}(\text{fr}, \text{th})}{(E_a(M), E_b(N)) \in \mathcal{S}(\text{fr}, \text{th})}$$

We write $(\text{fr}, \text{th}) \vdash M \leftrightarrow N$ for $(M, N) \in \mathcal{S}(\text{fr}, \text{th})$; otherwise $(\text{fr}, \text{th}) \not\vdash M \leftrightarrow N$.

A *framed process pair* is a triple $((\text{fr}, \text{th}), P, Q)$ where fr is a frame, th is a theory and P and Q are processes. A *framed relation* \mathcal{R} is a set of framed process pairs. \mathcal{R} is *consistent* if (fr, th) is consistent whenever $(\text{fr}, \text{th}) \vdash P \mathcal{R} Q$.

Definition 4. A consistent framed relation \mathcal{R} is a framed simulation if whenever $(\text{fr}, \text{th}) \vdash P \mathcal{R} Q$ we have that

1. If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $(\text{fr}, \text{th}) \vdash P' \mathcal{R} Q'$.
2. If $P \xrightarrow{aM} P'$, $a \in \text{fr}$, $B \subset \mathcal{N}$ is finite, $B \cap (\text{fn}(P, Q) \cup \text{fr} \cup \text{n}(\text{th})) = \emptyset$, $N \in \mathcal{M}$, and $(\text{fr} \cup B, \text{th}) \vdash M \leftrightarrow N$, then there exists Q' such that $Q \xrightarrow{aN} Q'$ and $(\text{fr} \cup B, \text{th}) \vdash P' \mathcal{R} Q'$
3. If $P \xrightarrow{(\nu \bar{c}) \bar{a} M} P'$, $a \in \text{fr}$, and $\{\bar{c}\} \cap (\text{fn}(P) \cup \text{fr} \cup \text{n}(\pi_1(\text{th}))) = \emptyset$, then there exist Q', N, \tilde{d} with $\{\tilde{d}\} \cap (\text{fn}(Q) \cup \text{fr} \cup \text{n}(\pi_2(\text{th}))) = \emptyset$ and
 - (a) $Q \xrightarrow{(\nu \tilde{d}) \bar{a} N} Q'$, and
 - (b) there exist fr', th' with $\mathcal{S}(\text{fr}, \text{th}) \subseteq \mathcal{S}(\text{fr}', \text{th}')$ and $(\text{fr}', \text{th}') \vdash M \leftrightarrow N$ such that $(\text{fr}', \text{th}') \vdash P' \mathcal{R} Q'$.

\mathcal{R} is a framed bisimulation if both \mathcal{R} and \mathcal{R}^{-1} are framed simulations.

It is worth noting how the new environment after output transitions is characterized: names and message pairs can be freely added as long as the synthesis is extended conservatively and the new output messages are kept indistinguishable.

Fenced bisimulation was defined by Elkjær et al. in [EHHO99], where it was proved to be a sound and complete approximation to framed bisimulation. (In Section 4, we show that this is in fact not the case.) The difference between the definitions is that fenced bisimulation replaces the existential quantification over frames and theories in case 3.(b) of Definition 4 by means of a *function* ξ that extends a given frame-theory pair with a new pair of messages. Since our message grammar is simpler the definition of ξ can be simplified for our case. In essence, the function ξ decrypts the new messages as far as possible using the knowledge of the environment, recursively adding the previous messages if a new key is obtained. The definition is straightforward, so we omit its presentation; the curious reader may find it in the full version. In [EHHO99], the authors show that ξ provides a *minimal* consistent extension whenever there exists one.

Definition 5. *A consistent framed relation \mathcal{R} is a fenced simulation if whenever $(\text{fr}, \text{th}) \vdash P \mathcal{R} Q$ we have that (as in Definition 4, except for the following case).*

3. (b) such that $\xi(\text{fr}, \text{th}, M, N) \vdash P' \mathcal{R} Q'$.

\mathcal{R} is a fenced bisimulation if both \mathcal{R} and \mathcal{R}^{-1} are fenced simulations.

3.2 Alley and Trellis Bisimulations

In [BDP99, BDP02], Boreale et al. define environment-sensitive semantics and a corresponding weak bisimulation for the spi calculus. As mentioned earlier, their bisimulation is called *alley* in this paper in order to distinguish it from the other bisimulations, which are also environment-sensitive. The authors also prove that alley bisimulation is a sound approximation of barbed equivalence, and that the approximation is complete for the class of “structurally image-finite” processes.

Formally, Boreale et al. define two levels of operational semantics, one for the behavior of processes, and another one for the corresponding behavior of environments. As in [FHJ01], we adapt this formalization to the style where the environment behavior is built into the definition of bisimulation.

In alley bisimulation, the environment is a pair of substitutions.

Definition 6. *A substitution σ is a finite partial function $\mathcal{N} \rightarrow \mathcal{M}$. We write $\sigma\{M/x\}$ for $\sigma \cup \{(x, M)\}$, where $x \notin \text{dom}(\sigma)$. Similarly, $\sigma\{M_1/x_1, \dots, M_n/x_n\}$ stands for $\sigma \cup \{(x_i, M_i) \mid i = 1, 2, \dots, n\}$ where the x_i are assumed to be pairwise different and not in $\text{dom}(\sigma)$. The set of free names of a substitution is defined as $\text{fn}(\sigma) := \text{n}(\text{range}(\sigma))$. We denote by **SS** the set of all alleys, i.e., the set of all substitution pairs.*

Any set of messages, e.g., the messages mentioned in the components of an alley, might be reduced via decryption using the notion of analysis.

Definition 7. The analysis $\mathcal{A}(S)$ and the irreducibles $\mathcal{I}(S)$ of a set $S \subseteq \mathcal{M}$ are defined as follows: $\mathcal{A}(S)$ is the smallest subset of \mathcal{M} containing S and satisfying

$$\text{(SET-DEC)} \quad \frac{E_a(M) \in \mathcal{A}(S) \quad a \in \mathcal{A}(S)}{M \in \mathcal{A}(S)}$$

and $\mathcal{I}(S) := \mathcal{A}(S) \setminus \{E_a(M) \mid a \in \mathcal{A}(S)\}$.

We introduce a function that decrypts messages M as far as possible, i.e., peeling out the *core* of M using the knowledge of a substitution σ . We use the shorthands $\mathcal{I}(\sigma)$ for $\mathcal{I}(\text{range}(\sigma))$ and $\mathcal{A}(\sigma)$ for $\mathcal{A}(\text{range}(\sigma))$. Let $\text{core}_\sigma(M)$ denote $\text{core}_\sigma(M')$ if $M = E_a(M')$ and $a \in \mathcal{I}(\sigma)$, otherwise it denotes M . Thus, we can decompose any message M into $E_{b_n}(\dots E_{b_2}(E_{b_1}(\text{core}_\sigma(M)))) \dots$ for any substitution σ with $\{b_1 \dots, b_n\} \subseteq \mathcal{I}(\sigma)$; if $\text{core}_\sigma(M) = E_a(N)$, then $a \notin \mathcal{I}(\sigma)$. As a special case, $\mathcal{C}(\sigma, x) := \text{core}_\sigma(\sigma(x))$. Therefore, $\mathcal{I}(\sigma) = \{\mathcal{C}(\sigma, x) \mid x \in \text{dom}(\sigma)\}$.

Two substitutions are consistent if they decrypt corresponding messages in precisely corresponding ways. In other words, it does not suffice to just decrypt to corresponding cores, but we also must use corresponding keys for the decryption.

Definition 8. A pair of substitutions (σ, ρ) is consistent, written $\sigma \sim \rho$, iff σ and ρ have the same domain $\{x_1 \dots, x_n\}$ and the following conditions hold:

1. $\mathcal{C}(\sigma, x_i) \in \mathcal{N} \iff \mathcal{C}(\rho, x_i) \in \mathcal{N}$
2. $\mathcal{C}(\sigma, x_i) = \mathcal{C}(\sigma, x_j) \iff \mathcal{C}(\rho, x_i) = \mathcal{C}(\rho, x_j)$
3. For each $i = 1, 2, \dots, n$ there is a tuple $\tilde{t} = t_1 \dots t_n$ such that

$$\begin{aligned} \sigma(x_i) &= E_{\mathcal{C}(\sigma, x_{i_m})}(\dots E_{\mathcal{C}(\sigma, x_{i_2})}(E_{\mathcal{C}(\sigma, x_{i_1})}(\mathcal{C}(\sigma, x_i))) \dots) \\ \rho(x_i) &= E_{\mathcal{C}(\rho, x_{i_m})}(\dots E_{\mathcal{C}(\rho, x_{i_2})}(E_{\mathcal{C}(\rho, x_{i_1})}(\mathcal{C}(\rho, x_i))) \dots) \end{aligned}$$

An alley process pair is a triple $((\sigma, \rho), P, Q)$ with $\text{dom}(\sigma) = \text{dom}(\rho)$. An *alley relation* \mathcal{R} is a set of alley process pairs. \mathcal{R} is *consistent* if $(\sigma, \rho) \vdash P \mathcal{R} Q$ implies that $\sigma \sim \rho$.

Definition 9. A consistent alley relation \mathcal{R} is an alley simulation if whenever $(\sigma, \rho) \vdash P \mathcal{R} Q$ the following conditions hold:

1. If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \implies Q'$ and $(\sigma, \rho) \vdash P' \mathcal{R} Q'$.
2. If $P \xrightarrow{aM} P'$ and there are η, ζ, \tilde{b} such that $\llbracket \eta\sigma \rrbracket = a, \llbracket \zeta\sigma \rrbracket = M$ with $\text{fn}(\eta) \subseteq \text{dom}(\sigma)$, $\tilde{b} = \text{fn}(\zeta) \setminus \text{dom}(\sigma)$ and $\tilde{b} \cap \text{fn}(P, Q, \rho, \sigma) = \emptyset$, then there exist \tilde{c}, Q' with $\tilde{c} \subset \mathcal{N}$, $|\tilde{c}| = |\tilde{b}|$, $\tilde{c} \cap \text{dom}(\sigma) = \emptyset$ such that $Q \xrightarrow{\llbracket \eta\rho \rrbracket \llbracket \zeta\rho \rrbracket} Q'$ and $(\sigma\{\tilde{b}/\tilde{c}\}, \rho\{\tilde{b}/\tilde{c}\}) \vdash P' \mathcal{R} Q'$.
3. If $P \xrightarrow{(\nu\tilde{c})\tilde{a}M} P'$ with $\text{fn}(P, \sigma) \cap \{\tilde{c}\} = \emptyset$ and if there is η such that $\text{fn}(\eta) \subseteq \text{dom}(\sigma)$ and $\llbracket \eta\sigma \rrbracket = a$ then there are Q', N, \tilde{d}, b, x with $\llbracket \eta\rho \rrbracket = b$ and $\text{fn}(Q, \rho) \cap \{\tilde{d}\} = \emptyset$ such that $Q \xrightarrow{(\nu\tilde{d})\tilde{b}N} Q'$ and $(\sigma\{M/x\}, \rho\{N/x\}) \vdash P' \mathcal{R} Q'$.

\mathcal{R} is an alley bisimulation if both \mathcal{R} and \mathcal{R}^{-1} are alley simulations.

Note the difference with respect to the previous bisimulations. Here, the environment is extended by mechanically just adding the new messages (for output) or the new names (for input) instead of also reducing the environment to contain only irreducible messages. Using the environment as true substitutions, consistency plays a vital role: e.g., it imposes the corresponding syntheses of messages (the input value is precisely generated from ζ using σ and ρ), and by adding output messages M and N bound to the same variable x , consistency also insists in the (minimal!) extension that M and N correspond according to Definition 8.3.

Trellis bisimulation is a strengthened variant of alley bisimulation, studied (not under this name) in [BDP99]. There, two different notions of consistency of environments were studied. One of them was rejected since it was considered too strong. It constitutes the basis for trellis bisimulation.

Definition 10. *A consistent pair of substitutions $\sigma \sim \rho$ is strongly consistent, written $\sigma \sim_s \rho$, if $\mathcal{C}(\sigma, x) \in \mathcal{N}$ implies that $\mathcal{C}(\sigma, x) = \mathcal{C}(\rho, x)$.*

This resembles the definition of consistency of frame-theory pairs in that two different names may never be considered equal. Strong consistency has a corresponding bisimulation, called *trellis* in this paper, that was defined and compared to fenced bisimulation in [FHJ01]. We recapitulate and strengthen the results of the comparison in Section 5.

Definition 11. *An alley relation \mathcal{R} is strongly consistent if $(\sigma, \rho) \vdash P \mathcal{R} Q$ implies $\sigma \sim_s \rho$. We call trellis bisimulation a strongly consistent alley bisimulation.*

3.3 Hedged Bisimulation

Hedged bisimulation is introduced in this paper in order to clarify the differences between framed, fenced, and alley bisimulation. Recall that alley bisimulation, unlike its counterparts, does not force two processes to always send the same names; it rather remembers that the respective names correspond to each other. The basic idea of hedges is to mimic the lack of correspondence in frame-theory pairs by dropping the separate frame component, but to include corresponding names as part of the theory. The resulting theory is called a *hedge*.

Definition 12. *A hedge is a theory, i.e., a finite subset of \mathcal{M}^2 . We denote by \mathbf{H} the set of all hedges. The synthesis $\mathcal{S}(\cdot)$ of a hedge is defined as $\mathcal{S}(h) = \mathcal{S}(\emptyset, h)$. We write $h \vdash M \leftrightarrow N$ for $(M, N) \in \mathcal{S}(h)$, $h \not\vdash M \leftrightarrow N$ otherwise.*

A hedge is consistent if the hedge only contains pairs of names and pairs of encrypted messages that can not be decrypted by the environment. We also require that no message is considered to be equivalent to two different messages.

Definition 13. *A hedge h is consistent iff whenever $(M, N) \in h$*

1. $M \in \mathcal{N} \iff N \in \mathcal{N}$
2. If $(M', N') \in h$ then $M = M' \iff N = N'$
3. If $M = E_a(M')$ and $N = E_b(N')$ then $a \notin \pi_1(h)$ and $b \notin \pi_2(h)$.

The difference between a consistent hedge and a consistent frame-theory pair is that we do not require that the hedge receives the same names from both processes, so they do not need to use the same channels and encryption keys. The difference between a consistent hedge and a consistent substitution pair is that the former only contains undecryptable messages (i.e., cores) and that no duplicate message pairs are allowed. The third condition for consistent substitutions (cf. Definition 8) roughly corresponds to the definition of hedge analysis.

Definition 14. *The analysis $\mathcal{A}(h)$ and the irreducibles $\mathcal{I}(h)$ of a hedge h are defined as follows: $\mathcal{A}(h)$ is the smallest subset of \mathcal{M}^2 containing h and satisfying*

$$\text{(HEDGE-DEC)} \frac{(\mathbb{E}_a(M), \mathbb{E}_b(N)) \in \mathcal{A}(h) \quad (a, b) \in \mathcal{A}(h)}{(M, N) \in \mathcal{A}(h)}$$

and $\mathcal{I}(h) \stackrel{\text{def}}{=} \mathcal{A}(h) \setminus \{(\mathbb{E}_a(M), \mathbb{E}_b(N)) \mid (a, b) \in \mathcal{A}(h) \wedge M, N \in \mathcal{M}\}$.

The analysis of hedges decrypts pairs of messages using pairs of names that are considered equivalent by the environment. The resulting notion of irreducibles in fact also corresponds to the ξ -function of fenced bisimulation.

Now that the environment and notions of consistency are defined, the definition of hedged bisimulation is straightforward. A *hedged relation* \mathcal{R} is a subset of $\mathbf{H} \times \mathcal{P} \times \mathcal{P}$. We say that \mathcal{R} is *consistent* if $h \vdash P \mathcal{R} Q$ implies that h is consistent.

Definition 15. *A consistent hedged relation \mathcal{R} is a hedged simulation if whenever $h \vdash P \mathcal{R} Q$ we have that*

1. If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $h \vdash P' \mathcal{R} Q'$.
2. If $P \xrightarrow{aM} P'$, $h \vdash a \leftrightarrow b$, $B \subset \mathcal{N}$ is finite, $B \cap (\text{fn}(P, Q) \cup \text{n}(h)) = \emptyset$, $N \in \mathcal{M}$, and $h \cup \text{Id}_B \vdash M \leftrightarrow N$, then there exists Q' such that $Q \xrightarrow{bN} Q'$ and $h \cup \text{Id}_B \vdash P' \mathcal{R} Q'$.
3. If $P \xrightarrow{(\nu \bar{c}) \bar{a} M} P'$, $h \vdash a \leftrightarrow b$ and $\{\bar{c}\} \cap (\text{fn}(P) \cup \text{n}(\pi_1(h))) = \emptyset$ there exist Q', N, \bar{d} with $\{\bar{d}\} \cap (\text{fn}(Q) \cup \text{n}(\pi_2(h))) = \emptyset$ such that $Q \xrightarrow{(\nu \bar{d}) \bar{b} N} Q'$ and $\mathcal{I}(h \cup \{(M, N)\}) \vdash P' \mathcal{R} Q'$.

\mathcal{R} is a hedged bisimulation if both \mathcal{R} and \mathcal{R}^{-1} are hedged simulations.

On process output we use $\mathcal{I}(\cdot)$ to construct the new hedge after the transition. This entails applying all decryptions that the environment can do, producing—as in fenced bisimulation—the minimal extension of the hedge h with (M, N) .

3.4 Bisimilarities

As usual, the largest bisimulations are called *bisimilarities*; they are denoted $\approx_{\mathbf{f}}$ (framed), $\approx_{\#}$ (fenced), $\approx_{\mathbf{a}}$ (alley), $\approx_{\mathbf{s}}$ (trellis), and $\approx_{\mathbf{h}}$ (hedged). All of them are equivalence relations on processes (when fixing environments in compositions).

4 Distinguishing Examples

4.1 Fenced vs Framed/Hedged

The following example distinguishes fenced bisimulation from its competitors due to a subtle requirement concerning the data involved in the simulation of output transitions, especially the conditions on the choice of bound names:

$$\begin{aligned} P &:= (\nu nkl) \bar{a}\langle E_l(E_k(n)) \rangle.P' & P' &:= (\nu m) \bar{a}\langle m \rangle.\mathbf{0} \\ Q &:= (\nu nk) \bar{a}\langle E_k(n) \rangle.Q' & Q' &:= (\nu m) \bar{a}\langle m \rangle.\mathbf{0} \end{aligned}$$

Although there is no reason for P and Q to be distinguished, fenced bisimulation does so because it insists that single fresh names be simulated without renaming.

We write $\text{pwd}(\tilde{n})$ to denote that \tilde{n} is a tuple of pairwise different names.

Proposition 1. $(\{a\}, \emptyset) \not\vdash P \approx_{\#} Q$.

Proof. The transitions of P are of the form $P \xrightarrow{(\nu nkl) \bar{a}\langle E_l(E_k(n)) \rangle} P'$ where $\text{pwd}(n, k, l, a)$. The transitions of Q are of the form $Q \xrightarrow{(\nu n'k') \bar{a}\langle E_{k'}(n') \rangle} Q'$ where $\text{pwd}(n', k', a)$. We then get the theory $\{(E_l(E_k(n)), E_{k'}(n'))\}$. Since n, k, l are pairwise different, there must be a name $z \in \{n, k, l\} \setminus \{n', k'\}$.

Now P' must simulate the transition $Q' \xrightarrow{(\nu z) \bar{a}z} \mathbf{0}$. We have $P' \xrightarrow{(\nu m) \bar{a}m} \mathbf{0}$ for all $m \neq a$. For the resulting frame-theory pair to be consistent, we must have $m = z$, but since $z \in n(\pi_1(\text{th}))$ this transition can not be used. \square

Fenced bisimulation fails since it cannot simulate an output of a particular bound name when this name is already known by the simulating environment. Here, both framed and hedged bisimulation succeed, but in different ways.

Proposition 2. $(\{a\}, \emptyset) \vdash P \approx_f Q$.

Proof. A framed bisimulation relating P and Q is given by

$$\begin{aligned} \mathcal{R} &= \{(\{a\}, \emptyset, P, Q)\} \\ &\cup \{(\{a, k\}, \{(E_l(E_k(n)), E_l(n))\}, P', Q') \mid \text{pwd}(a, k, l, n)\} \\ &\cup \{(\{a, k, m\}, \{(E_l(E_k(n)), E_l(n))\}, \mathbf{0}, \mathbf{0}) \mid \text{pwd}(a, k, l, m, n)\} \end{aligned}$$

Note the addition of k to the frame, which relieves the process P' from simulating the critical bound output of z by Q' (see the previous proof). The name created by Q' must be different from the names in the current frame and theory, so by simply adding k to the frame—which is allowed in framed bisimulation, but due to the minimality property not computed in fenced bisimulation—this name must be chosen different from k , and thus P' may also create it. \square

Proposition 3. $\{(a, a)\} \vdash P \approx_h Q$.

Proof. A hedged bisimulation relating P and Q is given by

$$\begin{aligned} \mathcal{R} &= \{(\{(a, a)\}, P, Q)\} \\ &\cup \{(\{(a, a), (E_l(E_k(n)), E_l(n))\}, P', Q') \mid \text{pwd}(a, k, l, n)\} \\ &\cup \{(\{(a, a), (E_l(E_k(n)), E_l(n)), (m, m)\}, \mathbf{0}, \mathbf{0}) \mid \text{pwd}(a, k, l, n, m)\} \\ &\cup \{(\{(a, a), (E_l(E_k(n)), E_l(n)), (w, k)\}, \mathbf{0}, \mathbf{0}) \mid \text{pwd}(a, k, l, n, w)\} \end{aligned}$$

Note the addition of (w, k) to the hedge. \square

In comparison, we may conclude that \approx_f equates the processes through a non-minimal extension of the frame (which could be considered “cheating”), while \approx_h equates them (more adequately) through correspondence.

Corollary 1. \approx_f is not a subset of $\approx_\#$.

4.2 Framed vs Hedged

The following example exhibits a striking deficiency of framed bisimulation that is remedied by hedged bisimulation: a frame-theory pair can distinguish between the plaintext of an encrypted message (n in the example) and another random piece of data (m in the example).

$$\begin{aligned} P &:= (\nu k, n) \bar{a}\langle E_k(n) \rangle. P' & P' &:= (\nu m) \bar{a}\langle m \rangle. \mathbf{0} \\ Q &:= (\nu k, n) \bar{a}\langle E_k(n) \rangle. Q' & Q' &:= \bar{a}\langle n \rangle. \mathbf{0} \quad \text{where } n \neq a. \end{aligned}$$

We show that $\{(a, a)\} \vdash P \approx_h Q$ and $(\{a\}, \emptyset) \not\vdash P \approx_f Q$.

Proposition 4. $\{(a, a)\} \vdash P \approx_h Q$

Proof. The required hedged bisimulation is given by

$$\begin{aligned} \mathcal{R} = & \{(\{(a, a)\}, P, Q)\} \\ & \cup \{(\{(a, a), (E_k(n), E_k(n))\}, P', Q') \mid \text{pwd}(a, k, n)\} \\ & \cup \{(\{(a, a), (E_k(n), E_k(n)), (m, n)\}, \mathbf{0}, \mathbf{0}) \mid \text{pwd}(a, k, m, n)\} \end{aligned}$$

Again, note that in the above \mathcal{R} the names m and n just correspond. They do not give access to further distinctions by means of decryption, and there is no way to distinguish themselves through further interaction with process pairs. \square

For framed bisimulation, we first study the processes P' and Q' .

Proposition 5. *There is no (fr, th) such that $a \in \text{fr}$ and $(\text{fr}, \text{th}) \vdash P' \approx_f Q'$.*

Proof. Assume the opposite, and fix $m \neq n$ such that $m \in \mathcal{N} \setminus (\text{fr} \cup \text{n}(\pi_1(\text{th})))$. As $a \in \text{fr}$ and $P \xrightarrow{(\nu m) \bar{a} m} \mathbf{0}$ there must exist $Q'', N, \tilde{d}, \text{fr}', \text{th}'$ such that $Q' \xrightarrow{(\nu \tilde{d}) \bar{a} N} Q''$, (fr', th') is consistent, and $(\text{fr}', \text{th}') \vdash m \leftrightarrow N$. As the only transition of Q' is $Q' \xrightarrow{\bar{a} n} \mathbf{0}$ we have that $N = n$. Clearly SYN-ENC (cf. Def. 3) can not derive $(\text{fr}', \text{th}') \vdash m \leftrightarrow n$. Since (fr', th') is consistent, which implies that $(m, n) \notin \text{th}'$, we must have that $m = n \in \text{fr}'$, which is a contradiction. \square

Now, we can use the results for P' and Q' to derive results for P and Q .

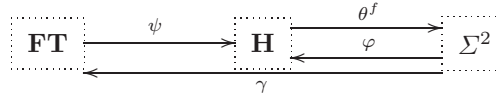
Proposition 6. $(\{a\}, \emptyset) \not\vdash P \approx_f Q$

Proof. As a is in the frame, P and Q can do a matching output step. By Proposition 5, no resulting framed process pair is in \approx_f . \square

5 Comparing Bisimulations

Due to space limitations, we only cite the respective results that we have proved. The full account is found in the long version of the paper, where we also provide a categorical framework that offers even more substantial comparisons.

Any comparison of environment-sensitive bisimulations must be based on a comparison of the various environments. We introduce the following mappings between frame-theory pairs (**FT**), hedges (**H**) and substitution pairs (Σ^2).



To move from frame-theory pairs to hedges, and to move from pairs of substitutions to hedges and frame-theory pairs is done by straightforward mappings:

- Definition 16.** – If $(\text{fr}, \text{th}) \in \mathbf{FT}$, then $\psi(\text{fr}, \text{th}) := \text{Id}_{\text{fr}} \cup \text{th}$.
- If $(\sigma, \rho) \in \Sigma^2$, then $\varphi(\sigma, \rho) := \{ (\mathcal{C}(\sigma, x), \mathcal{C}(\rho, x)) \mid x \in \text{dom}(\sigma) \}$.
 - If $(\sigma, \rho) \in \Sigma^2$, then $\gamma(\sigma, \rho) := (\mathcal{N} \cap \pi_1(\varphi(\sigma, \rho)) , \varphi(\sigma, \rho) \setminus (\mathcal{N} \times \mathcal{N}))$.

To move from hedges to substitutions, we need to invent the domain for the substitutions. Note that it does not matter which particular domain we select. Both $\mathcal{M} \times \mathcal{M}$ and \mathcal{N} are countably infinite, so there is a bijection between them.

Definition 17. Fix a bijection $f : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{N}$. If $h \in \mathbf{H}$, then θ^f defines, w.r.t. f , the left and right substitutions of h as follows:

$$\theta^f(h) := (\{ \{^M /_{f(M,N)} \} \mid (M, N) \in h \} , \{ \{^N /_{f(M,N)} \} \mid (M, N) \in h \})$$

The above mappings must adequately treat the various notions of consistency: for instance, as in the above definition of γ it suffices to consider the projection onto the set of names of $\pi_1(\sigma, \rho)$, if we assume that (σ, ρ) is strongly consistent. The full paper contains a large collection of relevant properties.

Assume that \approx_x and \approx_y are environment-sensitive bisimulations, where E_x and E_y are the sets of environments of \approx_x and \approx_y , respectively. In order to capture the relation between \approx_x and \approx_y , we now use the above-mentioned environment mappings $\lambda : E_x \rightarrow E_y$ to relate process equivalences like in

$$\text{“ } e_x \vdash P \approx_x Q \quad \text{if (and only if)} \quad \lambda(e_x) \vdash P \approx_y Q \text{ ”}$$

with instantiations to the various notions of environment-sensitive bisimulations. Note that the presence of environment mappings requires us to also exhibit results that allow us to transform environments in the other direction. Only then may we state that two notions are truly equivalent. The full paper treats the comparison framework behind this argumentation in much more detail.

Fenced and trellis bisimulation were already compared by Frendrup, Hüttel and Jensen [FHJ01]. We cite it here, because it has not yet been published, and because it only treated the case of strong bisimulation.

Theorem 1. $(\sigma, \rho) \vdash P \approx_s Q$ if and only if $\gamma(\sigma, \rho) \vdash P \approx_{\#} Q$.

This theorem states that fenced bisimilarity “contains” trellis bisimilarity. For the converse, it suffices that every frame-theory pair in $\approx_{\#}$ has a counterpart in \approx_s .

Theorem 2. If (fr, th) is consistent, then $\gamma(\theta^f(\psi(\text{fr}, \text{th}))) = (\text{fr}, \text{th})$.

The correspondence between fenced and framed bisimulation was addressed by Elkjær et. al. [EHHO99], although only for the strong case, but it only holds in one direction as shown in Section 4.1.

Theorem 3. If $(\text{fr}, \text{th}) \vdash P \approx_{\#} Q$ then $(\text{fr}, \text{th}) \vdash P \approx_f Q$.

The correspondence between framed and hedged bisimulation also only holds in one direction, according to the counterexamples in Section 4.2.

Theorem 4. If $(\text{fr}, \text{th}) \vdash P \approx_f Q$ then $\psi(\text{fr}, \text{th}) \vdash P \approx_h Q$.

One problem that arose in proving this theorem was that there is no minimality requirement on the extensions of frame-theory pairs, in contrast with hedges. An example of this is the framed bisimulation in Proposition 2, which extends the frame in a way not permitted to hedged bisimulation. This problem was solved by noting that the operation of throwing away information in a hedge is sound w.r.t. hedged bisimilarity.

By transitivity we get the relation between fenced and hedged bisimilarity, which also can be shown directly. The direct proof is easier, since both fenced and hedged bisimilarity perform minimal extensions on process output.

Corollary 2. If $(\text{fr}, \text{th}) \vdash P \approx_{\#} Q$ then $\psi(\text{fr}, \text{th}) \vdash P \approx_h Q$.

One of the main motivations to study hedged bisimilarity is that it is equivalent to alley bisimilarity and, by transitivity, barbed equivalence.

Theorem 5. $(\sigma, \rho) \vdash P \approx_a Q$ if and only if $\varphi(\sigma, \rho) \vdash P \approx_h Q$.

Theorem 6. $h \vdash P \approx_h Q$ if and only if $\theta^f(h) \vdash P \approx_a Q$.

In the light of these theorems, we may state that hedged bisimilarity is the best possible “framed bisimilarity”, combining the removal of one infinitary quantification of fenced bisimilarity with completeness w.r.t. barbed equivalence.

References

- [AG98] M. Abadi and A. D. Gordon. A Bisimulation Method for Cryptographic Protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998. 288, 289, 293
- [AG99] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Journal of Information and Computation*, 148:1–70, 1999. 287, 291

- [BDP99] M. Boreale, R. De Nicola and R. Pugliese. Proof Techniques for Cryptographic Processes. In *Proceedings of LICS '99*, pages 157–166. IEEE, Computer Society Press, 1999. 288, 289, 293, 295, 297
- [BDP02] M. Boreale, R. De Nicola and R. Pugliese. Proof Techniques for Cryptographic Processes. *SIAM Journal on Computing*, 2002. To appear. 289, 290, 292, 293, 295
- [BN02] J. Borgström and U. Nestmann. On Bisimulation in the Spi Calculus. Draft full version, available from <http://lamp.epfl.ch/~uwe/doc/spi/>, 2002. 290
- [EHHO99] A. S. Elkjær, M. Höhle, H. Hüttel and K. Overgård. Towards Automatic Bisimilarity Checking in the Spi Calculus. In volume 21(3) of *Australian Computer Science Communications*, pages 175–189. Springer, 1999. 289, 295, 302
- [FHJ01] U. Frendrup, H. Hüttel and J. N. Jensen. Two Notions of Bisimilarity for Cryptographic Processes. <http://www.cs.auc.dk/research/FS/ny/PR-pi/ESB/twoNotionsOfESB.ps>, 2001. 289, 295, 297, 301
- [Mil99] R. Milner. *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press, 1999. 287
- [MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In *Proceedings of ICALP '92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992. 289

Specifying and Verifying a Decimal Representation in Java for Smart Cards*

Cees-Bart Breunesse, Bart Jacobs, and Joachim van den Berg

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{ceesb,bart,joachim}@cs.kun.nl

Abstract. This article describes a case study concerning a component of a Java Purse applet developed by the smart card manufacturer Gemplus. This component is a representation of decimal numbers in Java. The decimal component is annotated with specifications consisting of invariants and pre- and postconditions, describing the functional behavior. These specifications are written in the specification language JML. After translation of the annotated source code to the theorem prover PVS, the correctness of these annotations is proved interactively.

1 Introduction

The direct topic of this article is a case study in program specification and verification. As such it gives an impression of the state of the art in tool-assisted verification of realistic programs—in the world of Java smart cards. Also, it describes some of the crucial ingredients of the actual verification process. Such a case study may be interesting, but is in itself of limited interest. The indirect topic, however, is the impact that such specification and verification efforts may have on program development and (ultimately) on certification of smart card applets. In brief, the message is that actual program verification:

- is becoming feasible for Java smart card applets, enabling higher levels of certification and competitive advantages for the manufacturers that apply such techniques;
- does not only lead to correct but also to optimized code—which is very relevant on a smart card with restricted resources. Typically, when class invariants are made explicit, they can be assumed to hold when methods are called, so that various checks in the code can be dropped safely (and provably so). This is not a new insight, see for instance [11, Sect. 11.6], but one which is worth re-emphasizing with a concrete illustration.

* The research presented forms part of the European IST project *VerifiCard*, see www.verificard.org.

Smart cards receive much attention in Europe—mostly as bank cards or as SIMs in GSM cell phones—but recently also in other parts of the world, for instance as general identity card. The European VerifiCard project aims to develop techniques for establishing the correctness for crucial components of the JavaCard platform and for application programs (applets). Such techniques are needed for the evaluation of smart cards at higher levels (within the Common Criteria framework). Evaluation at a high level is for instance required to give a digital signature with a smart card appropriate legal status.

JavaCard is a “superset of a subset” of Java, designed by Sun for programming smart cards: it is a subset in the sense that for example threads, floats, strings, multi-dimensional arrays are not supported. JavaCard is extended with a transaction logging mechanism, and a firewall protection mechanism to control inter-applet communication. The JavaCard API is also a subset of Java’s API, because packages like AWT with GUI classes are obviously not needed. The JavaCard API does have some additional classes, because it also serves as an interface to the operating system of the smart card. These specifics are of no concern to us here, so we use JavaCard as just a stripped-down version of Java.

Within the VerifiCard project, example applets are provided by industrial parties. This article, concentrates on the JavaCard Purse applet by Gemplus (see [1]). This electronic purse is a real-life example of a banking application. It consists of a global balance, the operations to modify this balance, a cryptographic protection mechanism to authenticate and authorize, and a pincode mechanism.

The case study we present in this article considers only a small part of the Gemplus electronic purse, as the complete applet is 65K of source code. One of the many classes in the purse is the Decimal class which provides the numeric representation of a decimal number, e.g. the balance. The Decimal class also contains methods to modify the balance, for example adding, subtracting or multiplying. A balance and methods to modify this balance might not seem an interesting verification challenge at first, but one should keep in mind that in JavaCard there are no floating point numbers available. This means that decimal numbers must be represented in a different fashion. Gemplus solved this problem by taking two shorts (the largest primitive type available in JavaCard) that together make up the decimal number. The Decimal class is an essential component of the Purse. Its (in)correctness thus affects other operations throughout the entire applet. All this makes it a good choice on which to concentrate one’s specification and verification efforts.

The Loop group in Nijmegen works on program specification and verification of Java source code. Verification is done within the theorem prover PVS (see [15]). Getting Java source code and the specifications for Java verified happens via a (shallow) embedding of the Java language and the specification language JML (see [10]) in PVS. For this purpose a compiler called the Loop-tool has been developed which does this translation automatically (see [3]). (A deep embedding of Java into Isabelle has been developed in Munich [14, 13].) The proof obligations in PVS can then be rewritten and simplified by using appro-

priate, provably sound Hoare and Weakest Precondition rules (see [9]). A related approach [12] uses Hoare logic at the Java syntax level to generate verification conditions for PVS.

Proving correctness of Java methods is proving that methods act according to their specification. The first step in establishing the correctness of methods is thus writing specifications. The way we specify Java source code is by using the annotation language JML (see [16]). With JML we write pre- and postconditions for methods, invariants for classes and other clauses to model, for example, exceptional behavior. These annotations are added to the code as a special kind of Java comments. One of the attractive features that JML offers is the wide range of tools, for systematic testing, static checking, or formal verification.

In a companion paper to ours [6, 7] the ESC/Java tool (see [17]) is applied to the same Purse case study, although using different specifications¹. By statically analyzing the source code with ESC/Java, many implementation details are made explicit, such as input checking on parameters, and invariants on the fields. An advantage of this approach is that ESC/Java automatically checks whether the implementation satisfies its specification by iterating the method bodies a number of times. However, this approach is both unsound and incomplete, and, therefore, it will not guarantee that the implementation satisfies its specification in all cases. In other words, it is not a formal proof.

The ideal way to combine these two approaches is to first use ESC/Java to detect the most common programming errors, and then to use the LOOP tool and PVS to actually verify the most critical fragments of the code.

This paper is organized as follows. In the first section we familiarize the reader with JML. In Sect. 2 we describe some members and specify the method for adding Decimals. In Sect. 3 we shortly discuss how the translation from annotated Java source code to PVS is achieved. In Sect. 4 we present the verification of the method for addition. In Sect. 5 we discuss the method for multiplying two Decimals.

1.1 Specifying Java Source Code

This section explains some ins and outs of JML by taking simple parts of the Decimal class and annotating them with JML. First a short note on the Decimal class itself. A decimal number is represented by an object of the Decimal class. A Decimal object has two short fields that together form the actual representation of this decimal number. These shorts are named *intPart* and *decPart*. The first one, *intPart*, represents the integer part. The second one, *decPart* represents the decimal part. Thus, the decimal number 9.715 is represented by an object of class Decimal where *intPart* = 9 and *decPart* = 715.

In Fig. 1 the source code and annotations for a small segment of the Decimal class is depicted. Method *oppose* returns the opposite decimal number.

¹ Typically for ESC/Java, all postconditions are simply True. In our case we have non-trivial postconditions describing the functional behavior of the methods at hand.

```

class Decimal {
2  short intPart, decPart;
   final static short PRECISION = 1000;
4
   //@ invariant PRECISION == 1000;
6
   /*@ normal_behavior
8   @   requires d != null;
   @   modifiable intPart, decPart;
10  @   ensures   intPart == -\old(intPart)
   @             && decPart == -\old(decPart);
12  @*/
   public Decimal oppose(Decimal d) {
14     intPart = -intPart;
       decPart = -decPart;
16     return this;
       }
18 }

```

Fig. 1. Basic JML annotated example: `oppose()`

The code with the JML specification in Fig. 1 can be compiled by a normal Java compiler because the annotations are within Java comments. Note that we use a “@” to distinguish JML annotations from normal comments. The precondition (*requires* clause) says that the argument to *oppose* may not be *null*. After execution and normal termination of *oppose* it is ensured that *intPart* and *decPart* are negated (*ensures* clause). We are able to refer to the values of *intPart* and *decPart* before execution of *oppose* by writing $\backslash\text{old}(\text{intPart})$ and $\backslash\text{old}(\text{decPart})$ respectively. The purpose of the *modifiable* clause is to maintain a list of all fields that can be modified by *oppose*.

In JML we may also write clauses that specify behavior when exceptional or abnormal termination occurs. The examples in this paper do not exhibit such behavior. Therefore, these clauses are omitted.

The class invariant is expressed in JML by an *invariant* clause at line 5. A class invariant should be maintained by every method in that class, which means that after execution of *oppose* the invariant should hold, assuming it holds before execution.

The predicates used in the *invariant*, *requires* and *ensures* clauses resemble Java boolean expressions. This is convenient in that one’s specification is written using basically the same syntax as the source code itself.

Now that we have enough knowledge of JML, let us take a look the *Decimal* class in a broader perspective.

```

public class Decimal extends Object {
2   public static final short PRECISION;
   private short intPart;
4   private short decPart;
   public Decimal add(Decimal) throws DecimalException;
6   public Decimal mul(Decimal) throws DecimalException;
   public Decimal round();
8   private void add(short, short);
   private void mul(short, short);
10  public Decimal oppose();
}

```

Fig. 2. Discussed fields and methods of the Decimal class

2 Decimal Class Specification

The Decimal class source code was handed to us without any formal specification. For methods like *add* or *mul* it is luckily not very hard to guess the intended functionality. In Fig. 2 we list the members of the Decimal class relevant to this paper. The whole class entails 40 members.

The sparse comments in the source code tell us that the decimal part should be between 0 and 999. The precision of decimal numbers is represented by the value of field *PRECISION* which equals 1000. By examining the method bodies of the Decimal class we find that the code contradicts with the informal comments. The value of *decPart* can be less than 0 (think about *oppose* in the previous section). We use this information to extend the invariant from Fig 1 to the one below. Make we make no explicit statements on the bounds of *intPart*.

```

//@ invariant -PRECISION < decPart && decPart < PRECISION
        && PRECISION == 1000;

```

Note that because of the limited precision of the 3 digits decimal part we have to deal with a significant rounding when multiplying, whereas method *add* runs without loss of precision. Rounding highly complicates the specification for multiplication as we shall see in Sect. 5.

In the next section we take a look at the private helper method *add*.

2.1 Specifying Adding Decimals: void add(short,short)

In Fig. 3 we present the specification and code for the *add* method. Note that only the specification is written by us, the code is used as is, complete with French variable names.

The arguments *e* and *f* of *add* are the integer and the decimal part respectively. At line 9 we first add *e* to *intPart*. The following if-then-else block (lines 12 to 19) ensures that *intPart* and *decPart* are both positive or both negative. To

see how this works, consider for example $intPart = 2$ and $decPart = -400$. The if-then-else block ensures that $intPart = 1$ and $decPart = 600$.

At line 22, the value of field f is added to $decPart$, and again we encounter a similar if-then-else block (lines 23 to 44) that does a conversion to ensure $intPart$ and $decPart$ are both positive or both negative. Lines 23 to 44 also guarantee that the decimal part is within range specified by our invariant: between $-PRECISION$ and $PRECISION$. The method body is needlessly complicated and some pieces are redundant. Namely, we can remove the first if-then-else block (lines 12 to 19) and the first part of the second if-then-else block (lines 23 to 32) and still prove the same specification to be correct. On the other hand, division and modulo operations (at lines 39 and 40) are expensive in terms of CPU cycles. Therefore, the implementation of *add* with the if-then-else blocks in place might be more efficient in terms of speed. By explicitly using the information in our invariant, we can even write an *add* without the expensive division and modulo operators, see Sect. 4.2.

Consider the specification for *add*. The *requires* $f < PRECISION \ \&\& \ f > -PRECISION$ says that the decimal part of the value we add satisfies the invariant. The *modifiable* clause (in which we state what variables may be changed by the method) is set to $intPart$ and $decPart$ which makes sense, because even though we modify other variables like *retenue* or *signe*, these variables are local to the method, and therefore do not have to be listed in the *modifiable* clause. Note that the *ensures* clause uses $intPart * PRECISION + decPart$ as a convenient representation of the value of a Decimal object.

Now that we have established a specification that seems correct and informative enough, we try to prove it correct. Before we can plunge into the specifics of a machine-assisted proof we first must know some more of the way Java and JML are represented within the theorem prover PVS. We do this in the next section.

3 Representing JML-Annotated Java in PVS

The LOOP group in Nijmegen has developed a tool that translates Java source code and corresponding JML annotations to a higher order logic like the one used in PVS. In PVS we can then prove the properties of this Java code by stepping through the method using special Hoare-rules. In this section we see the Hoare rule for composition as used in proofs. First we need to know how Java and JML are represented in PVS.

Converting Java to a logic is a non-trivial business. Also, it is not within the scope of this article to explain exactly how this translation takes place (see [3]), but for short we say that a (shallow) semantics of Java is formalized in the higher order logic of PVS. All primitive types, reference types, and constructs such as statements and expressions have their semantic equivalent in the higher order logic. The logic files generated by the LOOP-tool depend on a hand-written Prelude in which these constructs are defined (see [2, 8]). We restrict ourselves

```

/*@ normal_behavior
2  @   requires f < PRECISION && f > -PRECISION;
   @   modifiable intPart, decPart;
4   @   ensures
   @   (\old(intPart) + e)*PRECISION + \old(decPart) + f
6   @   == intPart*PRECISION + decPart;
   @*/
8  private void add(short e, short f){
   intPart += e;
10  //@ assert intPart * PRECISION + decPart
   //@      == (\old(intPart) + e)*PRECISION + \old(decPart);
12  if ( intPart > 0 && decPart < 0 ) {
       intPart--;
14     decPart = decPart + PRECISION;
   }
16  else if ( intPart < 0 && decPart > 0 ){
       intPart++;
18     decPart = decPart - PRECISION;
   }
20  //@ assert intPart * PRECISION + decPart
   //@      == (\old(intPart) + e)*PRECISION + \old(decPart);
22  decPart += f;
   if ( intPart > 0 && decPart < 0 ) {
24     intPart--;
       decPart = decPart + PRECISION;
26  }
   else if ( intPart < 0 && decPart > 0 ){
28     intPart++;
       decPart = decPart - PRECISION;
30  }
   else {
32     short retenue = 0;
34     short signe = 1;
       if ( decPart < 0 ) {
36         signe = -1;
           decPart = -decPart;
38     }
       retenue = decPart / PRECISION;
40     decPart = decPart % PRECISION;
       retenue *= signe;
42     decPart *= signe;
       intPart += retenue;
44  }
   }
}

```

Fig. 3. `add(short, short)` source code and JML specification

```

StatBehavior
((#
  requires := LAMBDA (pre: OM?) :
    DecimalJML_invariant?(pre) AND
    oppose?behavior_requires(pre),
  statement := oppose?body ,
  ensures := LAMBDA (post:OM?) : LAMBDA (ret?oppose: DecimalJML?Type) :
    DecimalJML_invariant?(post),
    oppose?behavior_ensures(ret?oppose)(post) AND
    oppose?behavior_modifiable(post)
#))

```

Fig. 4. Behavior specification for method `oppose()`

to explaining the transformed combination of Java and JML that is of direct concern to the verifier.

For every method a labeled product (i.e. a record) is generated that contains the method body and all specification clauses for that method. We call this labeled product a “behavior specification”. In Fig. 4 we show the (simplified) behavior specification in PVS notation for method `oppose` from Fig. 1. The PVS code (`# 1 := 5 #`) is a record where field 1 has value 5.

The `requires` predicate in Fig. 4 includes not just the precondition given in the code, but also explicitly includes the invariant, which is implicit in all JML preconditions. The `requires` clause is a lambda abstraction of type `OM?` where `OM?` represents the memory state, so that the predicate can be evaluated in the pre- and post-states.

The `ensures` clause has a lambda abstraction of type `DecimalJML?Type` which is the PVS representation of the type of the object returned by `oppose`. The postcondition also contains a predicate `oppose?behavior_modifiable`. This predicate represents the fact that the fields other than the ones mentioned in the JML clause `modifiable` may not be changed during execution of `oppose`. The actual method body is labeled by the `statement` tag, it is not yet unwrapped.

The meaning of the behavior specification is expressed by the PVS statement `StatBehavior` in Fig. 4. The definition of `StatBehavior` (not given here) is that for any initial state `pre` which satisfies the pre-condition `requires`, the postcondition `ensures` holds in state `post` that results from normal execution².

In general, proving a postcondition in the state that results from executing a method cannot be performed in one step, because a method and therefore its resulting state can be complex. Proving a behavior specification can thus best be done by splitting up the method body into smaller parts. This is achieved by making use of Hoare logic. We have implemented and proved correct Hoare rules for all Java language constructs like composition, if-then-else-clauses, while- and

² The labeled tuples used during the actual verification have four more labels (for exceptional behavior, return, break and continue), but these need not concern us here. See [9] for further details.

```

composition : LEMMA
  (EXISTS(P1 : [OM? -> bool]) :
    StatBehavior((# requires := P,
                  statement := s1,
                  ensures := P1 #))

    AND

    StatBehavior((# requires := P1,
                  statement := s2,
                  ensures := Q #))

  IMPLIES
  StatBehavior((# requires := P,
                statement := s1 ; s2,
                ensures := Q #))

```

Fig. 5. Hoare rule for composition

for-loops and try-catch-statements (see [9]). These Hoare rules are written in terms of behavior specifications as shown above. We discuss one such a Hoare rule which is the rule for composition. The standard rule for composition, written here as “;”, is given in natural deduction style below.

$$\frac{[P] \ s1 \ [P1] \ , \ [P1] \ s2 \ [Q]}{[P] \ s1 ; s2 \ [Q]} \textit{composition} \quad (1)$$

Our rule for composition is shown in PVS style in Fig. 5. Apart from the style, rule (1) is comparable to the one in Fig. 5.

Apart from Hoare rules, we also have Weakest Precondition (WP) rules at our disposal. Using these we can verify code fragments that contain no repetition automatically. Details of this WP calculus will appear in a separate publication.

Now that we know some more about Java annotated with JML and its meaning in the higher order logic of PVS, let us do some actual verification.

4 Decimal Class Verification

In this section we present a proof for the private *add*. The typical way in which we prove the correctness of methods in PVS is that we use the composition rule to chop up our proof obligation in several pieces. We then apply the WP-tactic to prove these pieces automatically. As we have just seen, the LOOP-tool converts easily understandable Java code and readable specifications to obscure PVS code, which is unpleasant for most people. The intermediate predicate for the composition rule needs to be entered in PVS by the verifier. We can also write intermediate predicates in Java source code by using a yet unmentioned feature of JML: the *assert* clause. The *assert* clause is not yet interpreted by the LOOP-tool such that it can instantiate the intermediate predicates for composition automatically. This will be possible in the near future, but for now we use the *asserts* to indicate where we cut up the method body.

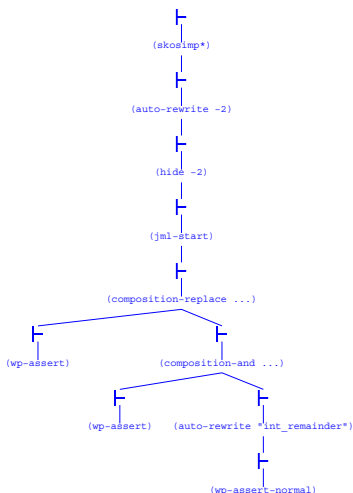


Fig. 6. Proof-tree for `add(short, short)`

Although PVS code is not quite suited to be presented here, the PVS proof-tree is. A proof-tree is generated during the process of proving and graphically presents the verifier with an overview of the chosen steps (applications of Hoare rules for example) and amount of subgoals still left to prove. The proof-tree for method `add` is shown in Fig. 6. Every node in the proof-tree represents a tactic applied by the verifier.

4.1 Verifying void `add(short, short)`

Because `add` contains no repetitions, we should be able to apply the WP-tactic to finish the proof. Unfortunately, if do not chop up the method body into smaller pieces, the proof obligation is too large for PVS to handle.

Therefore, the proof consists of two applications of the Hoare rule for composition. In Fig. 6 these applications are labeled “composition-replace” and “composition-and”. Applying composition twice results in three chunks of code, namely line 9, lines 12 to 19 and lines 19 to 44. These chunks are verified using WP-tactic. In the proof-tree in Fig. 6 this is denoted by a label “wp-assert”.

The two intermediate predicates used to apply the rules for composition are shown at lines 10 and 20 in Fig. 3 marking the position where the code is cut. Note that these assertions are almost identical to the ensures clause.

4.2 Optimized Addition

After taking a closer look at the `add` code we realized that the real issue is (possibly) restoring the invariant after the assignment `decPart+ = f`. Once this assignment has occurred, we still know that $-2 * PRECISION < decPart \ \&\& \ decPart <$

$2 * PRECISION$. Hence the invariant can be restored by adding or subtracting $PRECISION$ at most once. Therefore, addition can also be performed in the following way.

```

private void addOpt(short e, short f){
2   intPart += e;
   decPart += f;
4   if ( decPart <= -PRECISION ) {
       decPart += PRECISION;
6       intPart--;
   }
8   else if ( decPart >= PRECISION ) {
       decPart -= PRECISION;
10      intPart++;
   }
12 }

```

This short implementation satisfies³ the same specification as the longer variant in Fig. 3. It avoids the costly division and remainder operations of the original implementation, and is thus much better suited for implementation on a smart card. This optimized implementation can only be written after the class invariant has been made explicit. Hence specification can have an impact on code development.

5 Decimal Multiplication

In this section we specify the private method *mul* for multiplication. It has two arguments of type *short*, similar to the private *add* method. The *mul* method is probably the most difficult one of the whole *Decimal* class, both in terms of specification and of implementation.

We first consider its specification, in Fig. 7. Like in the specification of addition in Fig. 3, we use a combined formulation for *intPart* and *decPart*. The arguments are *e* and *f* for the *intPart* and *decPart* of the multiplier. An easy calculation shows that we can write the combined multiplication as follows.

$$\begin{aligned}
 & \frac{(\backslash\text{old}(\text{intPart}) * \text{PRECISION} + \backslash\text{old}(\text{decPart})) * (\text{e} * \text{PRECISION} + \text{f})}{\text{PRECISION}} \\
 &= \backslash\text{old}(\text{intPart}) * \text{e} * \text{PRECISION} \\
 & \quad + \backslash\text{old}(\text{intPart}) * \text{f} \\
 & \quad + \backslash\text{old}(\text{decPart}) * \text{e} \\
 & \quad + \text{X}
 \end{aligned} \tag{2}$$

where

$$\text{X} = \text{sgn}(\text{f} * \backslash\text{old}(\text{decPart})) * \frac{\text{trunc}(\text{abs}(\text{f})) * \text{trunc}(\text{abs}(\backslash\text{old}(\text{decPart})))}{1000} \tag{3}$$

³ To our embarrassment, verification showed an implementation error in our first version of *addOpt*, in which we used $<$ and $>$ instead of $<=$ and $>=$.

```

/*@ normal_behavior
2  @   requires -PRECISION < f && f < PRECISION;
   @   modifiable intPart, decPart;
4  @   ensures intPart * PRECISION + decPart ==
   @       \old(intPart) * e * PRECISION
6  @       + \old(intPart) * f
   @       + \old(decPart) * e
8  @       + ( ((f >= 0 && \old(decPart) >= 0) ||
   @           (f < 0 && \old(decPart) < 0))
10 @           ? 1
   @           : -1)
12 @       * (( ((-100 <= f && f <= 100)
   @           ? abs(f)
14 @           : 10 * (abs(f) / 10))
   @           *
16 @           ( (-100 <= \old(decPart) &&
   @               100 >= \old(decPart))
18 @               ? abs(\old(decPart))
   @               : 10 * (abs(\old(decPart))/10))
20 @           ) / 1000);
   @*/
22 private void mul(short e, short f){ ... // see Fig. 4 }

```

Fig. 7. JML specification of the private multiplication method

The difficulty lies in part **X**. In the Java implementation it involves a certain truncation, which is rather complicated to describe in JML: it covers the greater part of Fig. 7, from line 8 onwards. We shall try to explain what is going on, using the standard mathematical operations `abs` and `sgn`, where `sgn` is currently not available in JML

$$\text{abs}(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases} \quad \text{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Next we should keep in mind that division in Java (and JML) involves truncation. For instance, $10 * (347/10) = 340$ and $10 * (-347/10) = -340$. So we can define an auxiliary function:

$$\text{trunc}(x) = \begin{cases} 10 * (x/10) & \text{if } x \geq 100 \\ x & \text{otherwise} \end{cases}$$

The combined specification (2) and (3) can then be proved for the `mul` implementation in Fig. 4. We shall briefly sketch the proof, explaining some of the more special points. The method body in Fig. 4 can be split into two parts:

- (a) lines 2–24, establishing the equation (2) without **X**;
- (b) lines 25–51, establishing the equation (2) including **X**;

```

void mul(short e, short f){
2  short intBackup = intPart;
  short decBackup = decPart;
4  short nbIter = e;
  if ( nbIter < 0 )
6    nbIter = -nbIter;
  intPart = 0;
  decPart = 0;
8  for(short i=0;i<nbIter;i++)
    add(intBackup, decBackup);
10  if ( e < 0 )
12    oppose();
  short intPart_ = intPart;
14  short decPart_ = decPart;
  intPart = 0;
16  decPart = 0;
  nbIter = intBackup;
18  if ( nbIter < 0 )
    nbIter = -nbIter;
20  for(short i=0;i<nbIter;i++)
    add(0, f);
22  if (intBackup < 0 )
    oppose();
24  add(intPart_, decPart_);
  short signe = 1;
26  short arrondis1 = decBackup;

  if ( arrondis1 < 0 ) {
28    arrondis1 = -arrondis1;
    signe = -signe;
30  }
  short arrondis2 = f;
32  if ( arrondis2 < 0 ) {
    arrondis2 = -arrondis2;
34    signe = -signe;
  }
36  short decal = 0;
  while ( arrondis1 > 100 ) {
38    arrondis1 /= 10; decal++;
  }
40  while ( arrondis2 > 100 ) {
    arrondis2 /= 10; decal++;
42  }
  short temp = arrondis1
44    * arrondis2;
  short aux = 1000;
46  while ( decal > 0 ) {
    aux /= 10; decal--;
48  }
  temp /= aux;
50  temp *= signe;
  add(0, temp);l }

```

Fig. 8. Implementation of the private multiplication method

Part (a) uses two `for` loops, firstly for `abs(e)`-times adding `\old(intPart)` and `\old(decPart)` in lines 9 and 10, and secondly for `\old(intPart)` adding only `f` in lines 20 and 21. These two steps are combined in line 24, via the intermediate values `intPart_` and `decPart_`. Part (b) starts by setting `signe = sgn(f * \old(decPart))` which is the first part of (3), in two `if` statements. At the same time, `arrondis1 = abs(\old(decPart))` and `arrondis2 = abs(f)`. The two subsequent `while` loops handle the truncations. It follows from our invariant and precondition that both these `while` loops are executed at most once. The explicit use of the numbers 100 and 1000 is a bit clumsy here. It would be more abstract to use `PRECISION/10` and `PRECISION` instead⁴. The third `while` loop determines the appropriate compensation factor after truncation. This loop is executed zero, one or two times. Finally, `temp` becomes our `X` equation (3), which is added to the final result.

⁴ We have decided to follow this clumsiness in our JML specification in Fig. 7.

6 Conclusions

This paper describes a case study in JML specification and PVS verification for JavaCard, in the style of [4]. It forms an incremental improvement on [4], by tackling a Java class with complicated computational content and non-trivial functional specifications. It demonstrates that the “LOOP” verification technology extends to such examples. What else can be learned from this verification? We have several points.

- It cannot be over-emphasized that formalizing class invariants is very important, because it makes implicit assumptions explicit. In the *Decimal* class the Gemplus developers have (informally) described an invariant $0 \leq \text{decPart} < \text{PRECISION}$, but this one is broken by the (public) method *oppose*. Hence we come to a weaker invariant $-\text{PRECISION} < \text{decPart} < \text{PRECISION}$. More generally, formal specifications provide unambiguous documentation (as long as the semantics is clear, of course).
- Functional specifications can both be compact and readable (like for the *add* method in Fig. 3) and verbose and unreadable (like for *mul* in Fig. 7). Still, the specification adds useful information which is not readily available from inspecting the code. Actual verification may then be justified in critical applications.
- Code specification and verification are iterative processes. Most often (in our experience) one stops a particular verification at some stage to alter the specification—and restart the verification. But also, one may wish to alter the code itself. We have chosen not to do so in this case study; several others, like [5], also use the *Decimal* class as case study, so changing the code leads to confusion. But there were several occasions where we would have liked to alter the code.

In the end there is always the question: how much time did this take? Unfortunately, we cannot give an exact answer, because: this case study has been used by one of the authors (CBB) to learn the verification technology. Also, it has been used to improve our proof technology, especially by writing many dedicated proof strategies in PVS (especially by JvdB). And finally, halfway the verification we decided to switch to the next beta release of PVS, in order to maximize our influence on the development of this new version. All these factors caused considerable delays, which would not be there if this case study had been done with all parameters optimally tuned. But probably this never happens in practice. But if we have to make an estimate: a method like *mul* (with a specification) would take a day to verify.

References

- [1] Gemplus Purse applet.
http://www.gemplus.com/smart/r_d/publications/case-study/index.html.
 305

- [2] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000. 309
- [3] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001. 305, 309
- [4] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard’s Application Identifier Class. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of LNCS, pages 137–150. Springer Verlag, 2001. 317
- [5] N. Cataño Collazo. La clause modifiable de JML: Sémantique, vérification et application. Master’s thesis, INRIA de Sophia-Antipolis, 2001. 317
- [6] N. Cataño and Marieke Huisman. Formal specification of Gemplus’s electronic purse case study. In *Formal Methods Europe (FME 2002)*, Lect. Notes Comp. Sci. Springer, 2002. To appear. An earlier version was presented at Verificard’02, first annual meeting of the VerifiCard project, Marseille, January 2002. 306
- [7] N. Cataño Collazos and M. Huisman. Comments on the gemplus purse applet. http://www-sop.inria.fr/lemme/verificard/electronic_purse/. 306
- [8] M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 2001. 309
- [9] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2001. 306, 311, 312
- [10] G. T. Leavens, Albert L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See www.cs.iastate.edu/~leavens/JML.html. 305
- [11] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997. 304
- [12] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of Lect. Notes Comp. Sci., pages 63–77. Springer, Berlin, 2000. 306
- [13] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Techn. Univ. München, 2000. 305
- [14] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lect. Notes Comp. Sci., pages 119–156. Springer, Berlin, 1998. 305
- [15] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996. 305
- [16] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000. 306
- [17] Extended static checker ESC/Java. Compaq System Research Center. www.research.compaq.com/SRC/esc/Esc.html. 306

A Method for Secure Smartcard Applications

Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany
{haneberg,reif,stenzel}@informatik.uni-augsburg.de

1 Introduction

Smartcards as tamper-proof, secure devices have a high potential in e-commerce applications. The technology is available to have different, highly critical applications on one open multi-applicative card. However, they are not yet used in real life because of the high risks involved, and the legitimate security concerns: it is very difficult to design secure systems [2]. We show how formal methods can help to improve the situation.

We propose a methodology using specific techniques to develop secure smartcard applications: the application and the communication protocols are modelled with UML and algebraic specifications. The security against an attacker is proved after transformation of the UML parts into algebraic specifications using the KIV verification system¹ [14][3]. The correctness of the smartcard program – written in JavaCard – is formally verified with KIV against the same specification, thereby closing the gap between abstract protocol verification and JavaCard programming.

The paper is organized as follows: Sect. 2 introduces a small, but surprisingly interesting example. Section 3 presents the method in a nutshell. Section 4 describes the modelling technique, Sect. 5 the translation into algebraic specifications, and Sect. 6 deals with the smartcard programs. Section 7 presents related work, and Sect. 8 wraps up.

2 A Small Example

As a simple example for a smartcard application we describe an open copy card application for a library or university. We need three components:

- a smartcard that holds ‘value points’;
- a ‘filling station’, i.e. a machine that accepts money and loads value points onto the card;
- a ‘pay station’, i.e. a card reader connected to a copier that subtracts value points before printing a copy.

¹ <http://www.informatik.uni-augsburg.de/swt/fmg/>

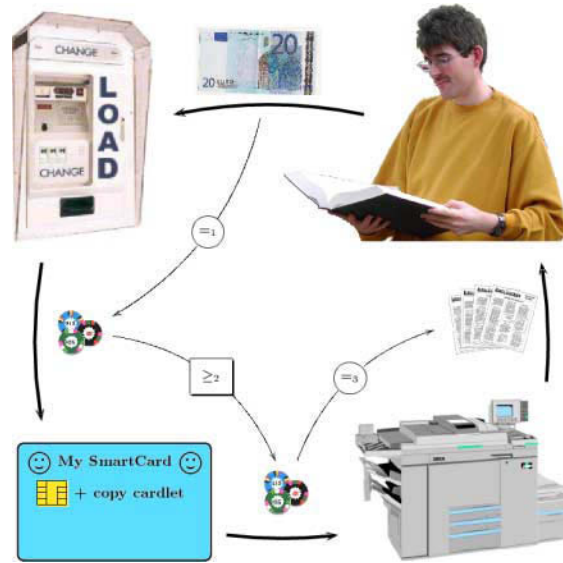


Fig. 1. Example scenario

Of course it is possible to use this application in other scenarios where ‘value points’ are an adequate business model, and of course we usually have several filling stations, several pay stations and many smartcards.

What is the advantage over existing copy cards? A user of the service can use any open, multi-applicative Java smartcard (that may contain already several different applications), and just download an additional *cardlet* (a program) for this service. Thus it is not necessary to carry a dozen different cards around, but ideally only one. On the other hand, the application provider saves money for throw-away magnetic stripe cards and reduces problematic waste.

Usage of the card is simple and illustrated in Fig. 1: the card holder inserts the card into the filling station, sees the number of value points left on the card, selects the number of points to load, and inserts the necessary money into the machine. Then the new points are loaded (added) onto the cardlet. To copy with the card, it is simply inserted into the card reader of the copier. The reader displays the remaining number of points, and the copier asks the cardlet to pay (subtract) one or more points before printing a copy.

What can go wrong in this scenario?

1. The filling station should not ‘swallow’ money, i.e. the value points loaded onto the card should equal the money inserted into the machine. On the other hand, the station should not load too many points ($=_1$ in Fig. 1).
2. Adding and subtracting points should work correctly even for limited resources (smartcard programming languages often have no integers but only bytes and shorts).

3. The number of points paid by a cardlet should always be less or equal to the number of loaded points (\geq_2).
4. The printed copies should equal the value of the paid points ($=_3$).
5. From the point of view of the application provider: the overall number of copies printed should not be worth more than the sum of money inserted into all filling stations ($=_1 + \geq_2 + =_3$).

We concentrate on those issues related to the smartcard. This means that item 5 can be reformulated as

The sum of all points issued at all filling stations should be larger or equal to the sum of all collected points at all pay stations.

This is the main security property for this application. We ensure this property and items 2 and 3 by applying formal methods. Item 2 is a matter of correct programming, but item 3 and the security property require some cryptography. The protocols for the communication between the filling and pay station (we will use the generic term *terminal* in the sequel) and the cardlet must be designed in a manner that ensures these goals.

Why is this not trivial? Because the card holder is often considered as hostile in smartcard scenarios. A person trying to cheat (an *attacker*) could try to

1. load ‘forged’ points onto a proper cardlet with a home PC and a card reader;
2. pay with ‘forged’ points using another, differently programmed smartcard;
3. trick the terminal or the cardlet into revealing secret keys etc. using forged cards, or technical equipment such as a second chip or additional circuitry implanted in the card.

It depends mainly on the abilities of the cardlet and the terminal what attacks can be averted. In this example we assume that the cardlet can encrypt and decrypt with a symmetric key, but cannot do asymmetric cryptography or generate random numbers (challenges, *nonces*).

A terminal and a cardlet work in master/slave mode. The terminal sends a command to the cardlet, the cardlet receives and processes the data, and returns an answer. The cardlet never initiates a communication or sends data on its own. So let us consider the following communication protocols for the example:

the load protocol: The filling station sends a nonce (a random number) to the cardlet. The cardlet encrypts the nonce with a secret key and returns the result. The terminal decrypts the answer (it must know the same secret key) and checks if it is indeed the nonce. This authenticates the cardlet, i.e. the terminal believes the cardlet to be genuine. (Only a genuine cardlet should know the secret key, and a replay attack is not possible because the nonce has never been used before.) In this example all cardlets and terminals use the same secret key, so key distribution is not a problem.

Then the terminal sends the number of points to load as a simple integer together with a secret (an arbitrary, but fixed value known to the terminal and the cardlet). The cardlet checks the secret and increases the number of points. All cardlets and terminals share the same secret.

The secret is used to authenticate the terminal. However, this is a very weak form of authentication due to the limited abilities of the cardlet. If an attacker can eavesdrop on the communication the secret is revealed. So this protocol is not secure against an attack with technical equipment.

But at least it should be secure against forged cardlets. We justify the above claim informally. If the first step (sending a nonce) is omitted an attacker can insert a card with a cardlet that simply stores the data it receives, thereby revealing the secret. It is necessary to return the encrypted nonce (otherwise the forged cardlet can return the nonce and receive the secret in the next step), and encrypting the secret does not help (the forged cardlet stores the encrypted secret which can later be sent to the genuine cardlet).

the pay protocol: The terminal sends the number of points to pay together with a nonce. The cardlet checks that enough points are available, adjusts the amount, and returns the encrypted nonce. The terminal checks the nonce and accepts the payment. Without this check an attacker could use a forged cardlet that simply answers *ok*. The authenticity of the terminal is not verified. The card owner is responsible for ensuring that he uses his card only in genuine terminals.

It turns out that the two protocols are quite different although they just add and subtract a value, and they are (perhaps) more complicated than expected. It must be emphasized that they are designed with an attacker in mind that has some specific capabilities (using forged cardlets), but not others (eavesdropping on or manipulating the communication). Such attackers require other protocols, but for our example they are sufficient.

3 The Method in a Nutshell

Our focus is on the smartcard in the complete scenario, i.e. the programs on the card and the communication with the terminals. We propose to take the following steps using particular techniques for developing secure smartcard applications:

1. Modelling and formalization

- (a) Model the relevant parts of the scenario with UML class diagrams augmented by algebraic specifications. This includes the cardlet, the terminals, and their data.
- (b) Formulate the security properties as class invariants and/or constraints.
- (c) Define the capabilities of an attacker as an algebraic specification.
- (d) Design the communication protocols with UML activity diagrams.

2. Proving security

- (e) The UML model is transformed into an algebraic specification (together with the attacker capabilities); the security properties become proof obligations.
- (f) Prove the security properties.

3. Refinement and verification

- (g) Refine the abstract data types used in the formal model of the scenario to real JavaCard data types (byte arrays etc.).
 - (h) Implement the card program.
 - (i) Proof obligations are generated from the protocol axioms and the refinement for the correct behaviour of the program.
 - (j) Prove the correctness of the refinement and the implementation.
4. **Allowing multiple applications**

A mandatory security policy for the smartcard allows to employ open multi-applicative cards. The application provider must be sure that nobody – not even the card holder – can manipulate or read the card program because it typically contains secret keys. This problem is solved in [15], and not considered here.

It can be seen that smartcard applications (even though small in terms of software size) combine several different types of problems, and require therefore a combination of different formal methods: smartcards and terminals are inherently object oriented, and the complete software system is highly distributed. There is an enormous gap between an abstract model of data types and their implementation on the card (*documents* vs. byte arrays) due to the limited resources of the card. Reasoning about an attacker and security requires a logical, qualitative definition of cryptographic functions. A successful formal treatment of these problems requires the combination of those techniques that are most effective for every problem into a homogeneous solution that leaves no gaps between different views on the system. In this sense the steps (c), (d), (g), (i), and 4. are very specific for smartcard applications. The next three sections describe the first three steps of our approach in more detail.

4 Specification of the Example

The objects of a smartcard scenario are modeled in a UML class diagram [17] with algebraic annotations, the protocols are written down as UML activity diagrams².

4.1 Modelling Objects and Data

In the example we concentrate on the smartcard part of the application. Therefore we only deal with the terminals, the cardlet and the communication between them. The class diagram for the example can be seen in Fig. 2. It is divided into three parts. First, we have the classes describing the data (region with light gray background in Fig. 2). Secondly, the classes describing the software components (white background) and thirdly, an infrastructure of classes needed to formalize the attacker and the security properties (dark gray background). To simplify the specification and because it does not affect the main characteristics of the

² Diagrams for all protocols and the complete specification are available on our website. (See section *publications* on <http://www.informatik.uni-augsburg.de/swt/fmg/>)

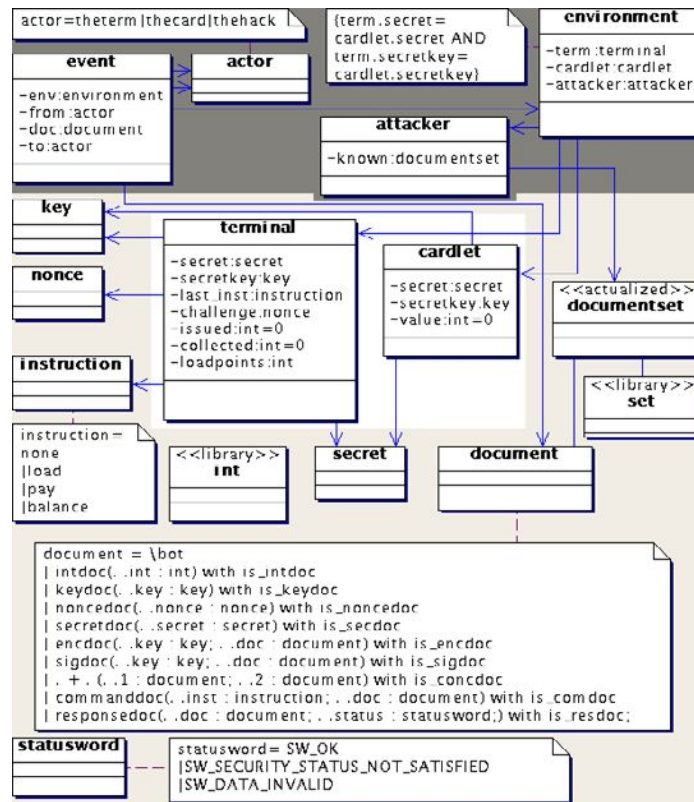


Fig. 2. The class diagram for the example application

scenario we use only one terminal (filling and pay station in one terminal instead of two).

The class **cardlet** describes the card program. It has three attributes, **value**, to store the number of value points currently loaded, **secret**, to store the secret used to authenticate the terminal and **secretkey**, to store the key for the symmetric encryption. The class **terminal** describes what attributes are necessary to store the internal state of the terminal. The **document** class describes what data can be exchanged between the terminal and the cardlet. The **document** class represents an abstract model of the possible documents. The other classes in the light gray region of the class diagram represent the basic entities of which documents can be made up. The class **actor** determines what participants are considered in the scenario. The class **environment** contains all active elements of the scenario, in this case the terminal, the cardlet and the attacker. The class **attacker** describes the knowledge of the attacker. Reasoning about the protocols is done by analyzing traces – in our case finite lists – of events (see e.g. [12]). Each event corresponds to one data exchange. The class **event** describes these data

exchanges. An event consists of the environment in which the communication took place, the sender of the data, the sent document and the receiver.

As can be seen in Fig. 2, we do not use pure UML class diagrams. Instead we enrich the class diagrams so that we are able to express certain constructs of structured algebraic specifications, e.g. a free datatype declaration as in CASL [6]. If the data type has only one constructor (a record type) it is written down as a normal class with its fields turned into attributes of the class. For example the class **cardlet** is a record with the fields **value**, **secret** and **secretkey**. If several different constructors are needed, we put the algebraic specification of the data type in a note and link the note to the class. This was done for the class **document**.

Other important structuring operations are the import of specifications from a library and the usage of generic data types. To import a specification from a library we add a class to the class diagram that is named like the library specification and given the stereotype \llbracket library \rrbracket . To actualize a generic specification we put a class in the class diagram with the stereotype \llbracket actualized \rrbracket . Additional information (mapping of symbols and the actual specification) is put into the class description.

These additions to the class diagram allow to view the class diagram as a graphical representation of a structured algebraic specification.

4.2 Security Properties and the Attacker

The security properties are typically added as constraints to the classes in the class diagram. Sometimes a security property can be expressed in an easier way if certain additional attributes (that are not necessary for an implementation), are added to a class. Because of the simple structure of the example it is possible to formulate the security property as a constraint of the class **terminal**. The property is:

$$\text{local-sec: collected} \leq \text{issued}$$

collected counts the number of points that were paid by cards, **issued** counts how much points have been loaded onto cards. The attributes **collected** and **issued** have been introduced to express the security property in a very simple and intuitive manner.

The axioms describing the attacker consist of two parts and these parts are located in different specifications. The first part deals with the knowledge of the attacker. It describes how the attacker can decompose documents into their parts and build up new documents from the ones he already possesses. The decomposition of documents is done by splitting composed documents into their parts, decrypting encrypted documents if the attacker has the key, and so on. During this analysis of a document the attacker adds all parts of the original document to his knowledge and can thereby acquire new keys or nonces that were contained in the analyzed document. This is similar to [12]. The corresponding axioms are located in **attacker**. The second part describes which data

transfers the attacker can observe or modify. This is formalized as a condition for admissible traces. One axiom is

$$\begin{aligned} & \text{admissible}(\text{tr} + \text{ev} + \text{ev}0) \\ \leftrightarrow & \text{admissible}(\text{tr} + \text{ev}) \wedge \text{stepadm}(\text{tr} + \text{ev}, \text{ev}0) \\ & \wedge (\text{ev.to} = \text{thehack} \rightarrow \text{ev}0.\text{known} = \text{add_known}(\text{ev.doc}, \text{ev.known})) \\ & \wedge (\text{ev.to} \neq \text{thehack} \rightarrow \text{ev}0.\text{known} = \text{ev.known}) \end{aligned}$$

It states that a trace ($\text{tr} + \text{ev} + \text{ev}0$) with at least two elements (traces with less than two events are always admissible) is admissible iff the trace without the last event ($\text{tr} + \text{ev}$) is admissible, if $\text{ev}0$ is an admissible next step for the trace $\text{tr} + \text{ev}$ and if the knowledge of the attacker was properly treated: if the receiver of the data transfer in event ev was the attacker, the sent document (ev.doc) is added to the known documents of the attacker, otherwise the knowledge remains unchanged. This models an attacker that cannot eavesdrop on the communication between a terminal and a cardlet.

In our opinion there is a limited number of typical attackers. They can be specified once and for all and stored in a library.

4.3 Specification of the Protocols

There are two protocols in our example, the *load* protocol and the *pay* protocol (Fig. 3).

We use UML activity diagrams for the specification. Each actor of a protocol (mostly one terminal and one cardlet) is represented by a swim lane. A swim lane is a partition of activity diagrams to express responsibilities. We use this to state who executes which operations and performs which tests. The flow of a protocol is from the start node (① in Fig. 3) to the end node (⑧). Activity nodes (②) describe state changes of the cardlet or the terminal (only activities labeled Process or Receive have a special function), objects (③) represent transferred data. The structure of the data is defined in a note (④) attached to the object. Branches (⑤) (or decision nodes) are used to state that a condition must be satisfied for a path to be executable. Signal sending nodes (⑥) model exceptions. Signal receipt nodes (⑦) model the catcher for these exceptions. Following an activity diagram from the start to the end node presents the intended flow of the protocol, the faultless completion of the protocol. Error handling is treated by the send and receipt nodes. The communication steps can be identified by looking at the border of the swim lanes. Each object flow link crossing a border represents a data exchange.

The *pay* protocol (Fig. 3) is quite short. The protocol starts with an activity of the terminal in which the attribute `last_inst` is set to the value `pay` and a new challenge is generated. The newly generated nonce is stored in the attribute `challenge`. Then a command must be sent to the cardlet. The command (object `col`) consists of the card instruction `pay` and a compound document consisting of the number of points that are to be paid (`loadpoints`) and the nonce that has just been created (`challenge`). The cardlet receives this command and calls the process method.

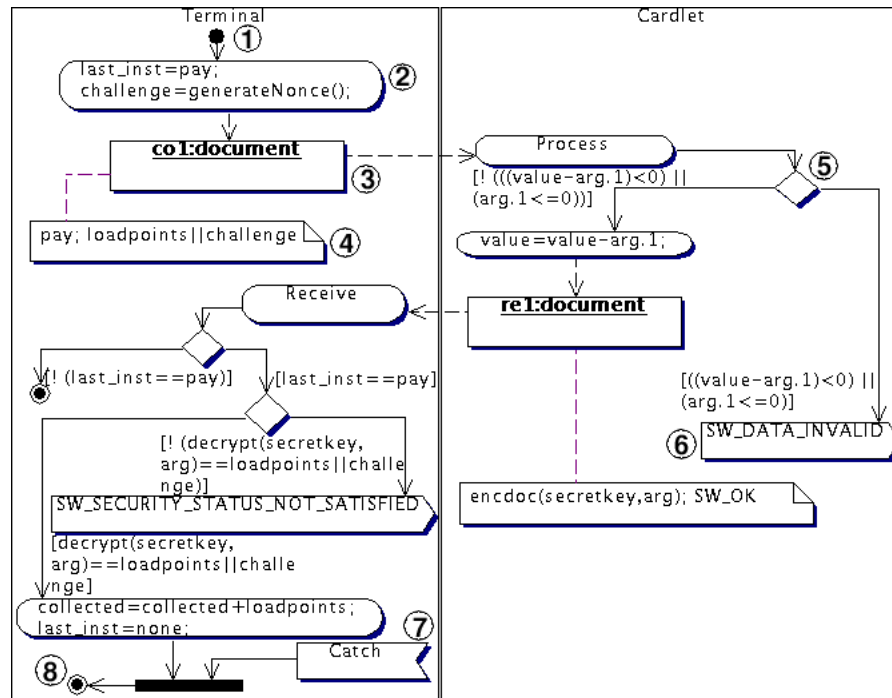


Fig. 3. Activity Diagram of the *Pay* Protocol

If there are not enough points left to pay ($value - arg.1 < 0$) or if the argument is not greater than 0 ($arg.1 \leq 0$) the cardlet will not proceed with the protocol and send the status word `SW_DATA_INVALID` to the terminal. If the test succeeds the value in the cardlet is decremented ($value = value - arg.1$) and a response is sent to the terminal. The response consists of the complete argument sent to the cardlet encrypted with `secretkey`, and `SW_OK` as status word. After receiving the response, the terminal checks two conditions. Most important, the terminal must verify that the decryption of the received document leads to the data sent to the cardlet earlier. Only if the verification succeeds, the terminal is convinced that the value in the cardlet was actually decremented and that the cardlet is authentic (because only authentic cardlets know the secret key). If this was successful the attribute `collected` in the terminal is incremented by `loadpoints` and the last instruction field is set to `none`. The signal receipt node leads directly to the stop node. Any status word other than `SW_OK` terminates the protocol.

The *load* protocol is a little bit longer and omitted here. Compared to the description in Sect. 2 the full protocol is much more detailed. Especially the error handling is fully elaborated, e.g. the error messages are fixed, and the number of value points is limited because of the resource restriction of real smartcards.

5 From Diagrams to Algebraic Specifications

5.1 Class Diagram

The classes are automatically transformed into specifications. For example the class `cardlet` is transformed into the following free data type specification (CASL) in the KIV system:

```
cardlet = mkcardlet( . .secret : secret; . .secretkey : key; . .value : int ) ;
```

The data type is named `cardlet`, its only constructor is `mkcardlet()` and it has 3 parameters. The names of the attributes of the class are used for the names of the selector functions (a leading dot is added to the name). For example, the attribute `secret` leads to a selector function `.secret : cardlet → secret`. The axioms for a free data type are generated automatically consistent and unique up to isomorphism.

The local security property from the `terminal` class is automatically transformed into:

```
sec_glob:  init(tr) ∧ admissible(tr)
           → ∀n. tr[n].env.term.collected ≤ tr[n].env.term.issued
```

The formula states that the local constraint must be true in all states of the terminal in all admissible traces. Herein `tr[n].env.term.collected` selects the collected attribute from the terminal in the environment of the n -th step of the trace. The local property is thereby lifted on a global prospect. Properties that cannot be formulated as invariants of an object can be expressed directly on the level of traces.

5.2 Activity Diagram

The result of the transformation of an activity diagram is a set of axioms describing two functions, `send()` and `receive()`. The function `send()` describes what happens when a command is sent to the cardlet, `receive()` deals with the terminal receiving a response. Commands and responses have two different effects. First, the internal state of the terminal or the cardlet is changed and second, a new command or response is generated. Therefore they return pairs, `send()` a pair of cardlet and document and `receive()` a pair of terminal and document:

```
send : cardlet × document → cardlet×document
send : terminal × document → cardlet×document
```

The axioms are generated by following the flow of the protocol and collecting the required information. The informations needed are the last command or response that was sent, the guard conditions after branches, the changes to the internal state and the structure of the next response or command. Let us look at some steps of the `pay` protocol (Fig. 3). The first activity sets `last_inst` to `pay` and

puts a newly generated nonce in **challenge**. These modifications to the terminal are stored by the generation process. The next node is an object representing a command that is to be sent to the cardlet. The attached note shows that the instruction in the command is **pay** and that **loadpoints** and **challenge** are to be sent as additional data. As we leave the terminal swim lane now, all information for the first axiom is gathered and it can be created. The modification of the state of the terminal is written down as a Java statement (e.g. **last_inst=pay**). Such assignments are translated into calls of special modification functions that take a terminal or cardlet and modify exactly one field. The assignment stated above is translated into $\text{term}_1 = \text{set_last_inst}(\text{term}, \text{pay})$. Combining all modifications and the next command we get the following axiom:

$$\begin{aligned} & \text{term}_1 = \text{set_last_inst}(\text{term}, \text{pay}) \\ & \wedge \text{term}_2 = \text{set_challenge}(\text{term}_1, \text{nonce}) \\ & \wedge \text{term}_3 = \text{set_loadpoints}(\text{term}_2, \text{get_int}(\text{doc})) \\ \rightarrow & \text{receive}(\text{term}, \text{commanddoc}(\text{pay}, \text{doc})) = \\ & \text{term}_3 \times \text{commanddoc}(\text{pay}, \text{intdoc}(\text{term}_3.\text{loadpoints}) + \\ & \quad \text{noncedoc}(\text{term}_3.\text{challenge})) \end{aligned}$$

In general the `receive()` function will get a `responseudoc` as second parameter. As this is the first step of the protocol, there is no last response from the cardlet. The start of a protocol must be triggered by an external event, in most cases an action of the card holder. This is modeled by sending a `commanddoc` to the terminal, telling the terminal which protocol is to be executed. In this case, the terminal executes the *pay* protocol and the number of points given in `doc` is to be paid.

The axiom above has no preconditions originating from decisions in the protocol. The equations on the left hand side are produced by the modifications to the terminal. Now we show another axiom for *pay*, this time an axiom for the cardlet, i.e. about `send()`. It must be tested that the number of points to pay is nonnegative and that there are enough points left to pay ($\neg (\text{arg.1} \leq 0 \vee \text{value} - \text{arg.1} < 0)$). If the condition is satisfied, the protocol demands that `value` is decremented by `arg.1` and that the cardlet creates a response indicating the correct completion of the pay operation. Taking the entry of the activity and the note defining the structure of the response, we get:

$$\begin{aligned} \text{pay_cardlet.1:} \quad & \neg (\text{cl}.\text{value} - \text{get_int}(\text{getparam}(1, \text{doc})) < 0 \vee \\ & \quad \text{get_int}(\text{getparam}(1, \text{doc})) \leq 0) \\ & \wedge \text{cl}_1 = \text{set_value}(\text{cl}, \text{cl}.\text{value} - \text{get_int}(\text{getparam}(1, \text{doc}))) \\ \rightarrow & \text{send}(\text{cl}, \text{commanddoc}(\text{pay}, \text{doc})) = \\ & \text{cl}_1 \times \text{responseudoc}(\text{encdoc}(\text{cl}_1.\text{secretkey}, \text{doc}), \text{SW_OK}) \end{aligned}$$

In the axiom `cl` and `cl1` are variables for cardlets. The error handling in this case is simple. If the condition is not satisfied, the cardlet has to return `SW_DATA_INVALID` and leave its internal state unchanged. The corresponding axiom is:


```

pay_cardlet_2:   cl.value - get_int(getparam(1, doc)) < 0 ∨
                get_int(getparam(1, doc)) ≤ 0
                → send(cl, commanddoc(pay, doc)) =
                   cl × responsedoc(⊥, SW_DATA_INVALID)

```

Aside from the nodes in the direct flow of the protocol we have a signal receipt node. It is labeled ‘Catch’ and catches all error signals. The catcher does not lead to any actions but instead leads to the stop node. This means that the function `receive()` produces an unchanged terminal as result and \perp as next command. The axiom is as follows:

```

is_resdoc(doc) ∧ doc .status ≠ SW_OK ∧ term .last_inst = pay
→ receive(term, doc) = term × ⊥

```

Altogether we get 17 axioms for the *pay* and the *load* protocol: 6 for *pay*, 9 for *load* and two for error handling.

The result of the transformation is an algebraic specification for the smart-card application. It consists of the data types, the protocol axioms and the attacker capabilities.

$$\mathbf{SPEC} = \text{classes} + \text{protocols} + \text{attacker}$$

The correctness of the security property (`sec_glob`) can now be established by proving that

$$\mathbf{SPEC} \models \text{sec_glob}$$

It may seem strange to split a complete protocol into several axioms. But the reason is simple. It corresponds to the way the cardlet works. The cardlet receives one command and returns a response. It is crucial to keep in mind that a protocol is not necessarily executed completely, or in the intended order, or with the intended data, because an attacker can rearrange instructions to his liking, and send arbitrary commands.

6 Correctness of the Card Implementation

A correct and secure protocol only works if it is implemented correctly in the terminal and the cardlet. However, the card implementation is more critical because it is almost impossible to correct an error if a large number of cardlets is already distributed. Furthermore, card programs are rather prone to errors because the limited resources require more or less tricky programming. And, they are more tractable to verification (no user interface, multi tasking etc.). Therefore, we deal only with the card implementation. We use JavaCard [10], a subset of Java that is tailored for limited resources (e.g. no garbage collection, no threads, no strings, no floats), and we use real JavaCard, not a lightweight, abstract, or simplified programming language. It should be noted that existing smartcards supporting JavaCard do not provide integers (32 bits) as a data type,

but only bytes (8 bits) and shorts (16 bits). For verification we use a dynamic logic [8] for JavaCard implemented in KIV [16].

The real communication between a terminal and a card happens not by sending and receiving abstract data types like *documents*, but by sequences of up to 256 bytes (APDUs, *application protocol data units* [9]). On the card side, a JavaCard method `process` receives the input as a byte array in an APDU object.

To prove the correctness of the card program the application developer must provide the following:

1. an encoding (a refinement) of those abstract data types that are sent to the card into byte arrays. Additionally, the length of the data must be fixed.
2. A refinement of the abstract data types used on the card to JavaCard data types.
3. An invariant for the data on the card.

With this information proof obligations for the consistency of the implementation with the protocols can be generated automatically, one for each protocol axiom plus the proof obligations for the refinement. We illustrate this with the second axiom of the *pay* protocol, *pay_cardlet_2* (see 5.2).

The cardlet expects to receive a number (the points to load as an integer) and a nonce. The encoding into a byte array is predefined (nonces are integers, and integers are coded in the usual manner as two's complement). We specify that the number will be 2 bytes long, and the nonce 6 bytes. These are additional requirements a terminal program must fulfill. With this definition $get_int(getparam(1, doc))$ becomes $getint(st, 2)$.

The value on the (abstract) cardlet is stored in an integer field `value`. Since we know that the number will always be between 0 and 32767 we can store the number in a `short` field, i.e. the integer field `value` is refined to a short field `value.cl.value` thus becomes $st[value].intval$ (the short value is converted back into an integer). The invariant $inv(st)$ is $0 \leq st[value].intval$. It is indeed necessary in this case because otherwise the test $st[value].intval - getint(st, 2)$ could produce a short underflow – one of the pitfalls of bounded resources.

Automatically *send* becomes a method invocation of `process`, and *response-doc* an *ISOException*. So the resulting proof obligation is

$$\begin{aligned}
 & \text{apduIns}(st) = \text{pay} \\
 & \wedge (st[value].intval - getint(st, 2) < 0 \vee getint(st, 2) \leq 0) \\
 & \wedge inv(st) \wedge st = st_0 \wedge ok(st) \\
 \rightarrow & \langle st / \text{process}(apdu); \rangle \\
 & (\quad st[\text{cardlet}] = st_0[\text{cardlet}] \wedge inv(st) \\
 & \quad \wedge \text{ISOException}(\text{SW_DATA_INVALID}, st))
 \end{aligned}$$

Every protocol axiom for the card becomes one proof obligation (2 for pay, 4 for load, 1 for an illegal instruction), plus 1 for the card initialization, plus 1 for the invariance property gives a total of 9 proof obligations for the example. They are verified with KIV and together ensure that the cardlet is consistent with the protocol specification.

7 Related Work

Formal Methods and UML. A considerable amount of work has been done to formalize the semantic of UML, e.g. [7] defines a formal execution semantic for activity diagrams. In contrast to us, the authors of this paper do not deal with modelling communication protocols but with workflow modelling. [20] presents an approach to transform class diagrams into Z specifications. In contrary to most publications, which only cover a subset of UML, [13] describes an approach to formalize UML models as a whole. The concept is to translate the UML diagrams into CASL-LTL specifications. All these publications aim at providing a complete semantics for some or all UML diagrams. This is not our intention, we use only that part of the expressiveness of UML that is necessary to describe our limited scenarios and for these parts we define a semantic suitable for our task.

Protocol Verification. The most seminal work for formal crypto protocol verification is, perhaps, by Burrows, Abadi, and Needham [5]. The algebraic specification technique we use is chiefly inspired by Paulson [12], where authentication protocols are analyzed. However, we distinguish between different types of participating agents, use different capabilities for the attacker, and introduce an internal state for the terminal and card (a very natural approach).

Java Semantics and Calculi. The formal analysis of Java (or JavaCard) has become an issue in the last years. A good starting point for a formal treatment of Java is [1]. There exist a number of Hoare calculi for Java (e.g. [19]), and another dynamic logic [4], but the proof support (heuristics, automation, user interface) is not clear.

JavaCard Projects. The VerifiCard [18] and the KeY project [11] (both home pages provide more information and references) are two projects aimed at correct JavaCard programs, but do not include design, correctness, and security of the protocols into an integrated formal approach. The contribution of our approach is to support the entire development process from application design and security to implementation correctness. To our knowledge, this has not been done before.

8 Summary

We have presented a method for the formal development of secure smartcard applications. The method combines and integrates different techniques (with algebraic specifications at the core) to tackle the different problems: objects and distributed systems, attackers and cryptographic protocols, JavaCard programs and limited resources. The techniques include UML models enriched by algebraic specifications, and dynamic logic for JavaCard verification. The method is tailored to take advantage of the special features of smartcard scenarios, and to make proving security and correctness as easy as possible. The method is illustrated with a small but surprisingly complex example, a copy card. The approach is implemented in the KIV specification and verification system.

References

1. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999. 332
2. R. Anderson and R. Needham. Programming satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer LNCS 1000, 1995. 319
3. M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000. 319
4. B. Beckert. A dynamic logic for the formal verification of java card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards*. Springer LNCS 2041, 2000. 332
5. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical report, SRC Research Report 39, 1989. 332
6. CoFI: The Common Framework Initiative. Casl — the CoFI algebraic specification language tentative design: Language summary, 1997. <http://www.brics.dk/Projects/CoFI>. 325
7. R. Eshuis and R. Wieringa. A formal semantics for uml activity diagrams - formalising workflow models. Technical report, University of Twente, February 2001. <http://wwwhome.cs.utwente.nl/~eshuis/adsem.pdf>. 332
8. D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979. 331
9. International Standards Organization, Geneva. *ISO 7816 – Identification Cards – Integrated circuit(s) cards with contacts*. several parts, 1987 – 1997. 331
10. *Java Card 2.1.1 Specification*, 2000. <http://java.sun.com/products/javacard/>. 330
11. KeY project homepage. <http://i12www.ira.uka.de/~key>. 332
12. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998. 324, 325, 332
13. G. Reggio, M. Cerioli, and E. Astesiano. An Algebraic Semantics of UML Supporting its Multiview Approach. In *Proc. AMiLP 2000 of the Twente Workshop on Language Technology n. 16, Enschede, University of Twente*, 2000. 332
14. W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, Berlin, 1995. 319
15. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proc. of the 6th European Symposium on Research in Computer Security (ESORICS)*, LNCS 1895, pages 17–36. Springer, 2000. 323
16. Kurt Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at <http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/>. 331
17. The Object Management Group (OMG). *OMG Unified Modeling Language Specification*, 1999. <http://www.omg.org/technology/uml>. 323
18. VerifiCard project homepage. <http://www.verificard.org>. 332
19. David von Oheimb. Axiomatic semantics for Java^{light} in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. 332
20. Benkt Wangler and Lars Bergman, editors. *An Overview of RoZ : A Tool for Integrating UML and Z Specifications*, volume 1789 of *Lecture Notes in Computer Science*. Springer, 2000. 332

Extending JML Specifications with Temporal Logic

Kerry Trentelman^{1*} and Marieke Huisman²

¹ Automated Reasoning Group, RISE, Australian National University

Kerry.Trentelman@anu.edu.au

² INRIA Sophia-Antipolis, France

Marieke.Huisman@sophia.inria.fr

Abstract. This paper proposes an extension of the Java Modeling Language (JML) with temporal specifications. The extension is inspired by the patterns and specification language used within the Bandera project, and is especially tailored to specify properties of Java(Card) programs; for example, it allows the exceptional behaviour of methods to be specified. In the tradition of JML, the extension has been designed to be simple, easy and intuitive to use for software engineers. As an example, we show how the JML extension can be used to specify temporal aspects of the JavaCard API. Later, a semantics for the extension is discussed. We show how to translate a subset of the extension back into standard JML, thus allowing the re-use of existing verification techniques for JML. For the ‘new’ part of the language, a trace-based semantics is given.

1 Introduction

Although the feasibility of program verification is now acknowledged, it is still labour-intensive and complex, requiring an appropriate specification language and an understanding of the underlying semantics. (See [10,12] for some examples of verification of Java programs using theorem proving.) Design by Contract made a number of in-roads into the area, using assertions to incorporate specification information into the program code itself [21]. Eiffel was the first programming language based on Design by Contract, its assertions describing the code’s implicit contracts, specifying requirements such as: pre-conditions that a client must meet before a method is invoked, post-conditions that a method must meet after it executes, and class invariants that must be preserved by each method. Following this approach, several specification languages designed for Java have since been developed; among them the Java Modeling Language, or JML [17,16]. However, verifications of Java programs using specification languages based on Design by Contract are, as a consequence of their nature, necessarily restricted to their functional behaviour.

* Supported by the Australian Research Council and Gemplus France via an APA(I) scholarship, and a visiting fellowship from INRIA Sophia-Antipolis.

At the same time, in the field of model checking, attention is shifting towards the verification of software and in particular to the interactions between the different components in a program. (See [4,8] for some examples of model checkers applied to Java.) A drawback of these approaches is that the specifications are often given in a complicated logic which makes them difficult to understand for many Java programmers. These specifications are also usually given separately – they are not part of the program – in contrast to *e.g.* Eiffel or JML specifications. To get a closer integration between functional and temporal specifications, these different specification techniques should be integrated.

Java with Assertions, or Jass, developed at the University of Oldenburg, is a first attempt to bridge this gap. Jass allows Java classes to be annotated with specifications in the form of assertions [3,14]. In particular, Jass features trace assertions which are used to monitor the ordering of method invocations and returns. Trace assertions – whose semantics are based on CSP – can be used to specify the order in which methods can or must be invoked, and also the conditions under which a method can be invoked. However, Jass trace assertions cannot be integrated with other Jass specifications, hence a specification stating *e.g.* that after a particular method call a certain variable should always be positive, is not allowed.

JML was designed by Gary Leavens *et al.* at Iowa State University. JML is used to specify Java classes and interfaces [17,16]. Many of the standard assertion features of JML are similar to Jass. However, JML’s ability to handle interfaces and abstract classes, and its feature of model variables – described in more detail in the next section – makes it much more expressive.¹ We therefore propose an extension of JML with temporal logic which is easier to understand than Jass trace assertions, and which can be used to specify the temporal behaviour of interactions between different objects in a Java(Card) program.

For the extension of JML we are inspired by the SanTos Specification Pattern project, a branch of the Bandera project [2]. The overall goal of the Bandera project is the development of a tool which automatically extracts abstract mathematical models based on specified properties from Java source code. The tool can then render these models in the input language of several different model-checking tools. Bandera employs program analysis, abstraction and transformation techniques in order to get finite-state models [4]. Specifications of Java source code are made using the Bandera Specification Language, BSL, which implements so-called specification patterns. A specification pattern is a language independent set of commonly used specification constructs for finite-state verification, with mappings into different temporal logics. Because of this implementation, BSL is independent from the source code it specifies [22,5]. However, BSL does not allow properties of exceptions to be specified; this is where it differs from JML and our work.

In order to develop a better understanding of what kind of specification constructs are necessary for our extension language, we looked at the JavaCard

¹ In fact, the designers of Jass are currently considering switching to JML as an assertion language.

API [1]. As part of the LOOP project at the University of Nijmegen [19], specifications for the JavaCard API have been written in JML [7,20]. A goal is to eventually verify these specifications with respect to the reference implementation of the API. In several specifications, temporal aspects – such as the order of method invocations – have been specified using model variables, simulating a state automaton. (Sect. 4 shows how similar specifications look in our JML extension.) However, some temporal specifications have not been given at all by the LOOP team, since standard JML is inadequate to specify these directly.

The temporal extension of JML presented in this paper has been designed with the following goals in mind:

- the language should be intuitive and easy to understand for software engineers familiar with Java(Card);
- the language should be tailored to specify properties about Java(Card) programs;
- the language should be integrated with standard JML, *i.e.* it should be possible to use existing JML expressions in the temporal logic formulae;
- the language should provide all specification constructs that have been identified as important by the specification patterns project and which are relevant for Java; and
- the language should have a clear semantics and appropriate verification techniques.

A subset of the specifications allowed by our language extension can be translated back into standard JML specifications; they can therefore be verified using standard verification techniques for JML, as advocated in *e.g.* the LOOP project [13]. This subset describes safety properties – nothing bad will happen – meaning that if a program gets into a particular state then a certain property is satisfied. Specifications which cannot be translated back into standard JML are liveness properties – eventually something good will happen – meaning that certain program states will be reached. This paper describes the semantics of both safety and liveness properties, but it does not present appropriate verification techniques for liveness properties; this is a topic of future work.

The paper is outlined as follows: in Sect. 2 JML is described in more detail; Sect. 3 outlines specification patterns; Sect. 4 informally introduces our specification language and presents an example featuring the transaction mechanism of JavaCard; Sect. 5 discusses the semantics of our specification language; and finally, in Sect. 6, we draw conclusions and discuss future work.

2 The Java Modeling Language

JML has been designed to be easy to use for Java programmers with little experience in logic. It allows class invariants and pre- and post-conditions for methods and constructors to be written within the program code. Specifications are formulated by making use of (side-effect-free) boolean Java expressions; they are written as Java comments, following `/*@` or enclosed between `/*@` and `*/`.

The additional `@` notifies the JML tool that it is a JML specification rather than an ordinary comment. The JML tool is a pre-compiler that translates specified programs into Java programs which explicitly monitor assertions at run-time. Specification violations that are found throw Java exceptions. A simple JML method specification is described below:

```
/*@ public normal_behavior
    requires P;
    ensures Q;
*/
```

Such a specification states that if a pre-condition `P` holds at the invocation of a public method then the method terminates normally, *i.e.* does not throw an exception, and the post-condition `Q` will hold at the end of the method invocation. This is comparable to a total correctness formula in Hoare logic.² A variation on the `normal_behavior` specification is a `behavior` specification. Here also, the conditions under which a method may, may not, or must throw an exception can be specified. A typical `behavior` specification looks as follows:

```
/*@ public behavior
    requires P;
    ensures Q;
    signals (e_1) R_1;
    :
    signals (e_n) R_n;
/*
```

If the pre-condition `P` holds at the beginning of a public method's invocation and the method terminates normally, then the corresponding post-condition `Q` is satisfied. If the method does not terminate normally and instead throws an exception which is an instance of a subclass of one of the `e_i`'s listed, then the exception's respective condition `R_i` should hold. These specifications can be translated into formulae of an extended Hoare logic dealing with abrupt termination outlined in [11,13].

Within (exceptional) post-conditions, expressions of the form `\old(E)` can be used, denoting the value of the expression `E` evaluated in the pre-state of the method.

To get a higher level of abstraction in the specifications, JML allows the declaration of so-called model variables (using the keyword `model`) which are variables that exist only within specifications. This is an extension of Hoare's data abstraction technique [9]. Model variables are typically used to represent the internal state of an object in an abstract way. For example, a typical specification

² Notice that only partial correctness can be monitored. To establish total correctness one needs to do formal verification.

of an object `BoundedThing`³ has two model variables: `size` and `MAX_SIZE`, denoting the size and the maximum size of the `BoundedThing`, respectively. Model variables can be ‘initialised’ by specifying an `initially` clause, *i.e.* `size` may be initialised to 0 via the specification `//@ initially size == 0`. To change the value of model variables, a `set` clause specifies an ‘assignment’ to a model variable.

Finally, there are specification constructs which describe the behaviour of a whole class. Examples of these are invariants and constraints. An `invariant` clause declares those properties that are true in all publicly visible, reachable states of an object, *i.e.* for each state that is outside of a public method’s execution. An invariant is supposed to be established by the class constructors and to be preserved by each (public) method. For example, the specification `//@ invariant 0 <= size && size <= MAX_SIZE` means that the value of `size` is always bound between 0 and `MAX_SIZE`. Within a method’s execution an invariant may be broken, but before the method terminates the invariant has to be re-established, even when the method terminates exceptionally. A `constraint` clause relates the pre-state and the post-state of every method, restricting how the variables may be changed by a method. For example, `//@ constraint MAX_SIZE == \old(MAX_SIZE)` specifies that the value of `MAX_SIZE` cannot change, since the value in a method’s post-state, `MAX_SIZE`, must be always equal to the value in its pre-state, `\old(MAX_SIZE)`.

3 Specification Patterns

Influenced by the notion of design patterns, the specification pattern approach to finite-state verification was first proposed by Matthew Dwyer *et al.* of the Specification Pattern project at the SanTos Laboratory, Kansas State University [6]. Specification patterns describe the structure of some aspect of a program’s behaviour and provide expressions of this behaviour in common formalisms such as linear temporal logic, LTL, or computational tree logic, CTL. (See *e.g.* [18] for a description of these formalisms.) Patterns capture an experience base of expert specifiers in temporal logic; users need not have an in-depth knowledge of the underlying semantics of the specification language they are using, they just need to look up the relevant pattern to match the requirement being specified. As explained in Sect. 1, the specification patterns form the basis for BSL, the input language for the model-checker front-end tool Bandera.

Patterns include: *occurrence* patterns such as universal, existence, absence and bounded existence; and *ordering* patterns such as precedence and response. Each pattern has a scope which gives the extent of the program execution over which the pattern must hold. The intents of the patterns listed above are given as follows:

- *universal*: a given state or event occurs throughout a scope
- *existence*: a given state or event must occur within a scope

³ All examples in this section are taken from the *Preliminary Design of JML* [16].

- *absence*: a given state or event does not occur within a scope
- *bounded existence*: a given state or event must occur at most/at least/exactly n times within a scope.
- *precedence*: a state or event must always be preceded by another state or event within a scope
- *response*: a state or event must always be followed by another state or event within a scope

There are five main scopes, describing time intervals which are closed at the left and open at the right.

- *globally*: throughout the entire program’s execution
- *after*: the execution after a given state or event
- *before*: the execution up to a given state or event
- *between*: any part of the execution between two designated state or events
- *after-until*: the execution after a given state or event until another state or event, or throughout the rest of the program if there is no subsequent occurrence of that state or event

Notice that the scope after-until describes what is often known as a weak until; it is not necessary that the state or event mentioned in the until actually happens.

In the tradition of design patterns, a specification pattern is given in the form of a package comprising its name, a precise statement of its intent, mappings into common specification formalisms, *e.g.* CTL and LTL, examples of known uses, and relationships to other patterns. The Specification Pattern project’s website [23] houses a large number of commonly occurring specification patterns.

4 Temporal Specifications in JML

Inspired by the Bandera patterns, and based on our experiences with specifying the temporal behaviour of the JavaCard API, we propose an extension of JML with temporal specifications. The syntax for our proposed temporal specifications is given in Fig. 1. The specifications are easy to understand and are able to hide much of the technicalities of temporal logic. The intention is that they can be written anywhere a class invariant can occur. In this section we present the informal intuition behind the different specification constructs and we discuss an example. In the next section we discuss the semantics underlying our JML extension more formally.

The basis of our logic are trace properties which describe a property that is true for (a part of) a program’s execution. Every temporal formula describes that part of the program’s execution trace over which the property should hold, *e.g.* **after** or **before** a certain event has happened, or **between** two particular events. We make a distinction between **until** or **unless**: until is what is often known as a strong until, in every execution the event has to happen; while unless can describe infinite behaviour in which the event may never happen. Notice also the asymmetry of **after** and **before** compared with **until** and **unless**: the first two can contain temporal formulae as subexpressions, whereas

$\langle \text{TempForm} \rangle$	$=$ <ul style="list-style-type: none"> after $\langle \text{Events} \rangle \langle \text{TempForm} \rangle$ before $\langle \text{Events} \rangle \langle \text{TraceProp} \rangle$ $\langle \text{TraceProp} \rangle$ until $\langle \text{Events} \rangle$ $\langle \text{TraceProp} \rangle$ unless $\langle \text{Events} \rangle$ between $\langle \text{Events} \rangle \langle \text{Events} \rangle \langle \text{TraceProp} \rangle$ at most $\langle \text{nat} \rangle \langle \text{Events} \rangle$ $\langle \text{TraceProp} \rangle$
$\langle \text{TraceProp} \rangle$	$=$ <ul style="list-style-type: none"> always $\langle \text{StateProp} \rangle$ eventually $\langle \text{StateProp} \rangle$ never $\langle \text{StateProp} \rangle$ $\langle \text{TraceProp} \rangle$ & $\langle \text{TraceProp} \rangle$ $\langle \text{TraceProp} \rangle$ $\langle \text{TraceProp} \rangle$
$\langle \text{Events} \rangle$	$=$ <ul style="list-style-type: none"> $\langle \text{Event} \rangle$ $\langle \text{Event} \rangle, \langle \text{Events} \rangle$
$\langle \text{Event} \rangle$	$=$ <ul style="list-style-type: none"> $\langle \text{method} \rangle$ called $\langle \text{method} \rangle$ normal $\langle \text{method} \rangle$ exceptional $\langle \text{method} \rangle$ terminates
$\langle \text{StateProp} \rangle$	$=$ <ul style="list-style-type: none"> $\langle \text{JMLProp} \rangle$ $\langle \text{method} \rangle$ enabled $\langle \text{method} \rangle$ not enabled $\langle \text{StateProp} \rangle$ & $\langle \text{StateProp} \rangle$ $\langle \text{StateProp} \rangle$ $\langle \text{StateProp} \rangle$ $!\langle \text{StateProp} \rangle$

Fig. 1. Temporal specification syntax

the latter only can contain trace properties. We choose to do this in order to keep the intuitive meaning of expressions clear and to disallow expressions such as (**before** $e_1 \phi$) **unless** e_2 , whose semantics is unclear in the case e_2 happens before e_1 . A formula **between** $e_1 e_2 \phi$ describes the same behaviour as **after** $e_1 (\phi$ **until** $e_2)$. If no ‘trace delimiter’ is specified, the property should hold over the complete trace. With every trace delimiter a set of events can be given as arguments; this means that one of these events has to happen. A special temporal formula is **at most** $n e$ which can be seen as syntactic sugar for **after** $e \dots (\text{after } e \text{ always false})$ (where the ellipsis stands for $n - 1$ occurrences of **after** e) meaning that no state is reachable after an event e occurs more than n times.

For a method m , the events that we distinguish are: m **called**, m **normal**, m **exceptional** and m **terminates**. Note that the occurrence of an event

m **terminates** means that either an event m **normal** or m **exceptional** has occurred. Every event describes a state change, and the trace properties describe those properties that have to hold in a sequence of states. Essentially, we have **always** and **eventually** properties which describe that a property has to be true in every state of the sequence, or in at least one state of the sequence, respectively. In the syntax a keyword **never** is introduced: **never** ψ being an abbreviation for **always** $!\psi$, where $!$ is the Java (and JML) negation.

The properties that have to hold for a particular state are ordinary JML properties – *i.e.* JML expressions with type boolean – with two extensions: m **enabled** and m **not enabled**. For a method m , a property m **enabled** is true in a particular state whenever: if m is called in that state and it terminates, it terminates normally. The property m **not enabled** expresses the contrary: if m is called and it terminates, it will throw an exception; hence if m is known to terminate, this expresses the same as $!m$ **enabled**. As in JML, we use the Java logical connectives $\&$ and $|$ to combine trace and state properties.

4.1 An Example: Temporal Aspects of the JCSYSTEM Class

The extension of JML, as discussed above, is based on our experiences with writing specifications for the temporal aspects of the JavaCard API [1]. As an example, the proposed temporal specifications for the JavaCard JCSYSTEM class will be presented. This class includes methods which control applet execution, resource and atomic transaction management, and inter-applet object sharing on a Java smart card [15]. Temporal aspects of the API for this class relate solely to transaction management which is of crucial importance to the card. The transaction mechanism works as follows: when an application creates or updates data on a Java smart card, the integrity of the data needs to be preserved throughout the communication. Either all the data is updated during the communication, or in the case of an interruption, it reverts back to its initial state. A call to the method JCSYSTEM.beginTransaction initiates the beginning of a set of updates. Each object update after this point is only conditionally updated; the field may appear to be updated, but the update is not yet committed. Conditionally updated fields are stored in the commit buffer. Data is committed to persistent storage – and cleared from the buffer – once the applet has called JCSYSTEM.commitTransaction. In the case of power loss or some other system failure before this method is called, conditionally updated fields revert back to their original values. The method JCSYSTEM.abortTransaction is called if the applet encounters any internal problems, *i.e.* an exception is thrown, or if it decides to cancel the transaction.

Quoted directly from the API documentation [1], the following specifications describe temporal aspects of the JCSYSTEM class:

- beginTransaction throws TransactionException.IN_PROGRESS if a transaction is already in progress;
- abortTransaction throws TransactionException.NOT_IN_PROGRESS if a transaction is not in progress; and

- `commitTransaction` throws `TransactionException.NOT_IN_PROGRESS` if a transaction is not in progress.

Firstly, `beginTransaction` throws an exception if a transaction is already in progress. A transaction in progress describes that state in which `beginTransaction` has been successfully called and neither `abortTransaction` nor `commitTransaction` has yet been invoked. Hence, after `beginTransaction` is called, if it is called again before `abortTransaction` or `commitTransaction`, it will throw an exception. We can write this in our proposed specification language as:

```

after beginTransaction called
  (always beginTransaction not enabled
   unless abortTransaction called, commitTransaction called)

```

The above API specifications also suggest complementary behaviour: if `beginTransaction` is called when a transaction is not in progress, the method will have normal behaviour. A transaction not in progress describes the state in which either `abortTransaction` or `commitTransaction` has been successfully called and `beginTransaction` not yet invoked.⁴ In our proposed specification language we can write this complement as:

```

after abortTransaction called, commitTransaction called
  (always beginTransaction enabled
   unless beginTransaction called)

```

In a similar way, based on the documentation above, specifications can be given which describe when `commitTransaction` and `abortTransaction` will, or will not be enabled. Combining these specifications with the specification for `beginTransaction` results in the following two class specifications:

```

after beginTransaction called
  (always (beginTransaction not enabled &
           abortTransaction enabled &
           commitTransaction enabled)
   unless abortTransaction called, commitTransaction called)

```

```

after abortTransaction called, commitTransaction called
  (always (beginTransaction enabled &
           abortTransaction not enabled &
           commitTransaction not enabled)
   unless beginTransaction called)

```

Members of the LOOP group, Hans Meijer and Erik Poll have already given a JML specification of the transaction methods of the `JCSYSTEM` class, outlined in [7]. This specification relies on the model variable `_transactionDepth` which is assigned the values 0 and 1. When `_transactionDepth` is 0, a transaction is

⁴ Of course, only `beginTransaction` will be enabled initially.

said to be not in progress; when it is 1, a transaction is in progress. For example, specifications for the `abortTransaction` and `commitTransaction` methods both contain the clause `//@ ensures _transactionDepth == 0`, whereas the specification for `beginTransaction` includes `//@ requires _transactionDepth == 0`. If there were more than two transaction depths, it could make for a very long and complicated specification. We believe that by hiding this beneath our temporal extension of JML makes specifications such as those for the transaction mechanism much simpler and more intuitive.

5 Semantics

After presenting the informal meaning of our proposed language extension it is necessary to describe the formal semantics. As mentioned before, a subset of the temporal formulae in our language extension can be translated back into standard JML. Below, in Sect. 5.2 we discuss how this can be done, and we show the resulting proof obligations for the specification of the transaction mechanism as presented in the previous section. But first we define the semantics.

5.1 Trace-Based Semantics

To give a semantics to our temporal extension we define for each possible execution trace whether it satisfies the formula. As mentioned above in Sect. 4, the keywords **between** and **at most** can be expressed using **after**, **always** and **until**; therefore we need not consider these cases. Similarly, we do not explicitly give a semantics for **never**, as this can be expressed in terms of **always**.

Based on Emerson's terminology in [18], we assume that we have a linear time structure $M = (S, E, x, \rightarrow)$ where S is a set of states, E a set of events, x an infinite sequence of states $x(0), x(1), x(2), \dots$ describing an execution of the underlying program, and $\rightarrow \subseteq S \times E \times S$ is a transition relation denoting whether a state can be reached from another state by a particular event. We assume that every state transition is labelled by a unique event.

We define $M, x \models \phi$, meaning that temporal formula ϕ is true of execution trace (or path) x . When M is understood, we simply write $x \models \phi$.

We first introduce some notation. We use s_i to denote $x(i)$. The notation x^j denotes the *suffix path* $s_j, s_{j+1}, s_{j+2}, \dots$ and x_i^j denotes the *segment path* $s_i, s_{i+1}, s_{i+2}, \dots, s_{j-1}, s_j, s_j, s_j, \dots$. The segment path is infinite in length when the last state of the segment is repeated infinitely often.

We say that a set of events E 'holds' on a state s_i (written $s_i \models E$) if and only if $i > 0$ and there exists an e such that $s_{i-1} \xrightarrow{e} s_i$ and $e \in E$, *i.e.* the state s_i is reached by an E -transition. The set E does not hold on a state s_i (written $s_i \not\models E$) if and only if s_{i-1} and s_i are related by an event not in E , or if $i = 0$. That is, no event can hold at s_0 . Moreover, E does not hold on an execution trace x (written $x \not\models E$) if and only if for all j , $s_j \not\models E$.

We overload \models for defining temporal formulae, trace properties and state properties. We now define $x \models \phi$ on the structure of a temporal formula.

Definition 1 (Semantics of temporal formulae). *Given an execution path x , a set of events E , a temporal formula ϕ , and a trace property ψ , we define:*

$$\begin{aligned} x \models \mathbf{after} E \phi & \text{ iff } \forall j. s_j \models E \Rightarrow x^j \models \phi \\ x \models \mathbf{before} E \phi & \text{ iff } \forall j. s_j \models E \Rightarrow x_0^{j-1} \models \phi \\ x \models \psi \mathbf{until} E & \text{ iff } \exists j. s_j \models E \wedge x_0^{j-1} \models \psi \\ x \models \psi \mathbf{unless} E & \text{ iff } (x \models \psi \wedge x \not\models E) \vee (\exists j. s_j \models E \wedge x_0^{j-1} \models \psi) \end{aligned}$$

A temporal formula **after** $E \phi$ holds exactly in those execution traces for which, if an E -transition occurs, in the suffix path after this transition, the temporal formula ϕ holds. A temporal formula **before** $E \phi$ holds exactly in those execution traces for which, if an E -transition occurs, in the segment path prior to this transition, the temporal formula ϕ holds. A formula ψ **until** E is true for all execution traces in which an E -transition occurs, and the trace property ψ is satisfied on the segment path to this E -transition. In addition, ψ **unless** E is also true if no event in E occurs and the trace property ψ holds throughout.

Definition 2 (Semantics of trace properties). *Given an execution path x and a state property χ , we define:*

$$\begin{aligned} x \models \mathbf{always} \chi & \text{ iff } \forall j. x^j \models \chi \\ x \models \mathbf{eventually} \chi & \text{ iff } \exists j. x^j \models \chi \end{aligned}$$

The semantics for conjunction and disjunction of trace properties is standard.

This means that a state property χ is always true if it holds for all suffix paths, and it is eventually true if there exists a suffix path for which it is true.

Lastly, we define when a state property is satisfied on a path. We assume we have a semantics for JML (and Java) expressions and statements, described as a function $\llbracket - \rrbracket_{\text{JML}}$, which defines how to evaluate an expression or statement in a particular state. In particular, $\llbracket m \rrbracket_{\text{JML}}$ defines the meaning of a method call. Further, we assume that we have predicates **term?**, a method terminates; **norm?**, a method terminates normally; and **excp?**, a method terminates exceptionally. See [10] for an example of a semantics which allows this.

Definition 3 (Semantics of state properties). *Given an execution path x , a method name m , and a JML property ξ , we define:*

$$\begin{aligned} x \models \xi & \text{ iff } \llbracket \xi \rrbracket_{\text{JML}}(s_0) \\ x \models m \mathbf{enabled} & \text{ iff } \mathbf{term?} \llbracket m \rrbracket_{\text{JML}}(s_0) \Rightarrow \mathbf{norm?} \llbracket m \rrbracket_{\text{JML}}(s_0) \\ x \models m \mathbf{not enabled} & \text{ iff } \mathbf{term?} \llbracket m \rrbracket_{\text{JML}}(s_0) \Rightarrow \mathbf{excp?} \llbracket m \rrbracket_{\text{JML}}(s_0) \end{aligned}$$

The semantics for conjunction, disjunction and negation of state properties is standard.

Thus, state predicates are evaluated in the first state of x .

Hence, for the specification,

after beginTransaction called
(always beginTransaction not enabled
unless abortTransaction called, commitTransaction called),

the execution traces for which this is true are characterised as follows:⁵

$$\begin{aligned} \forall j. s_j \models bT \Rightarrow & \\ & ((\forall l. \text{term? } \llbracket bT \rrbracket_{\text{JML}}(s_{j+l}) \Rightarrow \text{excp? } \llbracket bT \rrbracket_{\text{JML}}(s_{j+l})) \wedge (\forall m. s_{j+m} \not\models cT, aT)) \\ & \vee (\exists k. s_{j+k} \models cT, aT \wedge \\ & \quad \forall l. \text{term? } \llbracket bT \rrbracket_{\text{JML}}(x_{j+1}^{j+k-1}(0)) \Rightarrow \text{excp? } \llbracket bT \rrbracket_{\text{JML}}(x_{j+1}^{j+k-1}(0))) \end{aligned}$$

5.2 Translating Temporal Formulae Back to JML

As discussed above, a subset of our language extension can be translated back into standard JML expressions. This subset expresses safety properties: if a program reaches a certain state then a property is satisfied, but it is not guaranteed that the program will actually reach this state. In this section we consider only safety properties formed using **after**, **unless** and **always**. The specification of the transaction mechanism is an example of a safety property. We use this example to illustrate the translation back to JML. This translation might seem verbose, but as it can be performed by a tool, this is of no importance.

In order to translate the specifications of the transaction mechanism back into JML, a number of model variables such as `m_called`, `m_normal` and `m_exceptional` are declared for each relevant method `m`. These model variables are given a boolean value, initially `false`, and are used to trace the behaviour of the program. In the first line of every method `m`, a JML set-statement is added setting `m_called`.

```
//@ set m_called = true;
```

Moreover, the specification of method `m` contains the following normal and exceptional post-conditions:

```
//@ ensures m_normal & !m_exceptional;
//@ signals (Exception) m_exceptional & !m_normal;
```

These state that if the method terminates either normally or abruptly then the model variables are appropriately set.

The top-level expression in the specification of the transaction mechanism is **after** $e_1 \phi$, where ϕ is some temporal formula. Provided that ϕ is a safety property, this means that if an event e_1 has happened, the property ϕ is satisfied. In general, the safety property ϕ expresses one of the following:

- some trace property ψ holds forever, *i.e.* **always** ψ ;
- if an event e_2 happens, a safety property ϕ_2 holds, *i.e.* **after** $e_2 \phi_2$;
- ψ holds unless some event e_2 happens, *i.e.* **always** ψ **unless** e_2 .

In the last case, if such an event e_2 happens, it will ensure that e_1 is no longer true. Therefore, we would like to add an invariant stating that ψ holds when e_1 is true.

⁵ For readability, we use bT , cT and aT to denote both the method and the event method called.

```

class JCSystem {

/*@ public behavior
    ensures \old(beginTransaction_called) => false;
    signals (Exception) \old(abortTransaction_called)|
                \old(commitTransaction_called) => false;
*/
    beginTransaction(){
        //@ set beginTransaction_called = true;
        //@ set abortTransaction_called = false;
        //@ set commitTransaction_called = false;
        ..
    }

/*@ public behavior
    ensures \old(abortTransaction_called)|
                \old(commitTransaction_called) => false;
    signals (Exception) \old(beginTransaction_called) => false;
*/
    abortTransaction() {
        //@ set abortTransaction_called = true;
        //@ set beginTransaction_called = false;
        ..
    }

// commitTransaction specification similar to abortTransaction
}

```

Fig. 2. JML specification for transaction mechanism

```
//@ invariant e_1 ==> psi;
```

To ensure that at the moment e_2 occurs e_1 is set to `false` again, we add an additional set assertion at the moment that e_2 happens. For example, in the specification **after m called** (ψ **unless n called**), the following additional set-statement needs to be added in the beginning of method `n`:

```
//@ set m_called = false;
```

In the case of the transaction mechanism, the state properties that are specified declare that methods have normal or exceptional behaviour. This cannot be expressed by a class invariant, but it can be expressed by specifying that in such a case a particular post-condition cannot be established. For example, to express that **after m called** (**always n not enabled**), the method `n` requires an

additional specification stating that after `m` has been called, the method cannot terminate normally.

```
//@ ensures m_called => false;
```

Fig. 2 shows the relevant parts of the JML specification that are constructed in this way. The specification for `commitTransaction` is not included as it is similar to that for `abortTransaction`.

It is interesting to compare this specification with the specification given by the LOOP project at [7]. For example, in the JML version of the specification, the method `beginTransaction` is specified by: `//@ requires _transactionDepth == 0` and `//@ ensures _transactionDepth == 1` within a `normal_behavior` specification, and `//@ requires _transactionDepth == 1` within an `exceptional_behavior` specification. We can see that in the LOOP specification `_transactionDepth == 0` if and only if `!beginTransaction_called` in our specification. (Similar results can be given for `abortTransaction_called` and `commitTransaction_called`.) Thus these specifications are equivalent.

6 Conclusions and Future Work

We have presented an extension of JML with temporal logic. The extension is based on the specification patterns proposed within the Bandera project and is tailored especially to Java. Although not all specifications in our language extension can be translated back into standard JML, we believe that by using our syntax, specifications are more readable and intuitive. Our language extension also allows the specification of liveness properties – eventually something good will happen – which currently cannot be specified in standard JML.

We have described a semantics for the language and we have discussed how temporal properties can be verified if they can be translated back into standard JML. It is future work to integrate the language extension into the grammar for standard JML and to develop an appropriate verification technique for liveness properties. More future work is to extend the language, allowing trace properties to state that events eventually or never happen and to integrate (a subset of) our language extension with a run-time assertion generator, as is done for trace assertions in the Jass project [14] and in the Java PathExplorer project [8].

Acknowledgements

We thank Dilian Gurov, Kim Sunesen, Rajeev Goré and the anonymous referees for their useful feedback and critical comments on this paper.

References

1. JavaCard 2.1. Platform Application Programming Interface (API) Specification. <http://java.sun.com/products/javacard/html/doc/>. 336, 341
2. The Bandera project. <http://www.cis.ksu.edu/santos/bandera/>. 335

3. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *Runtime Verification*, number 55(2) in ENTCS. Elsevier, 2001. 335
4. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-states models from Java source code. In M. Jazayeri and A. Wolf, editors, *22nd International Conference on Software Engineering*, pages 439–448. ACM Press, June 2000. 335
5. J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing Checkable Properties of Dynamic Systems: the Bandera Specification Language. Technical Report 2001-04, Kansas State University, Dept. of Comp. and Info. Sc., 2001. 335
6. M. Dwyer, G. Avrunin, and J. Corbett. Property Specification Patterns for Finite-state Verification. In M. Ardis, editor, *2nd Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998. 338
7. Formal specifications for the JavaCard API 2.1.1. http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html. 336, 342, 347
8. K. Havelund and G. Roşu. Java PathExplorer – A Runtime Verification Tool. In *6th Intl. Symp. on AI, Robotics and Automation in Space*, June 2001. 335, 347
9. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. 337
10. M. Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001. 334, 344
11. M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 284–303. Springer, 2000. 337
12. M. Huisman, B. Jacobs, and J. van den Berg. A Case Study in Class Library Verification: Java’s Vector Class. *Software Tools for Technology Transfer*, 3/3:332–352, 2001. 334
13. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, number 2029 in LNCS, pages 284–299. Springer, 2001. 336, 337
14. The JASS project. <http://semantik.informatik.uni-oldenburg.de/~jass/>. 335, 347
15. JavaCard Technology. <http://java.sun.com/products/javacard/>. 341
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report 98-06, Iowa State University, Dept. of Comp. Sc., 1998. Latest revision: August 2001. 334, 335, 338
17. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: Notations and Tools Supporting Detailed Design in Java. In ACM Press, editor, *OOPSLA 2000 Companion*, pages 105–106, October 2000. 334, 335
18. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier, 1990. 338, 343
19. The LOOP project. <http://www.cs.kun.nl/~bart/LOOP/>. 336
20. H. Meijer and E. Poll. Towards a Full Formal Specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security (E-smart 2001)*, number 2140 in LNCS, pages 165–178. Springer, 2001. 336
21. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. 334
22. Robby. Bandera Specification Language: a Specification Language for Software Model Checking. Master’s thesis, Kansas State University, 2000. 335
23. A Specification Pattern System. <http://www.cis.ksu.edu/santos/spec-patterns/>. 339

Algebraic Dynamic Programming

Robert Giegerich and Carsten Meyer

Faculty of Technology, Bielefeld University
33501 Bielefeld, Germany
{robert,cmeyer}@techfak.uni-bielefeld.de

Abstract. Dynamic programming is a classic programming technique, applicable in a wide variety of domains, like stochastic systems analysis, operations research, combinatorics of discrete structures, flow problems, parsing with ambiguous grammars, or biosequence analysis. Yet, no methodology is available for designing such algorithms. The matrix recurrences that typically describe a dynamic programming algorithm are difficult to construct, error-prone to implement, and almost impossible to debug.

This article introduces an algebraic style of dynamic programming over sequence data. We define the formal framework including a formalization of Bellman's principle, specify an executable specification language, and show how algorithm design decisions and tuning for efficiency can be described on a convenient level of abstraction.

AMAST Categories: programming methodology, specification languages, program transformation, functional programming

1 Introduction

The development of successful dynamic programming recurrences is a matter of experience, talent, and luck.
An anonymous referee

1.1 Motivation

Dynamic programming (DP) is one of the classic programming paradigms, introduced even before the term Computer Science was firmly established. Bellman's "Principle of Optimality" [2] belongs to the core knowledge we expect from every computer science graduate. Significant work has gone into formally characterizing this principle [22,20], formulating DP in different programming paradigms [21,7] and studying its relation to other general programming methods such as greedy algorithms [3].

The analysis of molecular sequence data has fostered increased interest in DP. Protein homology search, RNA structure prediction, gene finding, and discovery of regulatory signals in RNA pose string processing problems in unprecedented

variety and data volume. A recent textbook in biosequence analysis [8] lists 11 applications of DP in its introductory chapter, and many more in the sequel.

Textbook examples are typically restricted to simple problems that can be solved without methodical guidance. Developing a DP algorithm for an optimization problem over a nontrivial domain has intrinsic difficulties. The choice of objective function and search space are interdependent, and closely tied up with questions of efficiency. Once completed, all DP algorithms are expressed via recurrence relations between tables holding intermediate results. These recurrences provide a very low level of abstraction, and subscript errors are a major nuisance even in published articles. The recurrences are difficult to explain, painful to implement, and almost impossible to debug: A subtle error gives rise to a suboptimal solution every now and then, which can hardly be detected by human inspection. In this situation it is remarkable that neither the literature cited above, nor computer science textbooks [6,16,18,5,1,24] provide guidance in the development of DP algorithms.

1.2 Overview of Our Approach

The Algebraic Dynamic Programming approach (ADP) introduces a *conceptual* splitting of a DP algorithm into a recognition and an evaluation phase. The evaluation phase is specified by an *evaluation algebra*, the recognition phase by a *yield grammar*. Each grammar can be combined with a variety of algebras to solve different but related problems, for which heretofore DP recurrences had to be developed independently. Grammar and algebra together describe a DP algorithm on a high level of abstraction, supporting the development of ideas and the comparison of algorithms. A notation for yield grammars is provided that serves as a domain-specific language. Expressed in this language, the ADP algorithm can run as an executable prototype in a functional language, and can be translated into traditional DP recurrences implemented in an imperative language. Such implementation issues are treated in [14]. Here we concentrate on ADP as a programming method, and show how clever “tricks” found in the DP literature can be expressed transparently in this framework.

1.3 Related Work

The ADP method has been developed recently in the application context of biosequence analysis. An informal presentation to the bioinformatics community has appeared in [10]. An old bioinformatics challenge, the folding of saturated RNA secondary structures [26], has been solved via ADP recently, but was published without reference to this method [9]. The aspect of ambiguity in dynamic programming (not covered here) is studied in [11].

2 Dynamic Programming, Traditional Style

2.1 Palindromic Patterns in Strings

We consider strings over some alphabet. For $x = x_1 \dots x_m$ let $|x| = m$ and $x(i, j)$ denote the subword $x_{i+1} \dots x_j$. Note that $|x(i, j)| = j - i$. When x is clear from the context, we write (i, j) for $x(i, j)$. A string x is a palindrome, if $x = x^{-1}$. A separated palindrome [16] has the form $x = uvu^{-1}$, where we call v the “loop” of the palindrome, and u its “stem”.

The string `panamacanal` is a separated palindrome with an empty stem, containing several non-trivial separated palindromes (e. g. `anamacana`, `acana`). Being a separated palindrome is not just a Yes-No question: The string `abccba` can be considered a separated palindrome in four different ways. Intuitively, $u = \text{abc}$, $v = \varepsilon$ is more palindromic than (say) $u = \text{a}$, $v = \text{bccb}$. Let us introduce a scoring function $palScore(x)$, by means of which some palindromes score higher than others. Many scores are conceivable; for simplicity, we choose the length of the stem.

The two combinatorial optimization problems defined next will serve as running examples in this article.

Best separated palindrome (BSP):

Given x , find $palScore(x) = \max\{|u| \mid uvu^{-1} = x\}$.

Local best separated palindrome (LBSP):

Given x , find $localpalScore(x) = \max\{palScore(v) \mid uvw = x\}$.

$palScore(x)$ can be defined recursively, the basic fact being that awa is a palindrome of score $l + 1$ iff w is a palindrome of score l . In order to calculate $localpalScore(x)$, we must distinguish subwords that constitute a palindrome of certain score, and subwords that somewhere contain such a palindrome. We obtain recursive definitions for $palScore$ and $localpalScore$, using auxiliary functions p_x and q_x :

$$\begin{aligned}
 palScore(x) &= p_x(0, m) \\
 localpalScore(x) &= q_x(0, m) \\
 q_x(i, i) &= 0 && \text{for } i \in 0 \dots m \\
 q_x(i, j) &= \max\{p_x(i, j), q_x(i + 1, j), q_x(i, j - 1)\} && \text{for } i < j \\
 p_x(i, i) &= 0 && \text{for } i \in 0 \dots m \\
 p_x(i, i + 1) &= 0 && \text{for } i \in 0 \dots m - 1 \\
 p_x(i, j) &= \text{if } x_{i+1} = x_j \text{ then } 1 + p_x(i + 1, j - 1) \text{ else } 0 && \text{for } 0 \leq i, j \leq m \text{ and } i + 2 \leq j
 \end{aligned}$$

Implemented as recursive functions, p_x and q_x are inefficient, since most calls to $p_x(i, j)$ or $q_x(i, j)$ will be (re-)evaluated many times, leading to a runtime exponential in m . Dynamic programming is recursion plus tabulation. By a change of data type we achieve runtime $O(m^2)$. We re-interpret functions q_x and p_x as

two $m + 1$ by $m + 1$ matrices, whose entries are calculated (and re-used) via the above recurrences.

To judge the difficulty of developing this style of recurrences, the reader is invited to modify the recurrences as follows:

Exercise 1: Restrict analysis to palindromes with non-empty stem.

Exercise 2: Generalize the scoring scheme by introducing negative scores for non-stem parts of the palindrome.

2.2 Related Problems

For reason of space, the simple problems¹ BSP and LBSP must suffice here for the exposition of our method. Further generalizations (allowed mismatches in the stem, recursively nested palindromes, and minimal free energy scoring [27]) leads to the problem of predicting RNA secondary structures.

Everything we develop in this paper pertains as well to the problem of pairwise string comparison, also called string edit distance or string alignment, where DP often is the method of choice. Although some definitions change in detail, the overall analogy is described by the following observation: An optimal alignment of x and y is equivalent to finding an optimal approximate separated palindrome with loop $\$$ for the string $x\$y^{-1}$, where $\$$ is a separator symbol not occurring elsewhere. We have not seen this observation exploited in the literature, maybe because the partial reversal of subscript ranges in forming $x\$y^{-1}$ makes the recurrences look quite different. In the ADP approach – where there are no subscripts – the correspondence is obvious. Pairwise sequence alignment is used for the exposition in [10].

3 Algebraic Dynamic Programming

3.1 Basic Terminology

Alphabets. An *alphabet* \mathcal{A} is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{ax \mid a \in \mathcal{A}, x \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geq 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.

Signatures and algebras. A (single-sorted) signature Σ over some alphabet \mathcal{A} consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1 \dots s_{k_o} \rightarrow S$, where each s_i is either S or \mathcal{A} . A Σ -algebra \mathcal{I} over \mathcal{A} , also called an interpretation, is a set $\mathcal{S}_{\mathcal{I}}$ of values together with a function $o_{\mathcal{I}}$ for each operator o . Each $o_{\mathcal{I}}$ has type $o_{\mathcal{I}} : (s_1)_{\mathcal{I}} \dots (s_{k_o})_{\mathcal{I}} \rightarrow \mathcal{S}_{\mathcal{I}}$ where $\mathcal{A}_{\mathcal{I}} = \mathcal{A}$.

A *term algebra* T_{Σ} arises by interpreting the operators in Σ as *constructors*, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables, $T_{\Sigma}(V)$, with $V \subset T_{\Sigma}(V)$. By convention, operator names are capitalized in the term algebra.

¹ In fact, for the given simplistic score function, BSP and LBSP can be solved more efficiently using suffix trees than via dynamic programming.

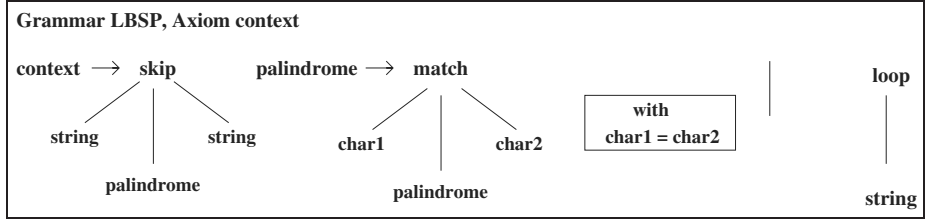


Fig. 1. The tree grammar LBSP for local separated palindromes (see Section 2.1)

Tree grammars. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. According to the special role of \mathcal{A} , only leaf nodes can carry symbols from \mathcal{A} . A term/tree with variables is called a *tree pattern*. A tree containing a designated occurrence of a subtree t is denoted $C[\dots t \dots]$.

A tree language over Σ is a subset of T_Σ . Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars originally studied in [4]. In [13] they were redefined to specify term languages over some signature. Our further specialization so far lies solely in the distinguished role of \mathcal{A} .

Definition 1 (Tree grammar over Σ and \mathcal{A} .)

A (regular) tree grammar \mathcal{G} over Σ and \mathcal{A} is given by

- a set V of nonterminal symbols
- a designated nonterminal symbol Ax called the axiom
- a set P of productions of the form $v \rightarrow t$, where $v \in V$ and $t \in T_\Sigma(V)$.

The derivation relation for tree grammars is \rightarrow^* , with $C[\dots v \dots] \rightarrow C[\dots t \dots]$ if $v \rightarrow t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma \mid v \rightarrow^* t\}$, the language of \mathcal{G} is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Ax)$. \square

For brevity, we add a lexical level to the grammar concept, allowing strings from \mathcal{A}^* in place of single symbols. By convention, **char** denotes an arbitrary character, **string** an arbitrary string. Also for brevity, we allow syntactic conditions associated with righthand sides. See Figure 1.

The yield of a tree is normally defined as its sequence of leaf symbols. Here we are only interested in the symbols from \mathcal{A}^* ; nullary constructors are not considered part of the yield. Hence the yield function y on $T_\Sigma(V)$ is defined by $y(t) = w$, where $w \in (\mathcal{A} \cup V)^*$ is the string of leaf symbols in left to right order.

3.2 Representing the Search Space by a Term Algebra

Any dynamic programming algorithm implicitly constructs a search space from its input. The elements of this search space have been given different names: *policy* in [2], *solution* in [22], *subject under evaluation* in [10]. Since the former two terms have been used ambiguously, and the latter is rather technical, we

shall use the term *candidate* for elements of the search space. Each candidate will be evaluated, yielding a *final state*, a *cost*, or a *score*, depending whether you follow [2], [22] or [8]. We shall use the term *answer* for the result of evaluating a candidate.

Typically, there is an ordering defined on the answer data type. The DP algorithm returns a maximal or minimal answer, and if so desired, it also reports one or all the candidates that evaluate(s) to this answer. Often, the optimal answer is determined first, and a candidate that led to it is reconstructed by backtracking. The candidates themselves do not have an explicit representation during the DP computation. This is the point that we will change.

Imagine that during computing the answer, we did not actually call those functions that perform the evaluation. Instead, we would apply them symbolically, building up a formula that – once evaluated – would yield this answer value. We shall make this formula an explicit representation of the candidate. For LBSP this is done in Figure 2.

These are the key ideas of algebraic dynamic programming:

Phase separation: We conceptually distinguish a recognition and an evaluation phase.

Term representation: Individual candidates are represented as elements of a term algebra T_Σ ; the set of all candidates is described by a tree grammar.

Recognition: The recognition phase constructs the set of candidates arising from a given input string, using a device called tabulating yield parser.

Evaluation: The evaluation phase interprets these candidates in a concrete Σ -algebra, and applies the objective function to the resulting answers.

Phase amalgamation: To retain efficiency, both phases are amalgamated in a fashion transparent to the programmer.

The virtue of this approach is that the conglomeration of issues bemoaned above – the recurrences deal with search space construction, evaluation and efficiency concerns in a non-separable way – is resolved by algorithm development on the more abstract level of grammars and algebras. In Section 5 we explain the benefits from the viewpoint of programming methodology, while [14] shows how an ADP algorithm, in spite of its abstractness, can be implemented without loss of efficiency.

3.3 Evaluation Algebras

Definition 2 (Evaluation algebra.) Let Σ be a signature over \mathcal{A} with sort symbol *Ans*. A Σ -evaluation algebra is a Σ -algebra augmented with an objective function $h : [Ans] \rightarrow [Ans]$, where $[Ans]$ denotes lists over *Ans*. \square

In most DP applications, the objective function minimizes or maximizes over all answers. We take a slightly more general view here. The objective may be to calculate a sample of answers, or all answers within a certain threshold of optimality. It could even be a complete enumeration of answers. We may compute the size of the search space or evaluate it in some statistical fashion, say by

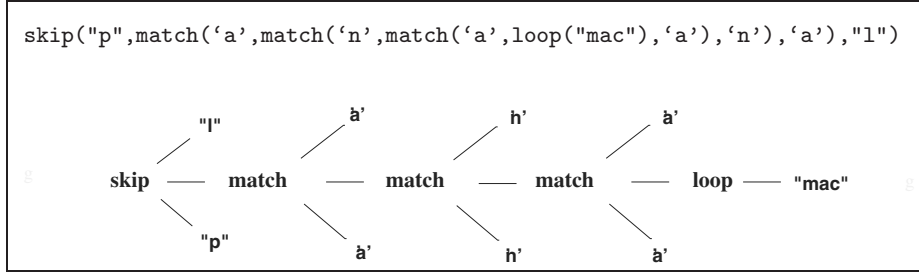


Fig. 2. The term representation of a LBSP candidate c for the input sequence panamacanal, and the tree representation of this term (lying on its side)

$Ans_{UNIT} = \mathbb{N}$	$Ans_{WEIGHT} = \mathbb{N}$
$match(a, s, b) = s + 1$	$match(a, s, b) = s + w(a, b)$
$loop(s) = 0$	$loop(s) = lscore(length(s))$
$skip(xs, s, ys) = s$	$skip(xs, s, ys) = s + \alpha * length(xs) + \beta * length(ys)$
$h([]) = []$	$h([]) = []$
$h(1) = [maximum(1)]$	$h(1) = [maximum(1)]$

Fig. 3. Algebras UNIT (left) and WEIGHT (right)

averaging over all answers. This is why in general, the objective function will return a list of answers. If maximization was the objective, this list would hold the maximum as its only element.

We formulate a signature Π for the palindrome example²:

match: $\mathcal{A} \times Pal \times \mathcal{A} \rightarrow Pal$	skipleft: $\mathcal{A}^* \times Pal \rightarrow Pal$
loop: $\mathcal{A}^* \rightarrow Pal$	skipright: $Pal \times \mathcal{A}^* \rightarrow Pal$
skip: $\mathcal{A}^* \times Pal \times \mathcal{A}^* \rightarrow Pal$	skipl: $\mathcal{A} \times Pal \rightarrow Pal$
	skipr: $Pal \times \mathcal{A} \rightarrow Pal$

We specify two Π -algebras (see Figure 3). The evaluation algebra UNIT uses unit score whereas the algebra WEIGHT uses a weight function ($w(a, b)$) to score a match depending on the corresponding characters. While UNIT ignores the length of the separating loop and the skipped characters, WEIGHT scores skipped prefix and suffix according to their length and some (negative) parameters α and β , and the loop length with a specific function $lscore$. UNIT corresponds to the recurrences in Section 2.1, while WEIGHT solves Exercise 2. For candidate c in Figure 2, we obtain

$$c_{UNIT} = 3$$

$$c_{WEIGHT} = 2w('a', 'a') + w('n', 'n') + lscore(3) + \alpha + \beta.$$

² The four operators on the right side are not used until later.

3.4 Yield Grammars

We obtain an explicit and transparent definition of the search space of a given DP problem by a change of view on tree grammars and parsing:

Definition 3 (Yield grammars and yield languages.) Let \mathcal{G} be a tree grammar over Σ and \mathcal{A} , and y the yield function. The pair (\mathcal{G}, y) is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = y(\mathcal{L}(\mathcal{G}))$. \square

Definition 4 (Yield parsing.) Given a yield grammar (\mathcal{G}, y) over \mathcal{A} and $w \in \mathcal{A}^*$, the yield parsing problem is to find $P_{\mathcal{G}}(w) := \{t \in \mathcal{L}(\mathcal{G}) \mid y(t) = w\}$. \square

The search space spawned by input w is $P_{\mathcal{G}}(w)$. Yield parsing is the computational engine underlying ADP. Its efficient implementation is explained in [14]. For the present development, we assume the availability of a correct and efficient yield parser.

3.5 Algebraic Dynamic Programming and Bellman’s Principle

Given that yield parsing traverses the search space, all that is left to do is evaluate candidates in some algebra and apply the objective function.

Definition 5 (Algebraic dynamic programming.)

- An ADP problem is specified by a signature Σ over \mathcal{A} , a yield grammar (\mathcal{G}, y) over Σ , and a Σ -evaluation algebra I with objective function h_I .
- An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_{\mathcal{G}}(w)$.
- Solving an ADP problem is computing

$$h_I\{t_I \mid t \in P_{\mathcal{G}}(w)\}$$

in polynomial time and space.

\square

There is one essential ingredient missing: efficiency. Since the size of the search space may be exponential in terms of input size, an ADP problem can be solved in polynomial time and space only under a condition known as Bellman’s principle of optimality. In his own words:

“An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.” [2]

We formalize this principle:

Definition 6 (Algebraic version of Bellman’s principle.) For each k -ary operator f in Σ , and all answer lists z_1, \dots, z_k , the objective function h satisfies

$$\begin{aligned} & h([f(x_1, \dots, x_k) \mid x_1 \leftarrow z_1, \dots, x_k \leftarrow z_k]) \\ &= h([f(x_1, \dots, x_k) \mid x_1 \leftarrow h(z_1), \dots, x_k \leftarrow h(z_k)]) \end{aligned}$$

Furthermore, the same property holds for the concatenation of answer lists:

$$h(z_1 ++ z_2) = h(h(z_1) ++ h(z_2))$$

□

The practical meaning of the optimality principle is that we may push the application of the objective function inside the computation of subproblems, thus preventing combinatorial explosion. Phase amalgamation is achieved by a yield parser that uses \mathcal{I} in place of Σ (computing answers instead of trees), and applies $h_{\mathcal{I}}$ to intermediate answer lists. For the sake of efficiency, we shall annotate the tree grammar to indicate the cases where h is to be applied.

Summarizing, we state that an ADP algorithm is completely specified by the annotated yield grammar and the concrete evaluation algebra. Compared to the classic description of DP algorithms via matrix recurrences we have achieved the following:

- An ADP specification is *more abstract* than the traditional recurrences. Separation between search space construction and evaluation is perfect. Tree grammars and evaluation algebras can be combined in a modular way, and the relationships between problem variants can be explained clearly.
- The ADP specification is also *more complete*: DP algorithms in the literature often claim to be parametric with respect to the scoring function, while the initialisation equations are considered part of the search algorithm [8]. In ADP, it becomes clear that initialisation semantically is the evaluation of empty candidates, and it is specified within the algebra.
- Our formalization of Bellman’s principle is *more general* than commonly seen. Objectives like complete enumeration or statistical evaluation of the search space now fall under the framework. If maximization is the objective, our criterion implies Morin’s formalization (strict monotonicity) [22] as a special case.
- The ADP specification is *more reliable*. The absence of subscripts excludes a large class of errors that are traditionally hard to find.

4 The ADP Programming System

We have implemented the ADP approach as a practical program development system. Here, we only sketch a part of the ADP language. Its implementation, described in [14], provides an executable prototype in Haskell, and a translation to efficient C-code. We also plan to provide the translation of an ADP specification into the traditional recurrences³, formatted in L^AT_EX.

³ A concession to traditionalistic attitudes in the bioinformatics community.

4.1 An ASCII Notation for Annotated Yield Grammars

Alike EBNF, productions in yield grammars are written as equations. The operator `<<<` is used to denote the application of a tree constructor to its arguments, which are chained via the `~~~`-operator. Operator `|||` separates multiple right-hand sides of a nonterminal symbol. The operator `...` indicates application of the objective function. Operator priority in decreasing order is `{~~~, <<<, |||, ...}`⁴. Parentheses are used as required for larger trees. The axiom symbol is indicated by the keyword `axiom`, and conditions are attached to productions via the keyword `with`. Using this notation, we write grammar `LBSP0`:

```
grammarLBSP0 = axiom context
context      = skip <<< string~~~ palindrome~~~ string          ... h
palindrome   = match <<< char~~~ palindrome~~~ char            with equal |||
              loop <<< string                                  ... h
```

The annotation `"... h"` declares that the objective function is to be applied to answer lists resulting from the annotated productions. Some further annotation, not shown here, indicates which intermediate results are to be tabulated.

`grammarLBSP0` can be combined with both evaluation algebras of Section 3.3 to solve LBSP under either scoring scheme. But the grammar `grammarLBSP0` allows for multiple representations of the trivial palindrome with empty stem. Such ambiguity is avoided by the following refined grammar `LBSP1`:

```
grammarLBSP1 = axiom start
start        = loop <<< string ||| context                      ... h
context      = skip <<< string~~~ palindrome~~~ string          ... h
palindrome   = match <<< char~~~ pal_inner~~~ char            with equal
pal_inner    = match <<< char~~~ pal_inner~~~ char            with equal |||
              loop <<< string                                  ... h
```

4.2 Efficiency of ADP Programs

From the viewpoint of programming methodology, it is important that asymptotic efficiency can be analyzed and controlled on the abstract level. This property is a major virtue of ADP – it allows to formulate efficiency tuning as grammar and algebra transformations.

If not reduced by the objective function `h`, the number of answers is exponential in terms of input size, and dominates efficiency. Typically however, the objective function is applied to reduce the number of answers wherever necessary. In this case, efficiency is determined by the yield grammar:

Definition 7 (Width of productions and grammar.) Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded by a constant. We define $width(t) = k - 1$. Let π be a production $v \rightarrow t_1 | \dots | t_r$. $width(\pi) = \max\{width(t_1, \dots, t_r)\}$, and $width(\mathcal{G}) = \max\{width(\pi) \mid \pi \text{ production in } \mathcal{G}\}$. \square

⁴ The implementation takes a different view that need not concern us here.

$Ans_{ENUM} = \mathbb{N}$	$Ans_{COUNT} = \mathbb{N}$	$Ans_{EXP} = \mathbb{R}$
match = Match	match(a,s,b) = s	match(a,s,b) = s * p(a,b)
loop = Loop	loop(s) = 1	loop(s) = 1
skip = Skip	skip(xs,s,ys) = s	skip(xs,s,ys) = s
h = id	h([]) = []	h([]) = []
	h([x ₁ , ..., x _r]) = [x ₁ + ... + x _r]	h([x ₁ , ..., x _r]) = [x ₁ + ... + x _r]

Fig. 4. Enumeration algebra (left), counting (middle) and expectation algebra (right)

Theorem 8 Assuming the number of answers is bounded by a constant, the execution time of an ADP algorithm described by tree grammar \mathcal{G} on input w of length n is $O(n^{2+width(\mathcal{G})})$.

Proof: See [14] \square

5 Programming Methodology

We demonstrate the benefits of the algebraic approach to dynamic programming by discussing issues of algorithm design, tuning and testing. Several “tricks” known from the DP literature can now be formulated as general techniques, i.e., in a problem independent fashion.

5.1 Problem Variation, Systematic Testing and Re-use: Multiple Algebras

The fact that a given grammar can be combined with different evaluation algebras has a variety of uses. Beside the two scoring algebras presented in Section 3.3 we formulate an algebra for the enumeration of all answers, an algebra to count the total number of answers, and an algebra to compute the expected number of answers $E_m(n, c)$ in a string of given length n and character composition c (see Figure 4). The function p gives the probability of two characters in the input string to be the same; it is given by a background distribution, or calculated using the character composition of the input string.

Techniques:

Testing: The enumeration algebra allows to inspect the search space of our algorithm, as a means to review design decisions.

Search space combinatorics: The counting algebra is a flexible approach to the combinatorics of structures, supporting mathematical search space analysis.

Significance: The expectation algebra is an elegant way to obtain significance values in string pattern matching.

A deeper study of expectation algebras and their use for significance analysis is found in [19].

5.2 Connecting Related Problems: From Global to Local Pattern Matching

Grammar LBSP describes (local) separated palindromes embedded somewhere in a string. Dropping the first production and making `palindrome` the axiom, we obtain a grammar BSP describing strings that themselves are separated palindromes. This observation allows to formulate a general way to move from global to local pattern matching (or vice versa):

Technique: Global-local transformation

Let \mathcal{G} be a grammar with axiom A_x , describing a pattern language. Adding a new axiom A'_x and the production

$A'_x = \text{skip} \lll \text{string} \sim\sim\sim A_x \sim\sim\sim \text{string}$

yields a grammar for the corresponding local patterns.

For readers familiar with the algorithms used in biosequence analysis: This is, in general terms and for arbitrary sequence analysis problems, the transition from the Needleman-Wunsch [23] to the Smith-Waterman algorithm [25]; however, for an input of form $x\$y^{-1}$ it must be applied at both ends of x and y .

5.3 Controlling Efficiency: Width Reduction

The asymptotic efficiency of an ADP algorithm is $O(n^{2+width(\mathcal{G})})$. Two techniques are available to improve asymptotic efficiency. Both can be described as grammar transformations; they require that a complementary transformation of the evaluation algebra is possible.

Splitting Productions Note that $width(LBSP) = 2$ and hence the runtime is $O(n^4)$ due to the first production. We modify the LBSP grammar such that the recognition of non-matched characters is split in one part for the non-matched prefix (`left_context`) and another for the suffix (`right_context`). In UNIT, we add the definitions `skipleft(xs,s) = s` and `skipright(s,ys) = s`.

This leads to a grammar with width 1 and execution time $O(n^3)$.

```
grammarLBSP2 = axiom start
start        = loop <<< string ||| left_context          ... h
left_context = skipleft <<< string~::~ right_context     ... h
right_context = skipright <<< palindrome~::~ string     ... h
palindrome   = match <<< char~::~ pal_inner~::~ char with equal
pal_inner    = match <<< char~::~ pal_inner~::~ char with equal |||
              loop <<< string                               ... h
```

Technique: Production splitting

The transformation of

$$u = f \lll a \sim\sim\sim b \sim\sim\sim c \dots h \quad \text{to} \quad \begin{aligned} u &= f^1 \lll a \sim\sim\sim u_1 \dots h \\ u_1 &= f^2 \lll b \sim\sim\sim c \dots h \end{aligned}$$

is correct iff:

- (i) $f_I(a, b, c) = f^1_I(a, f^2_I(b, c))$
- (ii) f^1_I and f^2_I satisfy Bellman's principle (Def. 6) with respect to h_I .

Splitting Words Production splitting reduces the grammar to productions containing at most two symbols with unbounded yield. Further improvement can be achieved by replacing terminal symbols with unbounded yield size by terminal symbols with bounded yield size and an extra recursion. In the palindrome example, we modify the tree productions that deal with the skipped prefix and suffix, such that each production contains just one nonterminal of unbounded yield size. Extending UNIT, the operators `skipl` and `skipr` included in the signature Π (Section 3.3) are defined as `skipl(a,s) = s` and `skipr(s,a) = s`.

This gives us an ADP algorithm with $width(\mathcal{G}) = 0$ and hence quadratic execution time.

```

grammarLBSP3 = axiom start
start       = loop <<< string ||| context ... h
context     = skipl <<< char~ context |||
            = skipr <<< context~ char ||| palindrome ... h
palindrome  = match <<< char~ pal_inner~ char with equal
pal_inner   = match <<< char~ pal_inner~ char with equal |||
            = loop <<< string ... h
    
```

Combining grammar LBSP3 with scoring algebra UNIT, this ADP algorithm is equivalent to the recurrences given for *localpalScore* in Section 2.1. Making `pal_inner` the axiom of grammar LBSP3, it is equivalent to the recurrences given for *palScore*.

Technique: Word splitting

The transformation of

$$u = f \lll a_1 \dots a_r \sim\sim\sim v \dots h \quad a_1 \dots a_r \in \mathcal{A}^*$$

into

$$u = f^1 \lll a \sim\sim\sim u \ ||| \ f^2 \lll v \dots h \quad a \in \mathcal{A}$$

is applicable iff

- (i) $f_I(a_1 \dots a_r, v) = f^1_I(a_1, \dots, f^1_I(a_r, f^2_I(v))) \dots$
- (ii) f^1_I and f^2_I satisfy Bellman's principle (Def. 6) with respect to h_I .

A famous application of this technique is the use of affine gap scores in [15]. The scoring function $f(a_1 \dots a_r, u) = r \times \alpha + \beta + u$ is decomposed into separate functions for gap opening $f^2(u) = \beta + u$ and extension $f^1(\alpha, u) = \alpha + u$.

5.4 The Taming of the Near-Optimal Solution Space

In many applications, one is interested not only in an optimal answer, but also in near-optimal answers. The difficulty is that their number grows exponentially even in the vicinity of the optimum. We discuss two techniques to deal with this situation. Without loss of generality we assume in this section that the objective function is minimization.

***k*-Best Answers** An obvious way to compute the *k* best answers is to use an objective function min_k , which keeps the *k* best answers for each subproblem. This only affects the evaluation algebra: replace the definition $\mathbf{h} = \text{min}$ by $\mathbf{h} = \text{min}_k$.

Sorted Lazy Enumeration A much more elegant approach is the following: Assume answers can be computed in the form of sorted lazy lists (streams). Then, we can enumerate a stream of answers in order of optimality, with negligible overhead for the best answers. This solution has been hinted at occasionally [7,17]. Since we have not seen it implemented, we sketch here how this can be achieved.

The key idea is to compute separate lists, i.e. $[[\mathbf{f} \ x \ y \mid y \leftarrow \mathbf{ys}] \mid x \leftarrow \mathbf{xs}]$ instead of $[\mathbf{f} \ x \ y \mid y \leftarrow \mathbf{ys}, x \leftarrow \mathbf{xs}]$. The resulting lists are sorted because Bellman's principle implies strict monotonicity of all functions in the evaluation algebra. The sorted lists are combined using the lazy merge function.

For shortness we give the exact definition in Haskell notation. Let \mathbf{p} and \mathbf{q} be yield parsers, returning sorted lists of answers when applied to a subword (i, j) of the input. Answers may be not just scalar values, but also curried functions. We define their combination as follows:

```
(p ||| q) (i,j) = merge (p(i,j)) (q(i,j))
(p ~~~ q) (i,j) = foldr merge [] [foldr merge [] [[x y | y <- q (k,j)]
                                                    | x <- p (i,k)]
                                                    | k <- [i..j]]
```

Techniques:

k-best enumeration: In the evaluation algebra, use $\mathbf{h} = \text{min}_k$.

sorted lazy enumeration: In the evaluation algebra, use $\mathbf{h} = \text{id}$ and use lazy merge yield parsing implementation.

6 Conclusion

What have we achieved in terms of putting dynamic programming over sequence data on a firm mathematical basis? The algebraic nature of our approach is essential, even where there is no deep algebraic reasoning behind it. (However, counting and statistical evaluation algebras become more intricate when the

modelled search space is not isomorphically embedded in T_Σ .) Casting the candidates of the search space into a representation as an algebraic data type opens the path to separate the description of the search space from its evaluation. This idea is quite general, and it appears that ADP is applicable to all DP problems over strings, including pairwise comparison, and possibly other domains. (We are developing DP algorithms for tree comparison, but this work has not yet been reformulated in ADP style [12].) Correctness arguments can now be given on a convenient level of abstraction – yield languages and evaluation algebras satisfying our new formulation of Bellman’s principle. By virtue of Theorem 8, abstractness does not come at the price of losing efficiency control.

We can give a clear methodical guidance for the development of successful DP recurrences. (Note that Steps 4 – 7 are actually independent.)

1. Design the signature Σ , representing explicitly all cases that might influence evaluation.
2. Design evaluation algebras, at least the enumeration, counting and one scoring algebra, describing the objective of the application.
3. Design the grammar defining the search space.
4. Improve efficiency by grammar and algebra transformation techniques.
5. Experiment with algorithmic ideas using the functional prototype.
6. Test the design via search space enumeration, and plausibility checking via counting algebra.
7. Generate C implementation and validate against the functional prototype.

Returning to our initial quotation of an anonymous referee from the bioinformatics community, we feel that we no longer need luck to find and implement correct DP recurrences. We can express and communicate our experience (and that of others) on an adequate level of abstraction, and devote our talent to tackling ever more ambitious DP problems.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, USA, 1983. 350
2. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957. 349, 353, 354, 356
3. R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Moeller, editor, *State-of-the-Art Seminar on Formal Program Development*. Springer LNCS 755, 1993. 349
4. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969. 353
5. G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, 1988. 350
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990. 350
7. S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U. K., 1997. 349, 362

8. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998. 350, 354, 357
9. D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, 2001. 350
10. R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 16:665–677, 2000. 350, 352, 353
11. R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer Verlag, 2000. 350
12. R. Giegerich, M. Höchsmann, and S. Kurtz. Local Similarity Problems on Trees: A Uniform Model and its Implementation. 2002. (submitted). 363
13. R. Giegerich and K. Schmal. Code selection techniques: Pattern matching, tree parsing and inversion of derivors. In *Proc. European Symposium on Programming 1988*, pages 247–268. Springer LNCS 300, 1988. 353
14. R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative paradigm. In E. A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002. 350, 354, 356, 357, 359
15. O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982. 361
16. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. 350, 351
17. S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System*. Dissertation, Technische Fakultät der Universität Bielefeld, 1995. 362
18. K. Mehlhorn. *Data structures and algorithms*. Springer Verlag, 1984. 350
19. C. Meyer and R. Giegerich. Matching and Significance Evaluation of Combined Sequence-Structure Motifs in RNA. *Z.Phys.Chem.*, 216:193–216, 2002. 359
20. L. Mitten. Composition principles for the synthesis of optimal multi-stage processes. *Operations Research*, 12:610–619, 1964. 349
21. O. de Moor. Dynamic Programming as a Software Component. In M. Mastorakis, editor, *Proceedings of CSCC, July 4-8, Athens*. WSES Press, 1999. 349
22. T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982. 349, 353, 354, 357
23. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970. 360
24. R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1989. 350
25. T. F. Smith and M. S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981. 360
26. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984. 350
27. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res.*, 9(1):133–148, 1981. 352

Analyzing String Buffers in C

Axel Simon and Andy King

Computing Laboratory, University of Kent
Canterbury, CT2 7NF, UK

Abstract. A buffer overrun occurs in a C program when input is read into a buffer whose length exceeds that of the buffer. Overruns often lead to crashes and are a widespread form of security vulnerability. This paper describes an analysis for detecting overruns before deployment which is conservative in the sense that it locates every possible buffer overrun. The paper details the subtle relationship between overrun analysis and pointer analysis and explains how buffers can be modeled with a linear number of variables. As far as we know, the paper gives the first *formal* account of how this software and security problem can be tackled with abstract interpretation, setting it on a firm, mathematical basis.

1 Introduction

Although C programs are ubiquitous, occurring in many security-critical, safety-critical and enterprise-critical applications, they are particularly prone to inexplicable crashes. Many crashes can be traced to buffer overruns [14]. A buffer overrun occurs when input is read into a buffer and the length of the input exceeds the length of the buffer. Buffer overruns typically arise from unbounded string copies using `sprintf()`, `strcpy()` and `strcat()`, as well as loops that manipulate strings without an explicit length check in the loop invariant [15].

Unchecked input which overflows a buffer can overwrite portions of the stack frame. Hackers have exploited this effect to redirect the instruction pointer in the stack frame to malicious code within the string and thereby gain partial or total control of a host [17]. Buffer overruns are a particularly widespread class of security vulnerability [5] and the National Security Agency predicts that overrun attacks will continue to be a problem for at least the next decade [20]. Finding security faults, such as string handling that may overrun, has been likened to the problem of finding a needle in a hay-stack [11]. This paper describes an abstract interpretation scheme that does not attempt to find the needle, but rather aims to prune the hay-stack down so that it can be searched manually.

1.1 Design Space for Overrun Analysis

One (surprising) tradeoff that is applied in program analysis is that of sacrificing soundness for tractability [24]. In the context of detecting overruns, one potential tradeoff in the design-space is that of achieving simplicity by, say ignoring

aliasing, at the cost of possibly missing real errors and possibly reporting spurious errors. Whether this is acceptable depends on whether the analysis aims to detect *most* errors [12,13,23] (debugging) or detect *every* error [8] (verification). One point in this design-space is represented by LCLint. LCLint is an annotation-assisted static checking tool founded on the philosophy that it is fine to “accept a solution that is both unsound and incomplete” [12]. This enables, for example, loops to be analyzed straightforwardly using heuristics. Its annotation mechanism achieves both compositionality and scalability. However, as [12,13] point out, it seems optimistic to believe that a programmer will add annotations to their programs to detect overrun vulnerabilities. Another point in the design space is the constraint based analysis of [23] which can detect potential overruns in unannotated (legacy) C code. This analysis ignores pointer arithmetic and is therefore unsound, but it is fast enough to reason about medium-sized applications without assistance. For speed and simplicity, the analysis is flow insensitive which means that it cannot reason about (non-idempotent) library functions such as `strcat()` which are a frequent source of overruns [15].

The work of Dor, Rodeh and Sagiv [8] is unique in that it attempts to formulate an analysis that is *conservative* in the sense that it never misses a potential overrun. This is a particularly laudable goal in the context of security where there is a history of elite hackers leveraging inconspicuous and innocuous features into major security holes. Our work builds on the foundation of Dor *et al* [8] and is a systematic examination of the abstractions and analyses necessary to reason about overruns in a *truly* conservatively way.

1.2 Conservative Overrun Analysis

The buffer abstraction of Dor *et al* [8] is essentially that of Wagner [23] but specified as a program transformation. Each buffer is abstracted by its size, *alloc*, and the position of the null character, *len*. This two-variable model, however, is not rich enough to accurately track the null position. Suppose, for example, if two pointers *p* and *q* point to different positions within the same buffer, then altering the *len* attribute of *p* (by writing a null character to the buffer) might alter the *len* attribute of *q*. Dor *et al* [8] therefore introduce special *p-overlaps-q* variables to quantify the pointer offset and thereby model string length interaction. This tactic potentially increases the number of variables quadratically which is unfortunate since relational numeric abstractions such as polyhedra [4] become less tractable as the number of variables increase. This is an efficiency issue. More subtle is the correctness issue that relates to pointers that definitely or possibly point to the same buffer. This is illustrated in the following code:

```

1 char *p, *q, s[32], t[32];
2 strcpy(s,"Boat ");          /* s[5] */
3 strcpy(t,"Aero");          /* s[5] t[4] */
4 p = t+4;                   /* p[0] s[5] t[4] */
5 strcat(p, "plane ");       /* p[6] s[5] t[10] */
6 if (rand() q=s; else q=t; /* p[6] q[5,10] s[5] t[10] */
7 strcat(q,"to LaRéunion"); /* p[6,18] q[17,22] s[5,17] t[10,22]*/

```

The comments indicate the possible null positions at the various lines. The `strcat` in line 7 either updates the `s` buffer or the `t` buffer (but not both) depending on the random value. The transformation described by Dor *et al* [8] does not have a mechanism for dealing with possible changes to a buffer – only definite changes are tracked. In particular the transformation does not correctly reflect that the null is either at position 5 or 17 in the `s` buffer. Our remedy to this is to infer that `q` possibly shares with `s` and `t` so that both 5 and 17 are possible null positions for `s` and symmetrically both 10 and 22 are potential null positions for `t`. Thus *possible* sharing information is used as an aid to correctness. Conversely, *definite* sharing information is used to increase precision. It enables a write to a buffer through one pointer to destructively update the null positions for those pointers that definitely share the same buffer. For example, the `strcat` at line 5 changes the null position of the `t` buffer from 4 to 10. The information that `t` and `p` definitely share is needed to infer that the null position in `t` is no longer 4. Moreover, possible sharing is also required to deduce that no other buffers are affected. To summarize, our work makes the following contributions:

- It gives the first formal account of buffer overrun analysis. A buffer abstraction map is proposed that specifies how to model a buffer. This abstraction is then used to formalize the correctness of the analysis. This means that a programmer can be confident that every possible overrun is detected.
- It shows that any string buffer overrun analysis that is both accurate and correct needs to reason about pointer aliasing. Inter-procedural points-to analysis [1,21] is required to ascertain which pointers *possibly* point to the same buffer. The paper also shows how to improve precision by employing a *definite* sharing analysis [9].
- It explains how buffers can be modeled with three variables per pointer and shows how the need for *p_overlaps_q* is finessed by sharing analysis, thereby avoiding quadratic blowup. The paper thus describes a practical foundation for analyzing string buffers.

Section 2 introduces the language String C to formalize operations on string buffers. Section 3 explains how polyhedra and sharing domains can be used to describe buffer properties, and then Section 4 explains how these properties can be tracked by analysis. Sections 5 and 6 and present the related and future work and Section 7 concludes. Proofs are omitted because of lack of space.

2 The String C Language and Its Semantics

The analysis is formulated in terms of a C-like mini language called String C that expresses the essential operations on buffers.

2.1 Abstract Syntax

Let $(n \in) \mathbb{N}$ denote the set of non-negative integers and $(l \in) Lab$ denote a set of labels which mark program points. Let $(x, y \in) X$ and $(f \in) F$ denote the

[skip]	$\rho \vdash \langle \text{skip}, \sigma \rangle \rightarrow \sigma$	
[num]	$\rho \vdash \langle x = n, \sigma \rangle \rightarrow \sigma. [\rho(x) \mapsto n]$	if $n \in \mathbb{N}$
[str]	$\rho \vdash \langle x = "n_1 \dots n_m", \sigma_1 \rangle \rightarrow \sigma_2. [\rho(x) \mapsto \langle a, a, a + m + 1 \rangle]$	if $n_1, \dots, n_m \in \mathbb{N}$ $\wedge [a, a + m + 1] \cap \text{dom}(\sigma_1) = \emptyset$ $\wedge \sigma_2 = \sigma_1. [a + i \mapsto n_{i+1}]_{i=0}^{m-1}.$ $[a + m \mapsto 0]$
[var]	$\rho \vdash \langle x = y, \sigma \rangle \rightarrow \sigma. [\rho(x) \mapsto \sigma. \rho(y)]$	
[add]	$\rho \vdash \langle x = y_1 + y_2, \sigma \rangle \rightarrow \sigma. [\rho(x) \mapsto v]$	if $\sigma. \rho(y_i) = v_i \wedge v_1 \boxplus v_2 = v$
[sub]	$\rho \vdash \langle x = y_1 - y_2, \sigma \rangle \rightarrow \sigma. [\rho(x) \mapsto v]$	if $\sigma. \rho(y_i) = v_i \wedge v_1 \boxminus v_2 = v$
[arr1]	$\rho \vdash \langle x = y_1[y_2], \sigma_1 \rangle \rightarrow \begin{cases} \sigma_2 & \text{if } a_b \leq a < a_e \\ \text{err} & \text{otherwise} \end{cases}$	if $\sigma_1. \rho(y_i) = v_i$ $\wedge v_1 \boxplus v_2 = \langle a_b, a, a_e \rangle$ $\wedge \sigma_2 = \sigma_1. [\rho(x) \mapsto \sigma_1(a)]$
[arr2]	$\rho \vdash \langle y_1[y_2] = x, \sigma_1 \rangle \rightarrow \begin{cases} \sigma_2 & \text{if } a_b \leq a < a_e \\ \text{err} & \text{otherwise} \end{cases}$	if $\sigma_1. \rho(y_i) = v_i \wedge \sigma_1. \rho(x) \in \mathbb{N}$ $\wedge v_1 \boxplus v_2 = \langle a_b, a, a_e \rangle$ $\wedge \sigma_2 = \sigma_1. [a \mapsto \sigma_1. \rho(x)]$
[malloc]	$\rho \vdash \langle x = \text{malloc}(y), \sigma_1 \rangle \rightarrow \sigma_1. \sigma_2. \sigma_3$	if $\sigma. \rho(y) = v$ $\wedge [b, b + v] \cap \text{dom}(\sigma) = \emptyset$ $\wedge \sigma_2 = [b + i \mapsto \text{rand}()]_{i=0}^v$ $\wedge \sigma_3 = [\sigma. \rho(x) \mapsto \langle b, b, b + v \rangle]$

Fig. 1. Concrete semantics for simple statements. The function `rand()` returns random values and models `malloc`'s behavior to allocate uninitialized memory

(finite) sets of variables and function names occurring in a program. F includes the symbol `main` to indicate the program entry point and each function $f \in F$ has an associated variable $f_r \in X$ to return values from functions in Pascal-style.

$$\begin{aligned}
\mathcal{C} ::= & f(x_1, \dots, x_n) \mathcal{L}; \mathcal{C} \mid \varepsilon \\
\mathcal{L} ::= & [\text{skip}]^l \mid [x = n]^l \mid [x = "n_1 \dots n_m"]^l \mid [x = y]^l \mid [x = y_1 + y_2]^l \\
& [x = y_1 - y_2]^l \mid [x = y_1[y_2]]^l \mid [y_1[y_2] = x]^l \mid [x = \text{malloc}(y)]^l \\
& [\text{if } x \text{ then } \mathcal{L}_1 \text{ else } \mathcal{L}_2]^l \mid [\text{while } x \mathcal{L}]^l \mid [\mathcal{L}_1; \mathcal{L}_2]^l \mid [\text{return}]^l \mid [x = f(y_1, \dots, y_n)]^l
\end{aligned}$$

The language $\mathfrak{L}(\mathcal{C})$ defines the set of valid String C programs. Given a fixed program, the map $c : F \rightarrow (\cup_{i \in \mathbb{N}} X^i) \times \mathfrak{L}(\mathcal{L})$ retrieves the formal arguments $\langle x_1, \dots, x_n \rangle$ and body L for a function f . String C does not distinguish between integers and unsigned chars because reasoning about buffers of different sized objects increases the number of cases in the abstract semantics and obscures the underlying ideas. String C can be enriched following the ideas detailed in [1,18].

2.2 Instrumented Operational Semantics

The semantics of String C instruments each pointer with details of its underlying buffer. Specifically, the set of pointer values is defined as $Pnt = \{ \langle a_b, a, a_e \rangle \in \mathbb{N}^3 \mid a_b < a_e \}$. The interpretation of a triple $\langle a_b, a, a_e \rangle$ is that the a_b and a_e record the first location within the buffer and the first location past the end of the buffer. The address a records the current position of the pointer within the

$[\text{if}_1]$	$\rho \vdash \langle \text{if } x \text{ then } L_1 \text{ else } L_2, \sigma \rangle \rightarrow \langle L_1, \sigma \rangle$	if $\sigma.\rho(x) \neq 0$
$[\text{if}_2]$	$\rho \vdash \langle \text{if } x \text{ then } L_1 \text{ else } L_2, \sigma \rangle \rightarrow \langle L_2, \sigma \rangle$	if $\sigma.\rho(x) = 0$
$[\text{while}_1]$	$\rho \vdash \langle \text{while } x \text{ } L, \sigma \rangle \rightarrow \langle L; \text{while } x \text{ } L, \sigma \rangle$	if $\sigma.\rho(x) \neq 0$
$[\text{while}_2]$	$\rho \vdash \langle \text{while } x \text{ } L, \sigma \rangle \rightarrow \sigma$	if $\sigma.\rho(x) = 0$
$[\text{seq}_1]$	$\frac{\rho \vdash \langle L_1, \sigma_1 \rangle \rightarrow \langle L_2, \sigma_2 \rangle}{\rho \vdash \langle L_1; L_3, \sigma_1 \rangle \rightarrow \langle L_2; L_3, \sigma_2 \rangle}$	
$[\text{seq}_2]$	$\frac{\rho \vdash \langle L_1, \sigma_1 \rangle \rightarrow \sigma_2}{\rho \vdash \langle L_1; L_2, \sigma_1 \rangle \rightarrow \langle L_2, \sigma_2 \rangle}$	
$[\text{env}_1]$	$\frac{\rho_2 \vdash \langle L_1, \sigma_1 \rangle \rightarrow \langle L_2, \sigma_2 \rangle}{\rho_1 \vdash \langle \text{env } \rho_2 \text{ in } L_1, \sigma_1 \rangle \rightarrow \langle \text{env } \rho_2 \text{ in } L_2, \sigma_1 \rangle}$	
$[\text{env}_2]$	$\frac{\rho_2 \vdash \langle L, \sigma_1 \rangle \rightarrow \sigma_2}{\rho_1 \vdash \langle \text{env } \rho_2 \text{ in } L, \sigma_1 \rangle \rightarrow \sigma_2}$	
$[\text{ret}]$	$\rho \vdash \langle \text{return}; L, \sigma \rangle \rightarrow \sigma$	
$[\text{call}]$	$\rho_1 \vdash \langle x = f(y_1, \dots, y_n), \sigma_1 \rangle \rightarrow \langle \text{env } \rho_1.\rho_2.[f_r \mapsto \rho(x)] \text{ in } L, \sigma_2.\sigma_1 \rangle$	if $c(f) = \langle z_1, \dots, z_n, L \rangle$ $\wedge \rho_2 = [z_i \mapsto a_i]_{i=1}^n$ $\wedge \text{dom}(\sigma_1) \cap \text{rng}(\rho_2) = \emptyset$ $\wedge \sigma_2 = [a_i \mapsto \sigma_1.\rho_1(y_i)]_{i=1}^n$

Fig. 2. Concrete semantics for control statements

buffer. For a valid buffer access $a_b \leq a < a_e$ must hold. The set of values is then defined as $Val = \mathbb{N} \cup Pnt$ whereas the set of environment and store maps are defined $Env = X \rightarrow \mathbb{N}$ and $Str = \mathbb{N} \rightarrow Val$ respectively. The following (partial) maps formalize address arithmetic.

Definition 1. The functions $\boxplus, \boxminus : Val^2 \rightarrow Val$ are defined as follows:

$$\begin{array}{ll}
 i \boxplus j & = i + j & i \boxminus j & = i - j \\
 i \boxplus \langle b_b, b, b_e \rangle & = \langle b_b, b + i, b_e \rangle & i \boxminus \langle b_b, b, b_e \rangle & = \perp \\
 \langle a_b, a, a_e \rangle \boxplus j & = \langle a_b, a + j, a_e \rangle & \langle a_b, a, a_e \rangle \boxminus j & = \langle a_b, a - j, a_e \rangle \\
 \langle a_b, a, a_e \rangle \boxplus \langle b_b, b, b_e \rangle & = \perp & \langle a_b, a, a_e \rangle \boxminus \langle b_b, b, b_e \rangle & = a - b
 \end{array}$$

Figures 1 and 2 detail the operational semantics for simple statements and the control statements. In Figure 2, $\text{dom}(f)$ and $\text{rng}(f)$ denote the domain and range of a mapping f and \cdot denotes function composition defined such that $g.f(x) = g(f(x))$. The env statement introduced in Figure 2 models scoping.

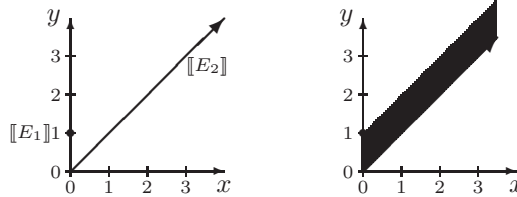
3 Abstract Domains

3.1 Linear Inequality (Polyhedral) Domain

Let Y denote the variables $\{y_1, \dots, y_n\}$, let Lin_Y denote the set of (possibly rearranged) linear equalities $\sum_{i=1}^n m_i y_i = m$ and (possibly rearranged) non-strict inequalities $\sum_{i=1}^n m_i y_i \leq m$ and $\sum_{i=1}^n m_i y_i \geq m$ where $m, m_i \in \mathbb{Z}$. Let Eqn_Y denote all finite subsets of Lin_Y . Note that although each set $E \in Eqn_Y$ is finite, Eqn_Y is not finite. Define $\llbracket e \rrbracket = \{ \langle x_1, \dots, x_n \rangle \in \mathbb{R}^n \mid \sum_{i=1}^n m_i x_i \odot m \}$

where $\odot \in \{\leq, =, \geq\}$. Then define $\llbracket E \rrbracket$ to be the polyhedron $\llbracket E \rrbracket = \cap\{\llbracket e \rrbracket \mid e \in E\}$ if $E \in Eqn_Y$. Eqn_Y is ordered by entailment, that is, $E_1 \models E_2$ iff $\llbracket E_1 \rrbracket \subseteq \llbracket E_2 \rrbracket$. Equivalence on Eqn_Y is defined $E_1 \equiv E_2$ iff $E_1 \models E_2$ and $E_2 \models E_1$. Let $Poly_Y = Eqn_Y / \equiv$. $Poly_Y$ inherits entailment \models from Eqn_Y . In fact $\langle Poly_Y, \models, \sqcap, \sqcup \rangle$ is a lattice (rather than a complete lattice) with $\llbracket E_1 \rrbracket \sqcap \llbracket E_2 \rrbracket = \llbracket E_1 \cup E_2 \rrbracket$ and $\llbracket E_1 \rrbracket \sqcup \llbracket E_2 \rrbracket = \llbracket E \rrbracket$ where $\llbracket E \rrbracket = cl(conv(\llbracket E_1 \rrbracket \cup \llbracket E_2 \rrbracket))$ and $cl(S)$ and $conv(S)$ denote the closure and convex hull of a set $S \in \mathbb{R}^n$ respectively [19]. Note that in general $conv(\llbracket E_1 \rrbracket \cup \llbracket E_2 \rrbracket)$ is not closed and therefore cannot be described by a system of non-strict linear inequalities as is illustrated below.

Example 1. Let $Y = \{x, y\}$, $E_1 = \{x = 0, y = 1\}$ and $E_2 = \{0 \leq x, x - y = 0\}$ so that $\llbracket E_1 \rrbracket = \{(0, 1)\}$ and $\llbracket E_2 \rrbracket = \{\langle x, y \rangle \mid 0 \leq x \wedge x = y\}$. These polyhedra are illustrated in the left-hand graph. Then $conv(\llbracket E_1 \rrbracket \cup \llbracket E_2 \rrbracket)$ includes the point $\langle 0, 1 \rangle$ but not the ray $\{\langle x, y \rangle \mid 0 \leq x \wedge x + 1 = y\}$ and hence is not closed. This convex space is depicted in the right-hand graph.



It is useful to augment \sqcap and \sqcup with three operations: projection, minimization and maximization. The minimization and maximization operations are respectively defined $\min(\langle m_1, \dots, m_n \rangle, [E]_{\equiv}) = \min\{\sum_{i=1}^n m_i x_i \mid \langle x_1, \dots, x_n \rangle \in [E]_{\equiv}\}$ and $\max(\langle m_1, \dots, m_n \rangle, [E]_{\equiv})$ is defined analogously. The vector $\langle m_1, \dots, m_n \rangle$ is written as $\sum_{i=1}^n m_i y_i$ for brevity. Note that $\min(\sum_{i=1}^n m_i y_i, [E]_{\equiv}) \in \mathbb{R} \cup \{-\infty\}$ whereas $\max(\sum_{i=1}^n m_i y_i, [E]_{\equiv}) \in \mathbb{R} \cup \{+\infty\}$. Projection is defined $\exists_{x_i}([E]_{\equiv}) = [E']_{\equiv}$ where $[E']_{\equiv} = \{\langle x_1, \dots, x_{i-1}, x, x_{i+1}, \dots, x_n \rangle \mid x \in \mathbb{R} \wedge \langle x_1, \dots, x_n \rangle \in [E]_{\equiv}\}$. For brevity, let $\exists_{y_{i_1}, \dots, y_{i_m}}([E]_{\equiv}) = \exists_{y_{i_m}}(\dots \exists_{y_{i_1}}([E]_{\equiv}) \dots)$. The variable set for $[E]_{\equiv}$ is defined $\text{var}([E]_{\equiv}) = \cap\{\text{var}(E') \mid E \equiv E'\}$ where $\text{var}(E')$ is the set of variables (syntactically) occurring in E' . Let $false = \{0 = 1\}_{\equiv}$ so that $\llbracket false \rrbracket = \emptyset$.

Projection can be computed using the Fourier algorithm [3], \min and \max using Simplex [3], \sqcap is straightforward to compute whereas \sqcup can either be calculated using the point, ray and line representation [4] or by using constraint relaxation techniques [7]. Let $s = \sum_{i=1}^n m_i y_i$. Then entailment can be tested by: $P \models \{s < m\}$ iff $\max(s, P) < m$ (even though $\{s < m\}$ is strict); $P \models \{s \leq m\}$ iff $\max(s, P) \leq m$; $P \models \{s = m\}$ iff $P \models \{s \leq m\}$ and $P \models \{s \geq m\}$. Moreover, $P \models \{e_1, \dots, e_k\}$ iff $P \models \{e_1\} \dots P \models \{e_k\}$. Henceforth, whenever unambiguous, $[E]_{\equiv}$ will be written as E for brevity. Finally note that $Poly_Y$ does not satisfy the ascending chain condition, that is, chains $P_1 \models P_2 \models \dots$ exist for which $\sqcup_{i>0} P_i$ does not exist.

Example 2. Let P_1 be an equilateral triangle, P_2 a hexagon, P_3 a dodecagon and so forth such that the vertices of P_i are contained within those of P_{i+1} . Then $P_1 \models P_2 \models P_3 \dots$ is an ascending chain which converges onto a circle which

itself is not a polyhedron. Thus widening is required to enforce termination in polyhedral fix-point calculations [4].

3.2 Polyhedral Buffer Domain

Let X_n , X_o and X_s denote sets of variables such that $x \in X$ iff $x_n \in X_n$, $x_o \in X_o$ and $x_s \in X_s$ and suppose that X , X_n , X_o and X_s are pair-wise disjoint. Let $PB_X = Poly_{X \cup X_n \cup X_o \cup X_s}$. The lattice $\langle PB_X, \models, \sqcap, \sqcup \rangle$ is used to describe numeric buffer properties salient to overrun analysis. In particular, suppose an environment ρ maps the variable $x \in X$ to the address $\rho(x) = b$ and a store σ maps b to a pointer triple $\sigma(b) = \langle a_b, a, a_e \rangle$. Then the variable $x_s \in X_s$ describes the number of locations between a_b and a_e (the size of the underlying buffer); $x_o \in X_o$ represents the offset of a relative to a_b (the position of x within the buffer); and $x_n \in X_n$ captures the first location between a_b and a_e whose contents is zero (the position of the null when the buffer is interpreted as a string). If $P \in PB_X$ describes the pair $\langle \rho, \sigma \rangle$, then P captures the numeric relationships between x_s , x_o and x_n . This idea is formalized below.

Definition 2. The polyhedral concretization map $\gamma_X^{PB} : PB_X \rightarrow \wp(Env \times Str)$ is defined $\gamma_X^{PB}(P) = \{ \langle \rho, \sigma \rangle \mid \alpha_X^{sc}(\langle \rho, \sigma \rangle) \sqcap \alpha_X^{bf}(\langle \rho, \sigma \rangle) \models P \}$ where

$$\alpha_X^{sc}(\langle \rho, \sigma \rangle) = \{ x = \sigma.\rho(x) \mid x \in X \wedge \sigma.\rho(x) \in \mathbb{N} \}$$

$$\alpha_X^{bf}(\langle \rho, \sigma \rangle) = \left\{ \begin{array}{l} x_o = a - a_b, \\ x_s = a_e - a_b, \\ x_n = b - a_b \end{array} \middle| \begin{array}{l} x \in X \wedge \sigma.\rho(x) = \langle a_b, a, a_e \rangle \wedge \\ b = \min(\{a_e\} \cup \{n \in [a_b, a_e - 1] \mid \sigma(n) = 0\}) \end{array} \right\}$$

If zero does not occur within the buffer then $\{n \in \mathbb{N} \mid a_b \leq n < a_e \wedge \sigma(n) = 0\} = \emptyset$ so the $\{a_e\}$ element is used to ensure the map is well-defined. It also records the definite absence of a zero (and thereby enables certain definite errors to be found). An abstraction map $\alpha_X^{PB} : \wp(Env \times Str) \rightarrow PB_X$ cannot be synthesized from γ_X^{PB} since PB_X is not complete. In particular, $\sqcup \{ \alpha_X^{sc}(\langle \rho, \sigma \rangle) \sqcap \alpha_X^{bf}(\langle \rho, \sigma \rangle) \mid \langle \rho, \sigma \rangle \in M \}$ is not well defined for arbitrary $M \in \wp(Env \times Str)$. To put it another way, M does not necessarily have a best polyhedral abstraction.

Tracking the first zero (rather than, say, every zero) keeps the number of variables in the model low which aids simplicity and computational efficiency. However, there are unusual combinations of string operations where just tracing the first null can lead to a loss of precision and hence false warnings.

Example 3. Consider the following (synthetic) C program that zeros the buffer located by `s`, then and writes the buffer character by character.

```
char *s = malloc(10), t[4];
memset(s, 0, 10);
s[0]='0'; s[1]='k';
strcpy(t, s);
```

The call to `memset` will set $s_n = 0$, i.e. the first null position is 0. After the first write the analysis can only deduce $s_n \geq 1$ since the first null (if it exists) must occur to the right of the '0'. Likewise after the second write the analysis infers $s_n \geq 2$. The write to the 4 character buffer `t` is safe if $s_n \leq 3$. However $s_n \geq 2$ does not imply $s_n \leq 3$ and therefore the call to `strcpy` generates a spurious warning. Inserting `s[2]='a'; s[3]='y'` in front of the call to `strcpy` will yield a definite error since $s_n \geq 4$ implies that $s_n \leq 3$ cannot be satisfied. On the other hand, writing the same four characters to the same positions in reverse order only leads to a warning: $s_n = 0$ holds valid until `s[0]` is written which then updates the null position to $s_n \geq 1$.

3.3 Possible and Definite Buffer Sharing Domains

Let $PS_X = \wp(X^2)$ and $DS_X = \wp(X^2)$. The complete lattices $\langle PS_X, \subseteq, \cap, \cup \rangle$ and $\langle DS_X, \supseteq, \cup, \cap \rangle$ are used to express (pair-wise) possible buffer sharing and definite buffer sharing respectively.

Definition 3. The abstraction maps $\alpha_X^{PS} : \wp(Env \times Str) \rightarrow PS_X$ and $\alpha_X^{DS} : \wp(Env \times Str) \rightarrow DS_X$ and concretization maps $\gamma_X^{PS} : PS_X \rightarrow \wp(Env \times Str)$ and $\gamma_X^{DS} : DS_X \rightarrow \wp(Env \times Str)$ are defined as follows:

$$\begin{aligned} \alpha_X^{PS}(M) &= \cup \{ \alpha_X^{sh}(\langle \rho, \sigma \rangle) \mid \langle \rho, \sigma \rangle \in M \} & \alpha_X^{DS}(M) &= \cap \{ \alpha_X^{sh}(\langle \rho, \sigma \rangle) \mid \langle \rho, \sigma \rangle \in M \} \\ \gamma_X^{PS}(S) &= \{ \langle \rho, \sigma \rangle \mid \alpha_X^{sh}(\langle \rho, \sigma \rangle) \subseteq S \} & \gamma_X^{DS}(D) &= \{ \langle \rho, \sigma \rangle \mid D \subseteq \alpha_X^{sh}(\langle \rho, \sigma \rangle) \} \end{aligned}$$

where $\alpha_X^{sh}(\langle \rho, \sigma \rangle) = \{ \langle x, y \rangle \in X^2 \mid \sigma(\rho(x)) = \langle a_b, a_x, a_e \rangle \wedge \sigma(\rho(y)) = \langle a_b, a_y, a_e \rangle \}$.

The intuition is that each $D \in DS_X$ captures the certain presence of sharing in that if $\langle x, y \rangle \in D$ then $\langle x, y \rangle \in \alpha_X^{sh}(\langle \rho, \sigma \rangle)$ for all $\langle \sigma, \rho \rangle \in \gamma_X^{DS}(D)$. Conversely, each $S \in PS_X$ describes the certain lack of sharing, that is, if $\langle x, y \rangle \notin S$ then $\langle x, y \rangle \notin \alpha_X^{sh}(\langle \rho, \sigma \rangle)$ for all $\langle \sigma, \rho \rangle \in \gamma_X^{PS}(S)$. This is the negative interpretation of S . The positive interpretation of S is that S describes the possible present of sharing, that is, if $\langle x, y \rangle \in S$ then there exists $\langle \sigma, \rho \rangle \in \gamma_X^{PS}(S)$ such that $\langle x, y \rangle \in \alpha_X^{sh}(\langle \rho, \sigma \rangle)$.

Proposition 1. α_X^{PS} and γ_X^{PS} are monotonic; $\alpha_X^{PS}(\gamma_X^{PS}(S)) \subseteq S$ for all $S \in PS_X$; and $M \subseteq \gamma_X^{PS}(\alpha_X^{PS}(M))$ for all $M \in \wp(Env \times Str)$.

Since $\wp(Env \times Str)$ and PS_X are complete lattices, it follows that the quadruple $\langle \wp(Env \times Str), \gamma_X^{PS}, PS_X, \alpha_X^{PS} \rangle$ is a Galois connection between $\wp(Env \times Str)$ and PS_X . Likewise $\langle \wp(Env \times Str), \gamma_X^{DS}, DS_X, \alpha_X^{DS} \rangle$ is also a Galois connection.

The overrun analysis presented in this paper pre-supposes possible and definite buffer sharing information. The inter-procedural flow-insensitive points-to analysis of [21] can be adapted to derive the former whereas intra-procedural analysis is likely to be sufficient for the latter. The sharing abstractions S^l and D^l are introduced to abstract away from a particular sharing analysis.

Definition 4. S^l and D^l are defined so that $\langle \rho, \sigma_2 \rangle \in \gamma_X^{PS}(S^l) \cap \gamma_X^{DS}(D^l)$ if $\rho \vdash \langle x = \text{main}(), \sigma_1 \rangle \rightarrow^* \langle L^l, \sigma_2 \rangle$ where $\rho = [x \mapsto 0]$, $\sigma_1 = [0 \mapsto n]$ and $n \in \mathbb{N}$.

3.4 Domain Interaction

A polyhedral abstraction can be used to refine a possible sharing abstraction whereas a definite sharing abstraction can improve a polyhedral abstraction.

Definition 5. The operators $\varrho_D : PB_X \rightarrow PB_X$ and $\varrho_P : PS_X \rightarrow PS_X$ are defined $\varrho_D(P) = P \sqcap \{x_n = y_n, x_s = y_s \mid \langle x, y \rangle \in D\}$ and

$$\varrho_P(S) = S \setminus \left\{ \langle x, y \rangle \in S \mid \begin{array}{l} P \models x_n < y_n \vee P \models y_n < x_n \vee \\ P \models x_s < y_s \vee P \models y_s < x_s \end{array} \right\}$$

Proposition 2. $\varrho_D(P) \models P$, $\gamma_X^{DS}(D) \cap \gamma_X^{PB}(P) = \gamma_X^{DS}(D) \cap \gamma_X^{PB}(\varrho_D(P))$, $\varrho_P(S) \subseteq S$ and $\gamma_X^{PS}(S) \cap \gamma_X^{PB}(P) = \gamma_X^{PS}(\varrho_P(S)) \cap \gamma_X^{PB}(P)$.

4 Abstract Semantics for String C

This section presents an analysis for detecting string buffer overruns; it is not intended to verify that a program conforms to the ANSI C standard – it is simply designed to alert the programmer to potential overruns.

4.1 Assignment

The starting point for the construction is a polyhedral operator for assignment.

Definition 6. Let $\odot \in \{\leq, =, \geq\}$, $s = \sum_{j=1}^n m_j y_j$ and $t \notin \text{var}(P \sqcap \{x \odot s\})$. Then destructive update, \triangleright , and additive update, \trianglerighteq , are respectively defined:

$$P \triangleright x \odot s = \exists_t (\exists_x (P \sqcap \{t \odot s\}) \sqcap \{x = t\}) \quad P \trianglerighteq x \odot s = P \sqcup (P \triangleright x \odot s)$$

If x does not occur in s , that is $x \notin \text{var}(\{t \odot s\})$, then $\exists_x (P \sqcap \{t \odot s\}) = \exists_x (P) \sqcap \{t \odot s\}$ and hence $P \triangleright x \odot s = \exists_x (P) \sqcap \{x \odot s\}$. This version of update requires few operations, is more efficient, and also suggests that $P \triangleright x \odot s$ can be used to simulate destructive update. In fact this is precisely the rôle of the \triangleright operator. The \trianglerighteq operator, additive update, is used to model a destructive update that may or may not have been applied. Safety follows because in the former case $P \triangleright x \odot s \models P \trianglerighteq x \odot s$ whereas in the latter case $P \models P \trianglerighteq x \odot s$.

Example 4. Let $P = \{x = z, x \leq y + 1\}$ and consider $P \triangleright \{x = x + 1\}$. Observe that $t \notin \{x, y, z\} = \text{var}(P \sqcap \{x = x + 1\})$. Since $\exists_x (\{x = z, x \leq y + 1, t = x + 1\}) = \{t = z + 1, t \leq y + 2\}$ and $\exists_t (\{t = z + 1, t \leq y + 2, x = t\}) = \{x = z + 1, x \leq y + 2\}$ it follows that $P \triangleright \{x = x + 1\} = \{x = z + 1, x \leq y + 2\}$. Moreover, $P \trianglerighteq \{x = x + 1\} = P \sqcup \{x = z + 1, x \leq y + 2\} = \{z \leq x \leq z + 1, x \leq y + 2\}$.

It is not unusual for a concrete operation to modify several attributes of a polyhedra together and thus it is useful to introduce a concept of parallel update. In particular, let $i \in \{1, \dots, k\}$ and consider $e_i = \{x_i \odot_i s_i\}$ where $\odot_i \in \{\leq, =, \geq\}$ and $s_i = m_i + \sum_{j=1}^{n_i} m_{i,j} y_{i,j}$. Define $P \triangleright \langle e_1, \dots, e_k \rangle = ((P \triangleright e_1) \dots \triangleright e_k)$ and likewise $P \trianglerighteq \langle e_1, \dots, e_k \rangle = ((P \trianglerighteq e_1) \dots \trianglerighteq e_k)$. The following proposition explains how $P \triangleright \langle e_1, \dots, e_k \rangle$ and $P \trianglerighteq \langle e_1, \dots, e_k \rangle$ are independent of the evaluation order.

[skip']	$\langle \text{skip}, P \rangle \rightarrow' P$
[num']	$\langle x = n, P \rangle \rightarrow' \exists_{x_n, x_o, x_s}(P) \triangleright \{x = n\}$ if $n \in \mathbb{N}$
[str']	$\langle x = "n_1 \dots n_m", P \rangle \rightarrow' \exists_x(P) \triangleright \{x_o = 0, x_s = m + 1, x_n = m\}$ if $n_i \in \mathbb{N}$
[var']	$\langle x = y, P \rangle \rightarrow' \begin{cases} \exists_{x_n, x_o, x_s}(P) \triangleright \{x = y\} & \text{if } y \in \text{sc}(P) \\ \exists_x(P) \triangleright \{x_n = y_n, x_o = y_o, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \\ \exists_{x, x_n, x_o, x_s}(P) & \text{otherwise} \end{cases}$
[add']	$\langle x = y + z, P \rangle \rightarrow' \begin{cases} \exists_{x_n, x_o, x_s}(P) \triangleright \{x = y + z\} & \text{if } y, z \in \text{sc}(P) \\ \exists_x(P) \triangleright \{x_n = z_n, x_o = y + z_o, x_s = z_s\} & \text{else if } y \in \text{sc}(P) \wedge z \in \text{bf}(P) \\ \exists_x(P) \triangleright \{x_n = y_n, x_o = y_o + z, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \wedge z \in \text{sc}(P) \\ \exists_{x, x_n, x_o, x_s}(P) & \text{otherwise} \end{cases}$
[sub']	$\langle x = y - z, P \rangle \rightarrow' \begin{cases} \exists_{x_n, x_o, x_s}(P) \triangleright \{x = y - z\} & \text{if } y, z \in \text{sc}(P) \\ \exists_x(P) \triangleright \{x_n = y_n, x_o = y_o - z, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \wedge z \in \text{sc}(P) \\ \exists_{x_n, x_o, x_s}(P) \triangleright \{x = y_o - z_o\} & \text{else if } y, z \in \text{bf}(P) \\ \exists_{x, x_n, x_o, x_s}(P) & \text{otherwise} \end{cases}$
[arr1']	$\langle x = y[z], P \rangle \rightarrow' \begin{cases} \text{err} & \text{if } P \models \{y_o + z < 0\} \\ \text{err} & \text{else if } P \models \{y_o + z \geq y_s\} \\ \text{warn} & \text{else if } P \not\models \{0 \leq y_o + z < y_s\} \\ \exists_{x_n, x_o, x_s}(P) \triangleright \{x = 0\} & \text{else if } P \models \{y_o + z = y_n\} \\ \exists_{x_n, x_o, x_s}(P) \triangleright \{x \geq 1\} & \text{else if } P \models \{y_o + z < y_n\} \\ \exists_{x, x_n, x_o, x_s}(P) & \text{otherwise} \end{cases}$
[arr2']	$\langle [y[z] = x]^l, P \rangle \rightarrow' \begin{cases} \text{err} & \text{if } P \models \{y_o + z < 0\} \\ \text{err} & \text{else if } P \models \{y_o + z \geq y_s\} \\ \text{warn} & \text{else if } P \not\models \{0 \leq y_o + z < y_s\} \\ P'' \triangleright \{v_i = y_n \mid v_i \in V\} & \text{otherwise} \end{cases}$
[malloc']	$\langle x = \text{malloc}(y), P \rangle \rightarrow' \exists_x(P) \triangleright \{x_n \geq 0, x_o = 0, x_s = y\}$

Fig. 3. Abstract semantics for simple statements where $W = \{w_n \mid \langle w, y \rangle \in D^l\}$, $P' = \varrho_{D^l}(P)$, $V = \{v_n \mid \langle v, y \rangle \in \varrho_{P'}(S^l)\} \setminus W$ and $P'' = \varrho_{D^l}(\text{update}_{x,y,z}(\exists_{W \setminus \{y_n\}}(P')))$

Proposition 3. Let $\pi : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$ be a permutation, $e_i = \{x_i \odot_i s_i\}$, $s_i = m_i + \sum_{j=1}^{m_i} m_{i,j} y_{i,j}$ and $x_i \notin \text{var}(e_j)$ for all $i \neq j$. Then $P \triangleright \langle e_1, \dots, e_k \rangle = P \triangleright \langle e_{\pi(1)}, \dots, e_{\pi(k)} \rangle$ and $P \triangleright \langle e_1, \dots, e_k \rangle = P \triangleright \langle e_{\pi(1)}, \dots, e_{\pi(k)} \rangle$.

For brevity, define $P \triangleright \{e_1, \dots, e_k\} = P \triangleright \langle e_1, \dots, e_k \rangle$ if $x_i \notin \text{var}(e_j)$ for all $i \neq j$. Proposition 3 ensures that $P \triangleright \{e_1, \dots, e_k\}$ is well-defined whereas Proposition 4 explains how \triangleright can be used to approximate \triangleright when a polyhedra can be updated with different combinations of equations (and possibly not at all).

Proposition 4. $P \triangleright E' \models P \triangleright E$ for all $E' \subseteq E$.

4.2 Non-array Statements

To construct abstract versions of the non-array statements, the sc and bf maps are introduced to detect whether an object is a scalar or a pointer. The following

proposition states an invariant of any correct analysis: a polyhedra cannot simultaneously record scalar and buffer attributes for a given variable. The lemma then asserts correctness for the abstract semantics of the non-array statements.

Definition 7. The maps $sc : PB_X \rightarrow \wp(X)$ and $bf : PB_X \rightarrow \wp(X)$ are defined $sc(P) = X \cap \text{var}(P)$ and $bf(P) = \{x \in X \mid \{x_n, x_o, x_s\} \cap \text{var}(P) \neq \emptyset\}$.

Proposition 5. If $sc(P) \cap bf(P) \neq \emptyset$ then $\gamma_X^{PB}(P) = \emptyset$.

Lemma 1. Suppose $L = [\text{skip}]^l \mid \dots \mid [x = y - z]^l$. Then if $\rho \vdash \langle L, \sigma_1 \rangle \rightarrow \sigma_2$, $\langle \rho, \sigma_1 \rangle \in \gamma_X^{PB}(P)$ and $\langle L, P_1 \rangle \rightarrow' P_2$ it follows that $\langle \rho, \sigma_2 \rangle \in \gamma_X^{PB}(P_2)$.

4.3 Array Statements

To reason about array statements that can error, the store is extended to $EStr = Str \cup \{\text{err}\}$ where err denotes an error state. Likewise, the polyhedral domain is extended to $EPB_X = PB_X \cup \{\perp, \text{err}, \text{warn}\}$. $\langle EPB_X, \preceq, \wedge, \vee \rangle$ is a lattice where the ordering \preceq is defined by $\perp \preceq P$ and $P \preceq \text{warn}$ for all $P \in EPB_X$ whereas for all $P_i \in PB_X$, $P_1 \preceq P_2$ iff $P_1 \models P_2$. Moreover, \wedge and \vee are given by:

$$P_1 \wedge P_2 = \begin{cases} P_1 \sqcap P_2 & \text{if } P_1, P_2 \in PB_X \\ \text{err} & \text{else if } P_1 = P_2 = \text{err} \\ P_1 & \text{else if } P_2 = \text{warn} \\ P_2 & \text{else if } P_1 = \text{warn} \\ \perp & \text{otherwise} \end{cases} \quad P_1 \vee P_2 = \begin{cases} P_1 \sqcup P_2 & \text{if } P_1, P_2 \in PB_X \\ \text{err} & \text{else if } P_1 = P_2 = \text{err} \\ P_1 & \text{else if } P_2 = \perp \\ P_2 & \text{else if } P_1 = \perp \\ \text{warn} & \text{otherwise} \end{cases}$$

The following (extended) concretization map explains how the abstract objects err and warn represent definite errors and possible errors in the concrete setting.

Definition 8. The concretization map $\gamma_X^{EPB} : EPB_X \rightarrow \wp(\text{Env} \times EStr)$ is defined: $\gamma_X^{EPB}(\perp) = \emptyset$, $\gamma_X^{EPB}(\text{err}) = \text{Env} \times \{\text{err}\}$, $\gamma_X^{EPB}(\text{warn}) = \text{Env} \times EStr$ and $\gamma_X^{EPB}(P) = \gamma_X^{PB}(P)$ if $P \in PB_X$.

The abstract semantics for array read generates an error (warning) if the read is definitely (possibly) outside the buffer. Array write is more subtle since a write action through one pointer can effect the null position attribute of all the pointers directed at the same buffer. The following operator details the way in which the null attribute is effected by the write.

Definition 9. The operator $\text{update}_{x,y,z} : PB_X \rightarrow PB_X$ is defined:

$$\text{update}_{x,y,z}(P) = \begin{cases} P_A & \text{if } P \models \{y_o + z > y_n\} \\ P_A & \text{else if } P \models \{x > 0, y_o + z < y_n\} \\ P_B & \text{else if } P \models \{x = 0, y_o + z < y_n\} \\ P_C & \text{else if } P \models \{y_o + z < y_n\} \\ P_D & \text{else if } P \models \{x > 0\} \wedge w_u = n_l = w_l \\ P_B & \text{else if } P \models \{x = 0\} \wedge w_u = n_l = w_l \\ P_E & \text{else if } P \models \{x > 0\} \\ P_A & \text{else if } P \models \{x = 0\} \wedge n_l \leq w_l \leq n_u < w_u \\ P_F & \text{else if } P \models \{x = 0\} \\ P_G & \text{otherwise} \end{cases} \quad \begin{cases} P_A = P \\ P_B = P \triangleright (y_n = y_o + z) \\ P_C = P \triangleright (y_n = y_o + z) \\ P_D = P \triangleright (y_n \geq y_o + z + 1) \\ P_E = P \triangleright (y_n \geq y_n) \\ P_F = P_C \sqcap \{y_n \leq \min\{n_u, w_u\}\} \\ P_G = P \triangleright (y_n \geq y_o + z) \\ w_l = \lceil \min(y_o + z, P) \rceil \\ w_u = \lfloor \max(y_o + z, P) \rfloor \\ n_l = \lceil \min(y_n, P) \rceil \\ n_u = \lfloor \max(y_n, P) \rfloor \end{cases}$$

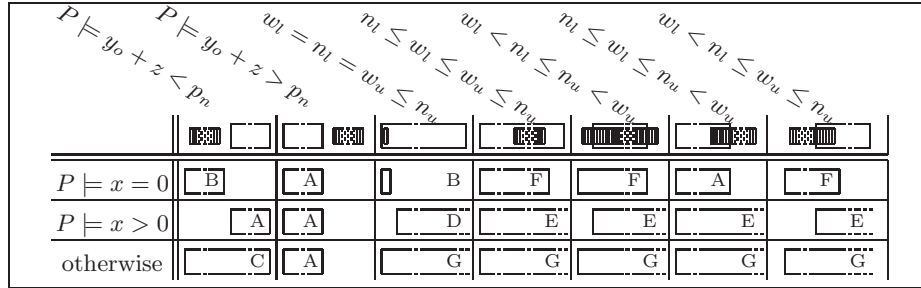




Fig. 4. Graphical representation of the update function

The operator reduces to a case analysis of the write position relative to the null position. The various cases are depicted in Figure 4 in which the striped bars  (hollow bars ) represent a range that includes the write (first null) position. In case *A* the null position is not altered. In *B*, the null position is refined to coincide with the write position. In *C*, the write possibly resets the null, hence the additive update. In *D*, the first element in the range of possible null positions is overwritten with non-zero. This may overwrite the actual null, hence the range is extended without bound at the end and shortened by one at the beginning. In *E*, the additive update $P \triangleright (y_n \geq y_n)$ extends the null position to the right to capture a possibly overwritten null. In *F*, case *B* is refined so that the null exceeds neither upper bound. Finally *G* provides a conservative approximation when the update value is not known.

The abstract semantics for array write first inspects D^l – the definite sharing abstraction for program point l – to collect together those program variables W that definitely share with y . The polyhedra P is refined using D^l to obtain P' . V is constructed from $\varrho_{P'}(S^l)$ – the possible sharing abstraction for program point l refined by P' – to obtain those variables which possibly share with y , but do not definitely share with y . Information on null positions of those variables of W (except y) is then removed from P' . The update is applied to revise the null attribute for y , and then this change is reflected to other variables of W . Additive update is used to propagate the update to the variables in V since they may or may not have been affected by the write. This step-wise construction enables correctness to be established, and given correct sharing abstractions D^l and S^l for point l (see Section 3.3), the following correctness result is obtained.

Theorem 1. Suppose $L = [\text{skip}]^l \mid \dots \mid [x = \text{malloc}(y)]^l$. Then if $\rho \vdash \langle L, \sigma_1 \rangle \rightarrow \sigma_2, \langle \rho, \sigma_1 \rangle \in \gamma_X^{PB}(P)$ and $\langle L, P_1 \rangle \rightarrow' P_2$ it follows that $\langle \rho, \sigma_2 \rangle \in \gamma_X^{EPB}(P_2)$.

Note that if $P_2 = \text{err}$, then a definite overrun is detected (if l is ever reached); if $P_2 = \text{warn}$, then a possible overrun may occur at l ; whereas if $P \in PB_X$ then no overrun can occur at l .

Example 5. The consecutive updates of Example 3 correspond to case D in the table. The reverse writing of the first three characters corresponds to fourth column and second row and thus to case E. The last write is again a D update.

[if ₁ ']	$\langle \text{if } x \text{ then } L_1 \text{ else } L_2, P \rangle \rightarrow' \langle L_1, P \sqcap \{1 \leq x\} \rangle$
[if ₂ ']	$\langle \text{if } x \text{ then } L_1 \text{ else } L_2, P \rangle \rightarrow' \langle L_1, P \sqcap \{x \leq -1\} \rangle$
[if ₃ ']	$\langle \text{if } x \text{ then } L_1 \text{ else } L_2, P \rangle \rightarrow' \langle L_2, P \sqcap \{x = 0\} \rangle$
[while ₁ ']	$\langle \text{while } x \ L, P \rangle \rightarrow' \langle L; \text{ while } x \ L, P \sqcap \{1 \leq x\} \rangle$
[while ₂ ']	$\langle \text{while } x \ L, P \rangle \rightarrow' \langle L; \text{ while } x \ L, P \sqcap \{x \leq -1\} \rangle$
[while ₃ ']	$\langle \text{while } x \ L, P \rangle \rightarrow' P \sqcap \{x = 0\}$
[seq ₁ ']	$\frac{\langle L_1, P_1 \rangle \rightarrow' \langle L_2, P_2 \rangle}{\langle L_1; L_3, P_1 \rangle \rightarrow' \langle L_2; L_3, P_2 \rangle}$
[seq ₂ ']	$\frac{\langle L_1, P_1 \rangle \rightarrow' P_2}{\langle L_1; L_2, P_1 \rangle \rightarrow' \langle L_2, P_2 \rangle}$
[ret']	$\langle \text{return}; L, P \rangle \rightarrow' P$
[call']	$\frac{\langle \theta(z_1) = y_1; \dots; \theta(z_n) = y_n; \theta(L), P_1 \rangle \rightarrow'^* P_2}{\langle x = f(y_1, \dots, y_n), P_1 \rangle \rightarrow' \langle x = \theta(f_r), P_2 \rangle}$ $\text{if } c(f) = \langle z_1, \dots, z_n, L \rangle \wedge \text{var}(\theta(c(f))) \cap \text{var}(P_1) = \emptyset$

Fig. 5. Abstract semantics for control statements where θ denotes a renaming (bi-jective) substitution

4.4 Control Statements

Figure 5 details the abstract semantics for the control statements. Note how the if and while branches restrict the polyhedron, possibly collapsing it to *false*. The main safety result, Theorem 1, can be lifted to full String C by induction on the number of steps from main in a sequence of abstract reductions. Of course, an analysis will have to address finiteness issues, by applying widening [4], and efficiency issues, by combining function call and exit with projection so as to minimize the number of variables considered in the analysis of each function.

5 Related Work

Apart from those static analyses already discussed [8,12,13,23], most of proposals for detecting overruns are either based on lexical analysis [22], testing [11] or stack protection mechanisms [2,10]. ITS4 [22] is a lexical analysis tool that searches for security problems using a database of potentially dangerous constructs. Lexical analysis is fast and calls to problematic library functions can be flagged. Such a limited approach, however, will fail to find problematic string buffer manipulation that is hand coded.

FIST [11] finds possible overruns by automatically perturbing the program states, for example, appending or truncating strings, or by mangling the stack. The developer or analyst selects buffers to check, and then FIST injects a state perturbation to generate a possible overrun. Manual analysis is required to determine whether the overrun buffer can actually occur.

StackGuard protects from stack smashing by aborting the program if the return address is over-written [10]. The StackGuard compiler, however, does not bar overruns and any overrun has the potential of side-effecting a variable

and thereby altering access and privileges. Baratloo, Singh and Tsai [2] also describe a mechanism for checking the return address before the jump. Any run-time approach, however, will always incur an overhead (of 40% in the case of StackGuard [10]). Moreover, as pointed out in [12], these run-time systems effectively just turn a buffer overflow attack into a denial-of-service attack.

Finally, the AST ToolKit [6] has been used to detect errors in string manipulation, though details on this work are sparse [8].

6 Future Work

One direction for future work will be to investigate the extent to which polyhedral sub-domains [16] impact on the precision of buffer overrun analysis; a programmer might be willing to trade extra warning messages for an analysis that scales smoothly to large applications. For brevity, the analysis omits how to recover from an error or warning state. Future work will thus investigate how to safely reshape the polyhedron to satisfy the requirements of an operation so as to avoid an unhelpful error cascade. Future work will also address how to extend the analysis to Unicode buffers and other problematic buffer operations. For example, pointer difference $i = s - t$ is well defined iff s and t point to the same buffer. Moreover, the analysis can be enriched to flag an error if s and t do not possibly share and raise a warning if s and t do not definitely share.

7 Conclusions

This paper has formalized the buffer overrun problem by following the methodology of abstract interpretation. First, an instrumented semantics for a C subset was presented that captures those properties relevant to overrun detection. Second, polyhedral and buffer sharing domains were proposed and then connected with the concrete semantics via concretization maps. Third, the instrumented semantics was abstracted to synthesize an analysis for detecting both definite and possible overruns. Fourth, correctness results were reported. The paper provides an excellent practical foundation for overrun analysis since it explains how buffer sharing and buffer overrun interact and how sharing analysis can be used to trim down the number of variables in a polyhedral buffer abstraction.

Acknowledgements

We thank Florence Benoy, John Gallagher, Les Hatton and Jacob Howe for interesting discussions.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Datalogisk Institut Kobenhavns Universitet, 1994. 367, 368
2. A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *Ninth USENIX Security Symposium*, 2000. 377, 378

3. V. Chandru and M. R. Rao. Linear programming. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999. 370
4. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Proceedings of Principles of Programming Languages*, pages 84–97. ACM Press, 1978. 366, 370, 371, 377
5. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Information Survivability Conference and Exposition*, volume II, pages 154–163. IEEE Press, 1998. 365
6. R. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Conference on Domain-Specific Languages*, pages 229–242. USENIX Association, 1997. 378
7. B. De Backer and H. Beringer. A CLP language handling disjunctions of linear constraints. In *International Conference on Logic Programming*, pages 550–563. MIT Press, 1993. 370
8. N. Dor, M. Rodeh, and M. Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 194–212. Springer-Verlag, 2001. 366, 367, 377, 378
9. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural analysis in the presence of function pointers. In *Programming Language Design and Implementation*, pages 242–256, June 1994. 367
10. C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, 1998. 377, 378
11. A. Ghosh, T. O’Connor, and G. McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *IEEE Symposium on Security and Privacy*, pages 104–114. IEEE Computer Society, 1998. 365, 377
12. D. Larochelle and D. Evans. Statically Detecting likely Buffer Overflow Vulnerabilities. In *Tenth USENIX Security Symposium*. USENIX Association, 2001. 366, 377, 378
13. D. Larochelle and D. Evans. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002. 366, 377
14. B. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990. 365
15. T. C. Miller and T. de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *USENIX Annual Technical Conference*, 1999. 365, 366
16. A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects*, volume 2053 of *LNCS*, pages 155–172, 2001. 378
17. A. One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49). 365
18. N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998. 368
19. R. T. Rockafellar. *Convex Analysis*. Princeton University Press, 1970. 370
20. B. Snow. Panel Discussion on the Future of Security. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999. 365
21. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages*, pages 32–41. ACM Press, 1996. 367, 372
22. J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Sixteenth Annual Computer Security Applications Conference*, 2000. 377
23. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*. Internet Society, 2000. 366, 377

24. D. Weise. Static Analysis of Mega-Programs. In *Static Analysis Symposium*, volume 1694 of *LNCS*, pages 300–302. Springer-Verlag, 1999. [365](#)

A Foundation of Escape Analysis^{*}

Patricia M. Hill¹ and Fausto Spoto²

¹ School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

² Dipartimento di Informatica, Verona, Italy
Ph.: +44 01132336807 Fax: +44 01132335468
spoto@sci.univr.it

Abstract. Escape analysis of object-oriented languages allows us to stack allocate dynamically created objects and to reduce the overhead of synchronisation in Java-like languages. We formalise the *escape property* \mathcal{E} , computed by an escape analysis, as an abstract interpretation of concrete states. We define the optimal abstract operations induced by \mathcal{E} for a framework of analysis known as *watchpoint semantics*. The implementation of \mathcal{E} inside that framework is a formally correct abstract semantics (analyser) for escape analysis. We claim that \mathcal{E} is the basis for more refined and precise domains for escape analysis.

1 Introduction

Escape analysis has been studied for functional and for object-oriented languages. It allows us to stack allocate dynamically created data structures which would normally be heap allocated. This is possible if the data structures do not *escape* from the method which created them. It is well known that stack allocation reduces garbage collection overhead at run-time *w.r.t.* heap allocation. In the case of Java, escape analysis allows us to remove unnecessary synchronisations when an object is accessed. This is possible if the object does not *escape* the methods of its creating thread. This makes object accesses faster at run-time.

In this paper, we lay the foundations for the development of practical escape analysers for object-oriented languages. To do this, we first formalise the *escape property* \mathcal{E} as a property (an *abstract interpretation*) of concrete states showing how it may interact with both the static type information and the late-binding mechanism. We show how, given a set of concrete operations, such as those given in [9,13] for executing a simple object-oriented language, optimal abstract operations induced by \mathcal{E} may be constructed.

Note that, in this paper, we are concerned with providing a foundation for escape analysis and, as a consequence, the domain \mathcal{E} , which represents just the property of interest, does not include any other related information that could improve the precision of the analyser. Notwithstanding this, we show that, in

^{*} This work has been funded by EPSRC grant GR/R53401.

some cases, \mathcal{E} with its abstract operations can be more precise than other proposed escape analyses. Moreover, our escape analyser can form a basis for more refined analysers, such as the one described in [8], with improved precision.

After a brief summary of our notation and terminology, in Section 3 we recall the watchpoint framework of [13] on which the analysis is based. Then, in Section 4, we formalise the escape property of interest and provide suitable abstract operations for its analysis while, in Section 5, we discuss our prototype implementation and experimental results. Section 6 concludes the paper by highlighting the differences between previous proposals and our own approach.

2 Preliminaries

A total (partial) function f is denoted by $\mapsto (\rightarrow)$. The *domain* (*codomain*) of f is $\text{dom}(f)$ ($\text{rng}(f)$). We denote by $[v_1 \mapsto t_1, \dots, v_n \mapsto t_n]$ the function f where $\text{dom}(f) = \{v_1, \dots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \dots, n$. Its *update* is $f[w_1 \mapsto d_1, \dots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \text{dom}(f)$ (to $\text{dom}(f) \setminus s$). If $f(x) = x$ then x is a *fixpoint* of f . The set of fixpoints of f is denoted by $\text{fp}(f)$.

The two components of a *pair* are separated by \cdot . A definition of S such as $S = a \cdot b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$. An element x will often stand for the singleton set $\{x\}$, like l in \triangleleft_l in the definition of `put_field` (Figure 4).

A *complete lattice* is a poset $C \cdot \preceq$ where *least upper bounds* (lub) and *greatest lower bounds* (glb) always exist. If $C \cdot \preceq$ and $A \cdot \preceq$ are posets, then $f : C \mapsto A$ is (*co*)-*additive* if it preserves lub's (glb's). A map $f : A \mapsto A$ is a *lower closure operator* (*lco*) if it is *monotonic*, *reductive* and *idempotent*.

We recall now the basics of abstract interpretation (AI) [5]. Let $C \cdot \preceq$ and $A \cdot \preceq$ be two posets (the concrete and the abstract domain). A *Galois connection* is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ such that $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive. It is a *Galois insertion* when $\alpha\gamma$ is the identity map *i.e.*, when the abstract domain does not contain *useless* elements. This is equivalent to α being onto, or γ one-to-one. If C and A are complete lattices and α is additive, it is the abstraction map of a Galois connection. An abstract operator $\hat{f} : A^n \rightarrow A$ is *correct w.r.t.* $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$. For each operator f , there exists an *optimal* (most precise) correct abstract operator \hat{f} defined as $\hat{f} = \alpha f \gamma$. The *semantics* of a program is the fixpoint of a map $f : C \mapsto C$, where C is the *computational domain*. Its *collecting version* [5] works over *properties* of C *i.e.*, over $\wp(C)$ and is the fixpoint of the powerset extension of f . If f is defined through suboperations, their powerset extensions *and* \cup (which merges the semantics of the branches of a conditional) induce the extension of f .

3 The Framework of Analysis

We build on the *watchpoint semantics* [13] which allows us to derive a compositional and *focused* analyser from a specification of a domain of abstract states

```

class angle :
field degree : int
method acute() : int is
  out := this.degree < 90

class figure :
method def() : void is skip
method rot(a : angle) : void is skip
method draw() : void is skip

class square extends figure :
fields side, xcenter, ycenter : int
field rotation : angle
method def() : void is
  this.side := 1;
  this.xcenter := 0;
  this.ycenter := 0;
  this.rotation := new angle; {π1}
  this.rotation.degree = 0;
method rot(a : angle) : void is
  this.rotation := a;
method draw() : void is
  % something using this.rotation here...

class circle extends figure :
fields radius, xcenter, ycenter : int
method def() : void is
  this.radius := 1; {w1}
  this.xcenter, this.ycenter := 0;
method draw() : void is
  % put something here...

class main :
method main() : void is
  f : figure;
  f := new square; {π2}
  f.def();
  rotate(f); {w2}
  f := new circle; {π3}
  f.def();
  rotate(f); {w3}
method rotate(f : figure) : void is
  a : angle;
  a := new angle; {π4}
  f.rot(a);
  a.degree := 0;
  while (a.degree < 360)
    f.draw();
    a.degree := a.degree + 1;

```

Fig. 1. An example of program

and operations which work over them. Then problems such as scoping, recursion and name clash can be ignored, since these are already solved by the watchpoint semantics. Moreover, this framework relates the precision of the analysis to that of its abstract domain so that traditional techniques for comparing the precision of abstract domains can be applied [4,5,8].

The analyser presented here is for a simple typed object-oriented language where the concrete states and operations are based on [9]. However, here we require a more concrete notion of object. This is because, for escape analysis, every object has to be associated with its *creation point*.

Definition 1. Let Id be a set of identifiers, \mathcal{K} a finite set of classes ordered by a subclass relation \leq such that $\mathcal{K} \cdot \leq$ is a poset and $\text{main} \in \mathcal{K}$. Let $Type$ be the set $\{int\} + \mathcal{K}$. We extend \leq to $Type$ by defining $int \leq int$. Let $Vars \subseteq Id$ be a set of variables such that $\{\text{out}, \text{this}\} \subseteq Vars$. We define

$$TypEnv = \{\tau : Vars \rightarrow Type \mid \text{dom}(\tau) \text{ is finite, if } \text{this} \in \text{dom}(\tau) \text{ then } \tau(\text{this}) \in \mathcal{K}\}.$$

Types and type environments are initialised as $\text{init}(int) = 0$, $\text{init}(\kappa) = \text{nil}$ for $\kappa \in \mathcal{K}$ and $\text{init}(\tau)(v) = \text{init}(\tau(v))$ for $\tau \in TypEnv$ and $v \in \text{dom}(\tau)$.

A class contains local variables (*fields*) and functions (*methods*). A method has a set of input/output variables called *parameters*, including *out*, which holds

$$\mathcal{K} = \left\{ \begin{array}{l} \text{angle,} \\ \text{figure,} \\ \text{square,} \\ \text{circle,} \\ \text{main} \end{array} \right\} \quad \mathcal{M} = \left\{ \begin{array}{l} \text{angle_acute,} \\ \text{figure_def, figure_rot, figure_draw,} \\ \text{square_def, square_rot, square_draw,} \\ \text{circle_def, circle_draw,} \\ \text{main_main, main_rotate} \end{array} \right\}$$

square \leq figure, circle \leq figure and reflexive cases

$$\begin{aligned} F(\text{angle}) &= [\text{degree} \mapsto \text{int}] & F(\text{figure}) &= F(\text{main}) = [] \\ F(\text{square}) &= [\text{side} \mapsto \text{int}, \text{xcenter} \mapsto \text{int}, \text{ycenter} \mapsto \text{int}, \text{rotation} \mapsto \text{angle}] \\ F(\text{circle}) &= [\text{radius} \mapsto \text{int}, \text{xcenter} \mapsto \text{int}, \text{ycenter} \mapsto \text{int}] \\ M(\text{angle}) &= [\text{acute} \mapsto \text{angle_acute}] \\ M(\text{figure}) &= [\text{def} \mapsto \text{figure_def}, \text{rot} \mapsto \text{figure_rot}, \text{draw} \mapsto \text{figure_draw}] \\ M(\text{square}) &= [\text{def} \mapsto \text{square_def}, \text{rot} \mapsto \text{square_rot}, \text{draw} \mapsto \text{square_draw}] \\ M(\text{circle}) &= [\text{def} \mapsto \text{circle_def}, \text{rot} \mapsto \text{figure_rot}, \text{draw} \mapsto \text{circle_draw}] \\ M(\text{main}) &= [\text{main} \mapsto \text{main_main}, \text{rotate} \mapsto \text{main_rotate}] \\ P(\text{angle_acute}) &= [\text{out} \mapsto \text{int}, \text{this} \mapsto \text{angle}] \\ P(\text{figure_rot}) &= [\text{a} \mapsto \text{angle}, \text{out} \mapsto \text{int}, \text{this} \mapsto \text{figure}] \\ P(\text{main_rotate}) &= [\text{f} \mapsto \text{figure}, \text{out} \mapsto \text{int}, \text{this} \mapsto \text{main}] \\ P(\text{figure_def}) &= [\text{out} \mapsto \text{int}, \text{this} \mapsto \text{figure}] \text{ (the other cases of } P \text{ are like this)} \end{aligned}$$

Fig. 2. The static information of the program in Figure 1

the result of the method, and **this**, which is the object over which the method has been called. *Fields* is a set of maps which bind each class to the type environment of its fields. The variable **this** cannot be a field. *Methods* is a set of maps which bind each class to a map from identifiers to methods. *Pars* is a set of maps which bind each method to the type environment of its parameters (its signature).

Definition 2. Let \mathcal{M} be a finite set of methods. We define

$$\begin{aligned} \text{Fields} &= \{F : \mathcal{K} \mapsto \text{TypEnv} \mid \text{this} \notin \text{dom}(F(\kappa)) \text{ for every } \kappa \in \mathcal{K}\} \\ \text{Methods} &= \mathcal{K} \mapsto (\text{Id} \mapsto \mathcal{M}) \\ \text{Pars} &= \{P : \mathcal{M} \mapsto \text{TypEnv} \mid \{\text{out}, \text{this}\} \subseteq \text{dom}(P(\nu)) \text{ for every } \nu \in \mathcal{M}\} . \end{aligned}$$

The *static information* of a program is used by the escape analyser.

Definition 3. The static information of a program consists of a poset $\mathcal{K} \leq$, a set of methods \mathcal{M} and maps $F \in \text{Fields}$, $M \in \text{Methods}$ and $P \in \text{Pars}$.

An example program is given in Figure 1. Note that **void** methods are an abbreviation for methods which always return the integer 0, implicitly discarded by the caller. The static information of that program is shown in Figure 2.

Definition 4. Let Π be a finite set of labels called creation points. A map $k : \Pi \mapsto \mathcal{K}$ relates every creation point with the class of the objects it creates. A

hidden creation point $\bar{\pi} \in \Pi$, internal to the operating system, creates objects of class **main**. Then we define $k(\bar{\pi}) = \mathbf{main}$. Every other $\pi \in \Pi$ decorates a **new** κ statement in the program. Then we define $k(\pi) = \kappa$. Let $\pi \in \Pi$, $F \in \text{Fields}$ and $M \in \text{Methods}$. We define $F(\pi) = F(k(\pi))$ and $M(\pi) = M(k(\pi))$.

We define a *frame* as a map that assigns values to variables. These *values* can be integers, locations or *nil* where a *location* is a memory cell. The value assigned to a variable must be consistent with its type. For instance, a class variable should be assigned to a location or to *nil*. A *memory* is a map from locations to objects where an *object* is characterized by its creation point and the frame of its fields. Figure 3 illustrates these different concepts. An *update* of a memory allows its frames to assign new (type consistent) values to the variables.

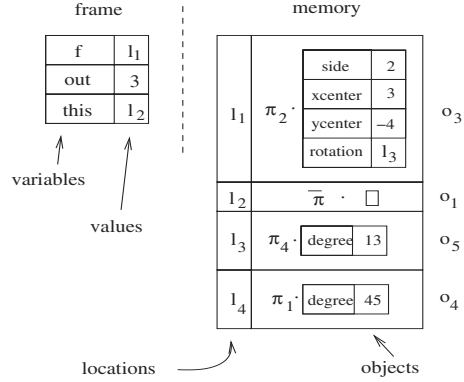


Fig. 3. Frame ϕ_1 and memory μ_1 for type environment $\tau_{w_2} = [\mathbf{f} \mapsto \mathbf{figure}, \mathbf{out} \mapsto \mathbf{int}, \mathbf{this} \mapsto \mathbf{main}]$

Definition 5. Let Loc be an infinite set of locations, $Value = \mathbb{Z} + Loc + \{\mathit{nil}\}$ and $\tau \in \text{TypEnv}$. We define frames, objects and memories as

$$Frame_{\tau} = \left\{ \phi \in \text{dom}(\tau) \mapsto Value \left| \begin{array}{l} \text{for every } v \in \text{dom}(\tau) \\ \text{if } \tau(v) = \mathit{int} \text{ then } \phi(v) \in \mathbb{Z} \\ \text{if } \tau(v) \in \mathcal{K} \text{ then } \phi(v) \in \{\mathit{nil}\} \cup Loc \end{array} \right. \right\}$$

$$Obj = \{\pi \cdot \phi \mid \pi \in \Pi, \phi \in Frame_{F(\pi)}\}$$

$$Memory = \{\mu \in Loc \rightarrow Obj \mid \text{dom}(\mu) \text{ is finite}\}.$$

Let $\mu_1, \mu_2 \in \text{Memory}$ and $L \subseteq \text{dom}(\mu_1)$. We say that μ_2 is an L -update of μ_1 , written $\mu_1 \triangleleft_L \mu_2$, if $L \subseteq \text{dom}(\mu_2)$ and for every $l \in L$ we have $\mu_1(l) \cdot \pi = \mu_2(l) \cdot \pi$.

Example 1. Let $\tau_{w_2} = [\mathbf{f} \mapsto \mathbf{figure}, \mathbf{out} \mapsto \mathbf{int}, \mathbf{this} \mapsto \mathbf{main}]$ be the type environment at program point w_2 in Figure 1. Letting $l_1, l_2 \in Loc$, the set $Frame_{\tau_{w_2}}$ contains ϕ_1 (see Figure 3) and ϕ_2 but it does not contain ϕ_3 and ϕ_4 , where

$$\begin{aligned} \phi_1 &= [\mathbf{f} \mapsto l_1, \mathbf{out} \mapsto 3, \mathbf{this} \mapsto l_2] & \phi_2 &= [\mathbf{f} \mapsto \mathit{nil}, \mathbf{out} \mapsto -2, \mathbf{this} \mapsto \mathit{nil}] \\ \phi_3 &= [\mathbf{f} \mapsto 2, \mathbf{out} \mapsto -2, \mathbf{this} \mapsto l_1] & \phi_4 &= [\mathbf{f} \mapsto l_1, \mathbf{out} \mapsto 3, \mathbf{this} \mapsto 3]. \end{aligned}$$

This is because **f** is bound to 2 in ϕ_3 (while it has class **figure** in τ_{w_2}) and **this** is bound to 3 in ϕ_4 (while it has class **main** in τ_{w_2}).

Example 2. Consider the program and its static information given in Figures 1 and 2. Since $F(\bar{\pi}) = F(\mathbf{main}) = []$, the only object created at $\bar{\pi}$ is $o_1 = \bar{\pi} \cdot []$.

Objects created at π_2 have class $k(\pi_2) = \mathbf{square}$. Examples of such objects (where $l_3 \in \mathit{Loc}$), that are consistent with the definition of $F(\mathbf{square})$, are

$$\begin{aligned} o_2 &= \pi_2 \cdot [\mathbf{side} \mapsto 4, \mathbf{xcenter} \mapsto 3, \mathbf{ycenter} \mapsto -5, \mathbf{rotation} \mapsto \mathit{nil}] , \\ o_3 &= \pi_2 \cdot [\mathbf{side} \mapsto 2, \mathbf{xcenter} \mapsto 3, \mathbf{ycenter} \mapsto -4, \mathbf{rotation} \mapsto l_3] . \end{aligned}$$

Finally, examples of objects created in π_1 and π_4 , respectively, are

$$o_4 = \pi_1 \cdot [\mathbf{degree} \mapsto 45] \quad o_5 = \pi_4 \cdot [\mathbf{degree} \mapsto 13] .$$

(o_1, o_3, o_4 and o_5 are illustrated in Figure 3.)

Example 3. Consider locations $l_1, l_2, l_3, l_4 \in \mathit{Loc}$ and the objects o_1, o_2, o_3, o_4, o_5 from Example 2. Then *Memory* contains $\mu_1 = [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4]$ (Figure 3), $\mu_2 = [l_1 \mapsto o_1, l_2 \mapsto o_1]$ and $\mu_3 = [l_1 \mapsto o_2, l_2 \mapsto o_3, l_3 \mapsto o_1]$.

We define a notion of type correctness which guarantees that variables are bound to locations which contain objects allowed by a given type environment.

Definition 6. Let $\tau \in \mathit{TypEnv}$, $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is weakly τ -correct w.r.t. μ if for every $v \in \mathit{dom}(\phi)$ such that $\phi(v) \in \mathit{Loc}$ we have $\phi(v) \in \mathit{dom}(\mu)$ and $k((\mu\phi(v)).\pi) \leq \tau(v)$.

We strengthen the correctness notion of Definition 6 by requiring that it holds for the fields of the objects in memory also.

Definition 7. Let $\tau \in \mathit{TypEnv}$, $\phi \in \mathit{Frame}_\tau$ and $\mu \in \mathit{Memory}$. We say that ϕ is τ -correct w.r.t. μ , and we write $\phi \cdot \mu : \tau$, if

1. ϕ is weakly τ -correct w.r.t. μ (Definition 6),
2. for every $o \in \mathit{rng}(\mu)$ we have that $o.\phi$ is weakly $F(o.\kappa)$ -correct w.r.t. μ .

Example 4. Let τ_{w_2} , ϕ_1 and ϕ_2 from Example 1, μ_1, μ_2 and μ_3 from Example 3. Let $\tau = \tau_{w_2}$. We have

- $\phi_1 \cdot \mu_1 : \tau$ (Figure 3). Condition 1 of Definition 7 holds because $\{v \in \mathit{dom}(\phi_1) \mid \phi_1(v) \in \mathit{Loc}\} = \{\mathbf{this}, \mathbf{f}\}$, $\{l_1, l_2\} \subseteq \mathit{dom}(\mu_1)$, $k(\mu_1(l_1).\pi) = k(\pi_2) = \mathbf{square} \leq \mathbf{figure} = \tau(\mathbf{f})$ and $k(\mu_1(l_2).\pi) = k(\bar{\pi}) = \mathbf{main} = \tau(\mathbf{this})$. Condition 2 holds because the only $o \in \mathit{rng}(\mu_1)$ such that $\mathit{rng}(o.\phi) \cap \mathit{Loc} \neq \emptyset$ is o_3 , since $\{l_3\} = \mathit{rng}(o_3.\phi) \cap \mathit{Loc}$. Moreover, $k(\mu_1(l_3).\pi) = k(\pi_4) = \mathbf{angle} = F(o_3.\pi)(\mathbf{rotation})$.
- $\phi_2 \cdot \mu_2 : \tau$. Condition 1 of Definition 7 holds since there is no $v \in \mathit{dom}(\phi_2)$ such that $\phi_2(v) \in \mathit{Loc}$. Condition 2 holds since $\mathit{rng}(\mu_2) = \{o_1\}$ and $o_1.\phi = []$.
- $\phi_1 \cdot \mu_2 : \tau$ is false since condition 1 of Definition 7 does not hold. Namely, $\tau(\mathbf{f}) = \mathbf{figure}$, $k((\mu_2\phi_1(\mathbf{f})).\pi) = k(o_1.\pi) = k(\bar{\pi}) = \mathbf{main}$ and $\mathbf{main} \not\leq \mathbf{figure}$.
- $\phi_2 \cdot \mu_3 : \tau$ is false since condition 2 of Definition 7 does not hold. Namely, $o_3 \in \mathit{rng}(\mu_3)$ is such that $o_3.\phi$ is not weakly $F(o_3.\pi)$ -correct w.r.t. μ_3 , since $o_3.\phi(\mathbf{rotation}) = l_3$, $\mathbf{main} \not\leq \mathbf{angle}$ but $k(\mu_3(l_3).\pi) = k(o_1.\pi) = k(\bar{\pi}) = \mathbf{main}$ and we have $F(o_3.\pi)(\mathbf{rotation}) = F(\mathbf{square})(\mathbf{rotation}) = \mathbf{angle}$.

$$\begin{aligned}
\text{nop}_\tau(\phi \cdot \mu) &= \phi \cdot \mu & \text{get_int}_\tau^i(\phi \cdot \mu) &= \phi[\text{res} \mapsto i] \cdot \mu, \quad i \in \mathbb{Z} \\
\text{get_nil}_\tau^\kappa(\phi \cdot \mu) &= \phi[\text{res} \mapsto \text{nil}] \cdot \mu, \quad \kappa \in \mathcal{K} & \text{get_var}_\tau^v(\phi \cdot \mu) &= \phi[\text{res} \mapsto \phi(v)] \cdot \mu, \quad v \in \text{dom}(\tau) \\
\text{restrict}_\tau^{vs}(\phi \cdot \mu) &= \phi|_{-vs} \cdot \mu, \quad vs \subseteq \text{dom}(\tau) & \text{expand}_\tau^{v:t}(\phi \cdot \mu) &= \phi[v \mapsto \text{init}(t)] \cdot \mu, \quad t \in \text{Type} \\
\text{put_var}_\tau^v(\phi \cdot \mu) &= \phi[v \mapsto \phi(\text{res})]|_{-res} \cdot \mu, \quad v \in \text{dom}(\tau) \\
\text{get_field}_\tau^f(\phi' \cdot \mu) &= \begin{cases} \phi'[\text{res} \mapsto ((\mu\phi'(\text{res})).\phi)(f)] \cdot \mu & \text{if } \phi'(\text{res}) \neq \text{nil} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\text{put_field}_\tau^f(\phi_1 \cdot \mu_1)(\phi_2 \cdot \mu_2) &= \begin{cases} \phi_2|_{-res} \cdot \mu_2[l \mapsto \mu_2(l) \cdot \pi \cdot \mu_2(l) \cdot \phi[f \mapsto \phi_2(\text{res})]] \\ \quad \text{if } (l = \phi_1(\text{res})) \neq \text{nil} \text{ and } \mu_1 \triangleleft_l \mu_2 \\ \text{undefined} & \text{otherwise} \end{cases} \\
=_{\tau}(\phi_1 \cdot \mu_1)(\phi_2 \cdot \mu_2) &= \begin{cases} \phi_2[\text{res} \mapsto 1] \cdot \mu_2 & \text{if } \phi_1(\text{res}) = \phi_2(\text{res}) \\ \phi_2[\text{res} \mapsto -1] \cdot \mu_2 & \text{if } \phi_1(\text{res}) \neq \phi_2(\text{res}) \end{cases} \\
+_{\tau}(\phi_1 \cdot \mu_1)(\phi_2 \cdot \mu_2) &= \phi_2[\text{res} \mapsto \phi_1(\text{res}) + \phi_2(\text{res})] \cdot \mu_2 \\
\text{is_nil}_\tau(\phi \cdot \mu) &= \begin{cases} \phi[\text{res} \mapsto 1] \cdot \mu & \text{if } \phi(\text{res}) = \text{nil} \\ \phi[\text{res} \mapsto -1] \cdot \mu & \text{otherwise} \end{cases} \\
\text{call}_\tau^{\nu, v_1, \dots, v_n}(\phi \cdot \mu) &= [\iota_1 \mapsto \phi(v_1), \dots, \iota_n \mapsto \phi(v_n), \text{this} \mapsto \phi(\text{res})] \cdot \mu \\
\text{where } \{\iota_1, \dots, \iota_n\} &= P(\nu) \setminus \{\text{out}, \text{this}\} \text{ (alphabetically ordered) and } \nu \in \mathcal{M} \\
\text{return}_\tau^{\nu}(\phi_1 \cdot \mu_1)(\phi_2 \cdot \mu_2) &= \begin{cases} \phi_1[\text{res} \mapsto \phi_2(\text{out})] \cdot \mu_2 & \text{if } \mu_1 \triangleleft_{\text{img}(\phi_1)|_{-res} \cap \text{Loc}} \mu_2 \\ \text{undefined} & \text{otherwise} \end{cases} \\
\text{new}_\tau^\pi(\phi \cdot \mu) &= \phi[\text{res} \mapsto l] \cdot \mu[l \mapsto \pi \cdot \text{init}(F(\pi))], \quad \pi \in \Pi, \quad l \in \text{Loc} \setminus \text{dom}(\mu) \\
\text{lookup}_\tau^{m, \nu}(\phi \cdot \mu) &\text{ iff } \phi(\text{res}) \neq \text{nil} \text{ and } M((\mu\phi(\text{res})).\pi)(m) = \nu \\
\text{is_true}_\tau(\phi \cdot \mu) &\text{ iff } \phi(\text{res}) \geq 0 & \text{is_false}_\tau(\phi \cdot \mu) &\text{ iff } \phi(\text{res}) < 0.
\end{aligned}$$

Fig. 4. The operations over concrete states

The state of the computation is a pair consisting of a frame and a memory. The variable **this** in the domain of the frame must be bound to an object.

Definition 8. Let $\tau \in \text{TypEnv}$. We define the states

$$\Sigma_\tau = \left\{ \phi \cdot \mu \left| \begin{array}{l} \phi \in \text{Frame}_\tau, \mu \in \text{Memory}, \phi \cdot \mu : \tau, \\ \text{if } \text{this} \in \text{dom}(\tau) \text{ then } \phi(\text{this}) \neq \text{nil} \end{array} \right. \right\}$$

and the operations over states in Figure 4 (for their signature, see [13]).

Example 5. In the hypotheses of Example 4, we have $\phi_1 \cdot \mu_1 \in \Sigma_\tau$ (Figure 3), $\phi_2 \cdot \mu_2 \notin \Sigma_\tau$ although $\phi_2 \cdot \mu_2 : \tau$ (Example 4), since **this** $\in \text{dom}(\tau)$ and $\phi_2(\text{this}) = \text{nil}$. We have $\phi_1 \cdot \mu_2 \notin \Sigma_\tau$ and $\phi_2 \cdot \mu_3 \notin \Sigma_\tau$ (Example 4).

In Figure 4, the subscript τ constrains the signature of the operations (see [13]). The variable *res* holds the intermediate results, like the top element of the operand stack of the Java virtual machine. The **nop** operation does nothing. The **get** operations load a constant, the value of another variable or of the field of an object in *res*. In the last case, that object is assumed to be stored in *res* before the **get** operation. The **put** operations store in *v* the value of *res* or of a field of an object pointed to by *res*. In this second case (**put.field**) the object whose field is modified must exist in the memory of the state holding the new value of the

field. This is expressed by the update relation (Definition 5). For every binary operation over values like $=$ and $+$, there is an operation on states. Note (in the case of $=$) that Booleans are implemented through integers (every non-negative integer means true). The operation `is_nil` checks whether `res` points to `nil`. The operation `call` (`return`) is used before (after) a call to a method ν . While `call ν` creates a new state in which ν can execute, the operation `return ν` restores the state σ which was current before the call to ν , and stores in `res` the result of the call. The update relation (Definition 5) requires that the variables of σ have not been changed during the execution of the method. The operation `expand` (`restrict`) adds (removes) variables. The operation `new π` creates a new object of creation point π . The predicate `lookup $^{m,\nu}$` checks if by calling the method identified by m of the object pointed to by `res`, the method ν is run. The predicate `is_true` (`is_false`) checks if `res` contains true (false).

Example 6. Let τ_{w_2} and ϕ_1 as in Example 1 and μ_1 as in Example 3 (see also Figure 3). The state $\sigma_1 = \phi_1 \cdot \mu_1$ (Example 5) could be the current state at program point w_2 in Figure 1. The computation continues as follows [13]. Let $\tau = \tau_{w_2}$ and $\tau' = \tau[\text{res} \mapsto \text{figure}]$. The sequence of operations which are executed is

$$\begin{array}{ll}
\sigma_2 = \text{new}_{\tau}^{\pi_3}(\sigma_1) & \text{create an object of class } \text{circle} \\
\sigma_3 = \text{put_var}_{\tau[\text{res} \mapsto \text{circle}]}^{\text{f}}(\sigma_2) & \text{assign it to } \text{f} \\
\sigma_4 = \text{get_var}_{\tau}^{\text{f}}(\sigma_3) & \text{read } \text{f} \\
\text{lookup}_{\tau'}^{\text{def,figure_def}}(\sigma_4) & \text{if it is true, let } \nu = \text{figure_def} \\
\text{lookup}_{\tau'}^{\text{def,square_def}}(\sigma_4) & \text{if it is true, let } \nu = \text{square_def} \\
\text{lookup}_{\tau'}^{\text{def,circle_def}}(\sigma_4) & \text{if it is true, let } \nu = \text{circle_def} \\
\sigma_5 = \text{call}_{\tau[\text{res} \mapsto P(\nu)(\text{this})]}^{\nu}(\sigma_4) & \text{initialise the selected method } \nu.
\end{array} \tag{1}$$

Let $o_6 = \pi_3 \cdot [\text{radius} \mapsto 0, \text{xcenter} \mapsto 0, \text{ycenter} \mapsto 0]$ and $l \in \text{Loc} \setminus \{l_1, l_2, l_3, l_4\}$. We have

$$\begin{aligned}
\sigma_2 &= [\text{f} \mapsto l_1, \text{out} \mapsto 3, \text{res} \mapsto l, \text{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] \\
\sigma_3 &= [\text{f} \mapsto l, \text{out} \mapsto 3, \text{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] \\
\sigma_4 &= [\text{f} \mapsto l, \text{out} \mapsto 3, \text{res} \mapsto l, \text{this} \mapsto l_2] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] .
\end{aligned}$$

Consider the `lookup` checks. They determine which is the target of the virtual call `f.def()` in Figure 1. We have $(\sigma_4.\phi)(\text{res}) = l \neq \text{nil}$ and $(\sigma_4.\mu)(l) = o_6$. Then the method of o_6 identified by `def` is called. We have $o_6.\pi = \pi_3$ and $k(\pi_3) = \text{circle}$. Moreover $M(\text{circle})(\text{def}) = \text{circle_def}$ (Figure 2). Then the only `lookup` check which succeeds is the last one, $\nu = \text{circle_def}$ is called and

$$\sigma_5 = [\text{this} \mapsto l] \cdot [l_1 \mapsto o_3, l_2 \mapsto o_1, l_3 \mapsto o_5, l_4 \mapsto o_4, l \mapsto o_6] .$$

Note that the object o_6 created by the `new` statement is now the `this` object of this instantiation of the method `circle_def`.

In [13] it is shown that every abstraction of $\wp(\Sigma_{\tau})$ and of the powerset extension of the operations in Figure 4 and of \cup induces an abstraction of the watchpoint semantics. We will use this result in the next section.

4 The Property \mathcal{E}

“Escape analysis determines whether the lifetime of data exceeds its static scope” [2]. “An object o is said to escape a method m if the lifetime of o may exceed the lifetime of m ” [3]. “Escape analysis computes bounds of where references to newly created objects may occur” [7]. These definitions of escape analysis for object-oriented languages are all unsatisfactory for a number of reasons. First of all, it is not clear which is the *actor* of the analysis (data, objects, references?). Furthermore, they use informal concepts (*lifetime*, *bound*, *newly created*). Finally, it is not clear whether escape analysis is a definite or possible analysis (do we want a subset or a superset of the “data whose lifetime exceeds their static scope”?). Thus a comparison of the precision of different analyses is possible only by an experimental evaluation where the number and degree of optimisations they achieve are compared. However, even this would be difficult as a standard and representative set of benchmarks is needed. In addition to this, the definitions refer to properties of programs rather than properties of states. For instance, *lifetime* refers implicitly to different moments in the execution of a program. So that, although the above definitions are useful for understanding the problem, they cannot be used to define abstract interpretations of states.

We now want to define more formally the *escape property* \mathcal{E} as a property of states, independently of the optimisations it allows. This is important because:

- It makes clear that escape analysis is a *possible* analysis of creation points,
- \mathcal{E} is the *simplest* abstract domain for escape analysis which is a concretisation of the escape property itself,
- As a property of states, it can be defined without any consideration about name clashes or multiple instances of the same variable during recursion,
- It allows a *formal* comparison of different domains *w.r.t.* their precision. It is indeed enough to compute and compare their *quotients w.r.t.* \mathcal{E} [4,8].

Here is our definition of escape analysis, which we will further formalise later.

Escape analysis *overapproximates*, for every program point p , the set of creation points of objects reachable in p from some variable or field in scope.

The choice of an *overapproximation* follows from the typical use of the information provided by *escape analysis*. For instance, an object can be stack allocated [2,3,7] if it does not *escape* the method which creates it *i.e.*, if it does not belong to a superset of the objects reachable at its end. For the same reason, we are interested in the creation points of the objects and not in their identity.

Example 7. Consider the program in Figure 1. Are there any objects created in π_4 and reachable in program point w_2 ? We have $\tau_{w_2} = [\mathbf{f} \mapsto \mathbf{figure}, \mathbf{out} \mapsto \mathbf{int}, \mathbf{this} \mapsto \mathbf{main}]$ (Example 1). We cannot reach any object from \mathbf{out} , because it can only contain integers. The variable \mathbf{this} has class \mathbf{main} which has no fields. Since in π_4 we create objects of class \mathbf{angle} , they cannot be reached from \mathbf{this} . The variable \mathbf{f} has class \mathbf{figure} , which has no fields. Reasoning as for \mathbf{this} , we could conclude that no object created in π_4 can be reached from \mathbf{f} .

That conclusion is wrong. Indeed, since `f` contains a `square`, the call `rotate(f)` results in a call `f.rot(a)` which stores `a`, created in π_4 , in the field `rotation`, and that field can later be accessed (for instance, by `f.draw()`).

The considerations in Example 7 lead to the definition of *reachability*, where we use the actual fields of the objects instead of those of its declared class. We use the following result to guarantee that α (Definition 9) is well-defined.

Lemma 1. *Let $\tau \in \text{TypEnv}$, $\phi \cdot \mu \in \Sigma_\tau$ and $o \in \text{rng}(\mu)$. Then $o.\phi \cdot \mu \in \Sigma_{F(o,\pi)}$.*

Definition 9. *Let $\tau \in \text{TypEnv}$ and $\sigma = \phi \cdot \mu \in \Sigma_\tau$. The set of the creation points of the objects reachable in σ is $\alpha_\tau(\sigma) = \cup\{\alpha_\tau^i(\sigma) \mid i \geq 0\} \subseteq \Pi$, where*

$$\begin{aligned} \alpha_\tau^0(\sigma) &= \emptyset \\ \alpha_\tau^{i+1}(\sigma) &= \cup\{\{o.\pi\} \cup \alpha_{F(o,\pi)}^i(o.\phi \cdot \mu) \mid v \in \text{dom}(\phi), \phi(v) \in \text{Loc}, o = \mu\phi(v)\}. \end{aligned}$$

The maps α_τ and α_τ^i are pointwise extended to $\wp(\Sigma_\tau)$.

Variables and fields of type *int* do not contribute to α_τ . Moreover, since Π is finite and $\alpha_\tau^i \subseteq \alpha_\tau^{i+1}$ (by induction over i), for every $S \subseteq \Pi$ there is $k \in \mathbb{N}$ such that $\alpha_\tau(S) = \alpha_\tau^k(S)$.

Example 8. In the hypotheses of Figure 3, we have $\alpha_\tau(\phi_1 \cdot \mu_1) = \{\bar{\pi}, \pi_2, \pi_4\}$. Note that o_4 is not reachable.

In general, it can be $\text{rng}(\alpha_\tau) \neq \wp(\Pi)$, since α_τ is not necessarily onto.

Example 9. In the hypotheses of Figure 2, let $\tau = [\mathbf{a} \mapsto \text{circle}, \text{this} \mapsto \text{main}]$ and $\sigma = \phi \cdot \mu \in \Sigma_\tau$. Then $\pi_4 \notin \alpha_\tau(\sigma)$, since for every $i \in \mathbb{N}$ we have

$$\alpha_\tau^{i+1}(\sigma) = \cup\{\{o.\pi\} \cup \alpha_{F(o,\pi)}^i(o.\phi \cdot \mu) \mid v \in \{\mathbf{a}, \text{this}\}, \phi(v) \in \text{Loc}, o = \mu\phi(v)\}. \quad (2)$$

Since $k(o,\pi) \in \{\text{circle}, \text{main}\}$ has no class fields, (2) is equal to

$$\{o.\pi \mid v \in \{\mathbf{a}, \text{this}\}, \phi(v) \in \text{Loc}, o = \mu\phi(v)\}. \quad (3)$$

Since π_1, π_2 and π_4 create objects incompatible with `circle` and `main`, (3) is contained in $\{\bar{\pi}, \pi_3\}$. Since i is arbitrary, we have the thesis.

Example 9 shows that *static type information provides escape information*, by indicating some creation points which create objects that cannot be reached.

Let $S \in \wp(\Pi)$. We define $\delta_\tau(S)$ as the largest subset of S which contains only those creation points deemed useful by the type environment τ . Note that if there are no possible creation points for `this`, all creation points are useless.

Definition 10. *Let $\tau \in \text{TypEnv}$ and $S \subseteq \Pi$. We define $\delta_\tau(S) = \cup\{\delta_\tau^i(S) \mid i \geq 0\}$ with*

$$\begin{aligned} \delta_\tau^0(S) &= \emptyset \\ \delta_\tau^{i+1}(S) &= \begin{cases} \emptyset & \text{if } \text{this} \in \text{dom}(\tau) \text{ and there is no } \pi \in S \text{ s.t. } k(\pi) \leq \tau(\text{this}) \\ \cup\{\{\pi\} \cup \delta_{F(\pi)}^i(S) \mid \kappa \in \text{rng}(\tau) \cap \mathcal{K}, \pi \in S, k(\pi) \leq \kappa\} & \text{otherwise.} \end{cases} \end{aligned}$$

In Definition 10 we consider all subclasses of κ (Example 7). We have $\delta_\tau = \delta_\tau^{\#II}$.

Proposition 1. *Let $\tau \in TypEnv$ and $i \in \mathbb{N}$. The maps δ_τ^i and δ_τ are lco's.*

Example 10. Program point w_1 in Figure 1 has type environment $\tau_{w_1} = [\text{out} \mapsto \text{int}, \text{this} \mapsto \text{circle}]$. Let $S = \{\bar{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$. For every $i \in \mathbb{N}$ we have

$$\begin{aligned} \delta_{\tau_{w_1}}^{i+1}(S) &= \cup\{\{\pi\} \cup \delta_{F(\pi)}^i(S) \mid \kappa \in \text{rng}(\tau_{w_1}) \cap \mathcal{K}, \pi \in S, k(\pi) \leq \kappa\} \\ &= \cup\{\{\pi\} \cup \delta_{F(\pi)}^i(S) \mid \pi \in S, k(\pi) \leq \text{circle}\} = \{\pi_3\} \cup \delta_{F(\text{circle})}^i(S). \end{aligned}$$

Since $\text{rng}(F(\text{circle})) = \{\text{int}\}$, we conclude that $\delta_{\tau_{w_1}}(S) = \{\pi_3\}$.

Example 11. Consider the program point w_2 in Figure 1 and its type environment τ_{w_2} from Example 1. Let $S = \{\bar{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}$. For every $i \in \mathbb{N}$ we have

$$\begin{aligned} \delta_{\tau_{w_2}}^{i+2}(S) &= \cup\{\{\pi\} \cup \delta_{F(\pi)}^{i+1}(S) \mid \kappa \in \{\text{figure, main}\}, \pi \in S, k(\pi) \leq \kappa\} \\ &= \{\bar{\pi}, \pi_2, \pi_3\} \cup \delta_{F(\text{square})}^{i+1}(S) \cup \delta_{F(\text{circle})}^{i+1}(S) \cup \delta_{F(\text{main})}^{i+1}(S) \\ &= \{\bar{\pi}, \pi_2, \pi_3\} \cup \delta_{F(\text{square})}^{i+1}(S) \\ &= \{\bar{\pi}, \pi_2, \pi_3\} \cup (\cup\{\{\pi\} \cup \delta_{F(\pi)}^i(S) \mid \pi \in S, k(\pi) \leq \text{angle}\}) \\ &= \{\bar{\pi}, \pi_2, \pi_3\} \cup (\{\pi_1, \pi_4\} \cup \delta_{F(\text{angle})}^i(S)) = \{\bar{\pi}, \pi_1, \pi_2, \pi_3, \pi_4\}. \end{aligned}$$

Then all creation points are useful in w_2 (compare this with Example 7).

The following result proves that δ_τ can be used to define $\text{rng}(\alpha_\tau)$.

Lemma 2. *Let $\tau \in TypEnv$. Then $\text{fp}(\delta_\tau) = \text{rng}(\alpha_\tau)$ and $\emptyset \in \text{fp}(\delta_\tau)$. If, in addition, $\text{this} \in \text{dom}(\tau)$, then for every $X \subseteq \Sigma_\tau$ we have $\alpha_\tau(X) = \emptyset$ if and only if $X = \emptyset$.*

Lemma 2 allows us to assume that $\alpha_\tau : \wp(\Sigma_\tau) \mapsto \text{fp}(\delta_\tau)$. Moreover, it justifies the following definition of the simplest domain \mathcal{E} for escape analysis. It coincides with the *escape property* itself. Therefore, Lemma 2 can be used to compute the possible values of the escape property at a given program point. However, it does not specify which of these is best. This is the goal of an escape analysis.

Definition 11. *Let $\tau \in TypEnv$. The escape property is $\mathcal{E}_\tau = \text{fp}(\delta_\tau)$, ordered by set inclusion.*

Example 12. Let τ_{w_1} and τ_{w_2} be as in Examples 10 and 1, respectively. We have

$$\mathcal{E}_{\tau_{w_1}} = \{\emptyset, \{\pi_3\}\}, \quad \mathcal{E}_{\tau_{w_2}} = \{\emptyset\} \cup \left\{ S \in \wp(II) \mid \begin{array}{l} \bar{\pi} \in S \text{ and} \\ \pi_1 \in S \text{ or } \pi_4 \in S \text{ entails } \pi_2 \in S \end{array} \right\}.$$

The constraint on $\mathcal{E}_{\tau_{w_2}}$ says that to reach an **angle** (created in π_1 or in π_4) from the variables in $\text{dom}(\tau_{w_2})$, we must be able to reach a **square** (created in π_2).

By Definition 9, if $\tau \in TypEnv$, then α_τ is strict and additive and, by Lemma 2, α_τ is onto \mathcal{E}_τ . Thus we have the following result.

$$\begin{array}{ll}
\text{nop}_\tau(S) = S & \text{get_int}_\tau^i(S) = S \\
\text{get_nil}_\tau^\kappa(S) = S & \text{get_var}_\tau^v(S) = S \\
\text{is_true}_\tau(S) = S & \text{is_false}_\tau(S) = S \\
\text{put_var}_\tau^v(S) = \delta_{\tau|_{-v}}(S) & \text{is_nil}_\tau(S) = \delta_{\tau|_{-res}}(S) \\
\text{new}_\tau^\pi(S) = S \cup \{\pi\} & =_\tau(S_1)(S_2) = +_\tau(S_1)(S_2) = S_2 \\
\text{expand}_\tau^{v:t}(S) = S & \text{restrict}_\tau^{vs}(S) = \delta_{\tau_{-vs}}(S) \\
\text{call}_\tau^{v_1, \dots, v_n}(S) = \delta_{\tau|_{\{v_1, \dots, v_n, res\}}}(S) & \cup_\tau(S_1)(S_2) = S_1 \cup S_2 \\
\text{get_field}_\tau^f(S) = \begin{cases} \emptyset & \text{if } \{\pi \in S \mid k(\pi) \leq \tau(res)\} = \emptyset \\ \delta_{\tau|_{[res \mapsto F(\tau(res))(f)]}}(S) & \text{otherwise} \end{cases} & \\
\text{put_field}_\tau^f(S_1)(S_2) = \begin{cases} \emptyset & \text{if } \{\pi \in S_1 \mid k(\pi) \leq \tau(res)\} = \emptyset \\ \delta_{\tau|_{-res}}(S_2) & \text{otherwise} \end{cases} & \\
\text{return}_\tau^\nu(S_1)(S_2) = \cup\{\{\pi\} \cup \delta_{F(\pi)}(II) \mid \kappa \in \text{rng}(\tau|_{-res}) \cap \mathcal{K}, \pi \in S_1, k(\pi) \leq \kappa\} \cup S_2 & \\
\text{lookup}_\tau^{m, \nu}(S) = \begin{cases} \emptyset & \text{if } S' = \{\pi \in S \mid k(\pi) \leq \tau(res), M(\pi)(m) = \nu\} = \emptyset \\ \delta_{\tau|_{-res}}(S) \cup (\cup\{\{\pi\} \cup \delta_{F(\pi)}(S) \mid \pi \in S'\}) & \text{otherwise.} \end{cases} &
\end{array}$$

Fig. 5. The optimal abstract operations over \mathcal{E}

Proposition 2. Let $\tau \in \text{TypEnv}$. The map α_τ (Definition 9) is the abstraction map of a Galois insertion from $\wp(\Sigma_\tau)$ to \mathcal{E}_τ .

Note that if, in Definition 11, we had defined \mathcal{E}_τ as $\wp(II)$, the map α_τ would induce just a Galois connection instead of a Galois insertion, as a consequence of Lemma 2.

The domain \mathcal{E} of Definition 11 can be used to compare abstract domains *w.r.t.* the *escape property* [4]. Moreover, since it is an abstract domain, it induces optimal abstract operations which can be used for an actual escape analysis.

Proposition 3. The operations in Figure 5 are the optimal counterparts induced by α of the operations in Figure 4 and of \cup . They are implicitly strict on \emptyset , except for **return**, which is strict in its first argument only, and for \cup .

Note how the operations in Figure 5 make extensive use through δ of the static type information of the variables to improve the precision of the analysis.

We provide now the worst-case complexity of the analysis induced by \mathcal{E} .

Proposition 4. Let $p = \#II$, n be the number of commands of the program P to be analysed, b the number of nodes of the call graph of P and t the maximal number of fields of an object or of variables in scope. The cost in time of the analysis induced by \mathcal{E} is $O(bnt2^p)$.

Example 13. Let us mimic, in \mathcal{E} , the concrete computation (1) of Example 6. We start from the abstraction (Definition 9) of σ_1 i.e., $\{\bar{\pi}, \pi_2, \pi_4\}$ (Example 8).

$$\begin{aligned} S_1 &= \alpha_\tau(\{\sigma_1\}) = \alpha_\tau(\sigma_1) = \{\bar{\pi}, \pi_2, \pi_4\} \\ S_2 &= \text{new}_\tau^{\pi_3}(S_1) = \{\bar{\pi}, \pi_2, \pi_3, \pi_4\} \\ S_3 &= \text{put_var}_\tau^{\text{f}}[\text{res} \mapsto \text{circle}](S_2) = \delta_{\left[\begin{smallmatrix} \text{out} \mapsto \text{int}, \text{res} \mapsto \text{circle} \\ \text{this} \mapsto \text{main} \end{smallmatrix} \right]}(\{\bar{\pi}, \pi_2, \pi_3, \pi_4\}) = \{\bar{\pi}, \pi_3\} \\ S_4 &= \text{get_var}_\tau^{\text{f}}(S_3) = \{\bar{\pi}, \pi_3\} . \end{aligned}$$

Note that $\pi_2 \notin S_3$ i.e., the analysis induced by \mathcal{E} is far from being a blind *accumulation* of creation points. Moreover, that information allows us to guess precisely the target of the virtual call $\text{f.def}()$. Indeed, the abstract `lookup` computes the abstract states S' , S'' and S''' which hold if its target is `figure_def`, `square_def` and `circle_def`, respectively, and only one of them is non-empty.

$$\begin{aligned} S' &= \text{lookup}_{\tau'}^{\text{def}, \text{figure_def}}(S_4) = \emptyset \\ &\text{since } \{\pi \in S_4 \mid k(\pi) \leq \text{figure} \text{ and } M(\pi)(\text{def}) = \text{figure_def}\} = \emptyset \\ S'' &= \text{lookup}_{\tau'}^{\text{def}, \text{square_def}}(S_4) = \emptyset \\ &\text{since } \{\pi \in S_4 \mid k(\pi) \leq \text{figure} \text{ and } M(\pi)(\text{def}) = \text{square_def}\} = \emptyset \\ S''' &= \text{lookup}_{\tau'}^{\text{def}, \text{circle_def}}(S_4) \\ &= \delta_\tau(S_4) \cup \left(\bigcup \{ \{\pi\} \cup \delta_{F(\pi)}(S_4) \mid \pi \in S \} \right) = \{\bar{\pi}, \pi_3\} \cup \{\pi_3\} = \{\bar{\pi}, \pi_3\} , \\ &\text{since } S = \{\pi \in S_4 \mid k(\pi) \leq \text{figure}, M(\pi)(\text{def}) = \text{circle_def}\} = \{\pi_3\}. \end{aligned}$$

The abstract operations over \mathcal{E} are strict on \emptyset (Proposition 3). Thus only S''' contributes non-trivially to the static analysis. Namely, the starting state of the selected target method `circle_def` is

$$S_5 = \text{call}_\tau^{\text{circle_def}}[\text{res} \mapsto \text{circle}](S''') = \delta_{[\text{res} \mapsto \text{circle}]}(S''') = \{\pi_3\} .$$

Note that S_i is the *exact* abstraction (Definition 9) of the state σ_i of Example 6 for $i = 1, \dots, 5$.

5 Implementation

We have implemented the domain \mathcal{E} for escape analysis of Section 4 inside a static analyser for simple object-oriented programs, called LOOP [12,13].

Consider the program in Figure 1. The type environment of the input of `main` is $\tau_{in} = [\text{this} \mapsto \text{main}]$, that of its output is $\tau_{out} = [\text{out} \mapsto \text{int}]$. The abstract domain for τ_{in} is $\mathcal{E}_{\tau_{in}} = \{\emptyset, \{\bar{\pi}\}\}$ (Definition 11). LOOP, focused on the watchpoints w_2 and w_3 , concludes that if the input of `main` is \emptyset , which is the abstraction of the empty set of concrete states (Lemma 2), its output is \emptyset and no watchpoints are observed. If the input is $\bar{\pi}$, the output is still \emptyset (no objects can be reached from an integer) *but* the watchpoints w_2 and w_3 are observed. They have the same type environment (Example 1). LOOP concludes that, in w_2 , only

	[2]	[3]	[7]	[11]	[14]	\mathcal{E}
Is the analysis context-sensitive?	Y	Y	N	Y	Y	Y
Is the analysis compositional?	Y	N	Y	Y	Y	Y
Is there a correctness proof?	Y	N	N	N	N	Y
Is there an optimality proof of the operations?	N	N	N	N	N	Y
Is the access to a field somehow approximated?	Y	Y	N	Y	Y	Y
Are virtual calls somehow restricted?	N	N	N	N	N	Y
Can it derive that π_4 is not reachable in w_3 (Figure 1) ?	N	N	N	N	N	Y

Fig. 6. A comparison of different escape analyses

objects created in $\{\bar{\pi}, \pi_1, \pi_2, \pi_4\}$ can be reached. Since the creation point π_4 is internal to the method `main_rotate`, this means, in particular, that the `new angle` in π_4 cannot be stack allocated *for this instantiation of main_rotate*. Instead, LOOP concludes that in w_3 only objects created in $\{\bar{\pi}, \pi_3\}$ are reachable. This time, the `new angle` in π_4 can be stack allocated.

The fact that no objects created in π_4 are reachable in w_3 follows from two properties of our analysis. The first is that it is context-sensitive *i.e.*, it allows us to distinguish what happens in w_2 and in w_3 . The second is that it uses *optimal* abstract operations (Figure 5) which, in particular, can sometimes restrict the target of a virtual call. In our example, since we know that `f` is bound to a `circle` after π_3 , we conclude that the subsequent call `rotate(f)` results in a virtual call `f.rot(a)` which resolves into the method `circle_rot`. That method does not use `a`. Thus, the object stored in `a` does not escape the `rotate(f)` call.

6 Discussion

The first escape analysis for functional languages [10] has been made more efficient in [6] and then extended to some imperative constructs and applied to very large programs [1]. For object-oriented languages, it has been considered in [2,3,7,8,11,14]. In [11], a *lifetime analysis* propagates the *sources* of data structures. In [2] integer *contexts* specify the part of a data structure which can escape. Both [3] and [14] use *connection graphs* to represent the concrete memory, but in [14] those graphs are slightly more concrete than in [3]. In [7] a program is translated to a constraint, whose solution is the set of *escaping* variables.

Here, we have defined the *escape property* \mathcal{E} as a property of concrete states (Definition 11), and shown that it induces a non-trivial analysis, since

- The set of the creation points of the objects reachable from the current state can both grow (`new`) and shrink (δ) *i.e.*, *static type information contains escape information* (Examples 9 and 13);
- That set is useful, sometimes, to restrict the possible targets of a virtual call *i.e.*, *escape information contains class information* (Example 13).

In Figure 6, we compare the escape analyses presented in [2,3,7,11,14] with our own (\mathcal{E}). Here *context-sensitive* means that the analysis can provide different information in different program points; *compositional*, that the analysis of a compound command is defined in terms of the analysis of its components; *approximating the access to a field*, that the analysis can provide a better approximation of the creation points of the objects stored in a field than the set of all creation points; and *restricting the virtual calls*, that the analysis is able to discard some targets of a virtual call which can never be selected at run-time. These two last aspects contribute to the precision of the analysis.

The escape analysis induced by our domain \mathcal{E} is not precise enough for practical use, while those in [2,3,7,11,14] have been developed to that purpose. However, the last line of Figure 6 shows that it is sometimes more precise than those analyses. Since \mathcal{E} coincides with the *escape property* itself, either these other analyses are not a concretisation of this property, or their operations do not fully preserve it. We claim hence that \mathcal{E} makes a good starting point for a formal foundation of escape analysis. More precise domains are needed, but they should be *refinements* of \mathcal{E} *i.e.*, uniformly more precise than \mathcal{E} , which, as we said above, is not the case for For an example of such refinements, see [8].

References

1. B. Blanchet. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages (POPL'98)*, pages 25–37, San Diego, CA, USA, January 1998. ACM Press. 393
2. B. Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(1) of *SIGPLAN Notices*, pages 20–34, Denver, Colorado, USA, November 1999. 388, 393, 394
3. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *SIGPLAN Notices*, pages 1–19, Denver, Colorado, USA, November 1999. 388, 393, 394
4. A. Cortesi, G. Filé, and W. Winsborough. The Quotient of an Abstract Interpretation. *Theoretical Computer Science*, 202(1-2):163–192, 1998. 382, 388, 391
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977. 381, 382
6. A. Deutsch. On the Complexity of Escape Analysis. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 358–371, Paris, France, January 1997. ACM Press. 393
7. D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In D. A. Watt, editor, *Compiler Construction, 9th International Conference, (CC'00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 82–93, Berlin, Germany, March 2000. Springer-Verlag. 388, 393, 394

8. P. M. Hill and F. Spoto. A Refinement of the Escape Property. In A. Cortesi, editor, *Proc. of the VMCAI'02 workshop on Verification, Model-Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 154–166, Venice, Italy, January 2002. 381, 382, 388, 393, 394
9. T. Jensen and F. Spoto. Class Analysis of Object-Oriented Programs through Abstract Interpretation. In F. Honsell and M. Miculan, editors, *Proc. of FOSSACS 2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275, Genova, Italy, April 2001. Springer-Verlag. 380, 382
10. Y. G. Park and B. Goldberg. Escape Analysis on Lists. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92)*, volume 27(7) of *SIGPLAN Notices*, pages 116–127, San Francisco, California, USA, June 1992. 393
11. C. Ruggieri and T. P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *15th ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 285–293, San Diego, California, USA, January 1988. 393, 394
12. F. Spoto. The LOOP Analyser. <http://www.sci.univr.it/~spoto/loop/>. 392
13. F. Spoto. Watchpoint Semantics: A Tool for Compositional and Focussed Static Analyses. In P. Cousot, editor, *Proc. of the Static Analysis Symposium, SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 127–145, Paris, July 2001. Springer-Verlag. 380, 381, 386, 387, 392
14. J. Whaley and M. C. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(1) of *SIGPLAN Notices*, pages 187–206, Denver, Colorado, USA, November 1999. 393, 394

A Framework for Order-Sorted Algebra

John G. Stell

School of Computing, University of Leeds
Leeds, LS2 9JT, U. K.
jgs@comp.leeds.ac.uk

Abstract. Order-sorted algebra is a generalization of many-sorted algebra obtained by having a partially ordered set of sorts rather than merely a set. It has numerous applications in computer science. There are several variants of order sorted algebra, and some relationships between these are known. However there seems to be no single conceptual framework within which all the connections between the variants can be understood. This paper proposes a new approach to the understanding of order-sorted algebra. Evidence is provided for the viability of the approach, but much further work will be required to complete the research programme which is initiated here.

The programme is based on the investigation of two topics. Firstly an analysis of the various categories of order-sorted sets and their relationships, and, secondly, the development of abstract notions of order-sorted theory, as opposed to presentations given by a signature of operation symbols. As a first step, categories of order-sorted sets are described, adjunctions between the categories are obtained, and results on order-sorted theories as categories, in the sense of Lawvere, are obtained.

1 Introduction

1.1 Order-Sorted Algebra in Computer Science

Formal systems in Computer Science often make use of sorts or types to structure the universe of discourse. In what is usually called ‘many-sorted’ algebra, the sorts form a set, but do not support additional structure. Many-sorted algebra and logic has been widely applied in Computer Science; some of these applications are described in the book [MT93].

A generalization of many-sorted algebra is obtained by taking the set of sorts to be partially ordered. This generalization is known as order-sorted algebra. Order-sorted algebra is generally applied where the the universe of discourse supports an hierarchical structure. Order-sorted concepts have been widely used in Computer Science. The following list gives a few examples of applications.

Computer Algebra. Hearn and Schrufer [HS95] have used order-sorted algebra as the underlying model in the design of a computer algebra system.

Object Orientation. Several approaches [Wie91, AG94] to a formal semantics for the object-oriented programming paradigm rely on a partially ordered set of sorts to express inheritance between classes.

Spatial Databases. Erwig and Güting’s data model for spatial data [EG94] is specified in a multi-level order-sorted algebra.

Formal Methods Algebraic Specification with a poset of sorts, to deal with partially defined operations, was one of the original motivations for order-sorted algebra [Gog78]. Subsequent research has included work on parameterization [Poi90, Qia94], and on the incorporation of higher-order functions [Hax95].

Logic Programming. The language PROTOS(L), has an operational semantics based on polymorphic order-sorted resolution [BM94].

1.2 Variants of Order-Sorted Algebra

The early paper by Oberschelp [Obe62] defines sorts by unary predicates on an unsorted universe, and defines subsorts by implications among these predicates. However, it was the work of Goguen [Gog78] which introduced the concept of order-sorted algebra and applied it to problems in computation. Since then the theory has been developed by several authors, and the key papers include [Wal92, GM92, SS89, Poi90]. Note that Robinson [Rob94] uses ‘order-sorted’ in a quite different sense.

There are several distinct versions of order-sorted algebra, and individual variants have been studied in detail, there is no satisfactory coherent account of all the relationships between the various approaches. The multiplicity of forms is illustrated in Figure 1, which is taken from Goguen and Diaconescu’s survey [GD94]. This survey is useful, but the relationships shown in Figure 1 rely on descriptions which take the form of a particular syntactic presentation, and then a semantics based on this syntax for each of the variants. Taking such an approach does not provide presentation independent descriptions, which is an important aspect of the framework in the present paper. Another approach to understanding order-sortedness is via membership equational logic [BJM00], which allows the representation of various versions of order-sorted algebra within its formalism. The following section proposes a programme for understanding order-sorted algebra which is alternative to both these approaches.

2 A Proposed Framework for Order-Sorted Algebra

It is the contention of this paper that to provide a framework for understanding all the various forms of order-sorted algebra two issues need to be addressed.

1. It is necessary to understand the various possible categories of order-sorted sets and their relationships.
2. It is necessary to have a notion of theory, in the presentation independent sense, for order-sorted algebra.

These two issues are discussed in more detail below. Later sections of the paper make a substantial contribution to carrying out the proposed programme.

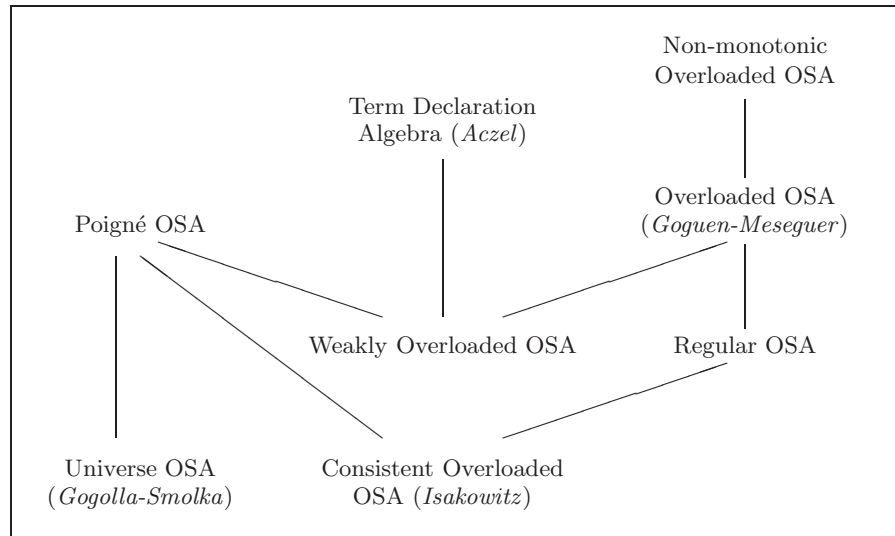


Fig. 1. Goguen and Diaconescu's view of order-sorted algebra

2.1 Categories of Order-Sorted Sets

In unsorted, or single-sorted, algebra, the interpretation of a signature consists of a carrier set for the algebra, and functions corresponding to the function symbols in the signature. This situation has a straightforward generalization to the many-sorted case, but the extension to order-sorted algebra is substantially more complex. The complications are due to the fact that there are several different choices for a category of order-sorted sets whose objects are carriers for order-sorted algebras. This multiplicity of categories may explain why several variants of order-sorted algebra have arisen, and section 4 below provides a comprehensive treatment of the various categories.

2.2 Order-Sorted Theories

In unsorted algebra, the notion of a theory as opposed to a particular presentation of a theory has proved useful. Category theory provides two related approaches to this viewpoint: monads and Lawvere theories. Neither has been fully explored for all the variants of order-sorted algebra. Moving to this level of abstraction loses the concrete signature. However, this is appropriate when the aim, as in this paper, is to study the nature of order-sortedness rather than to apply it. For programming or specification a concrete syntax is necessary.

Lawvere Theories In Lawvere's approach, a theory is a category with certain structure, and its algebras are structure preserving functors to a suitable category. For equational logic the theory category has products, moving beyond

equational logic leads to a hierarchy of categories with more structure [AR94] which provide presentation independent descriptions of theories for richer logics.

Section 6 provides a generalization of algebraic theories by using a suitably enriched notion of product. These poset enriched theories, having total local products in the sense of Jay [Jay90] are adequate for capturing at least one of the forms of order-sorted algebra identified by Goguen and Diaconescu. The algebras for the weakly overloaded case are functors to the category of sets and partial functions. The strongly overloaded approach is more complicated, and some further work is needed.

There has been some other work which is relevant to the quest for the appropriate notion of algebraic theory in the order-sorted case. It is known [GM92] that any variety of strongly overloaded order-sorted algebras is equivalent to a quasi-variety of many-sorted algebras. This might appear to suggest that presentation-independent approaches to order-sorted algebra are already known via such approaches to quasi-varieties of many-sorted algebras, for example [Kea75]. However, this analysis can only be applied immediately to the strongly overloaded case. It also fails to provide a sharp characterization of varieties of order-sorted algebras, as not every quasi-variety of many-sorted algebras is equivalent to a variety of order-sorted algebras.

In Aczel's paper on term declaration logic [Acz91], finitary generalized composita are suggested as the appropriate analogue of algebraic theories for this form of order-sorted algebra, but it is admitted that further work is needed to verify this. The question of whether finitary generalized composita can be used for other forms of order-sorted algebra has not been considered.

Monads An alternative approach to presentation independent theories is provided by the notion of a monad. In [Ste91], I showed how the versions of order-sorted algebra used in Schmidt-Schaß [SS89] and in [MGS89] could be related by mapping each to a morphism between two monads. One of the monads is on **Set**, and the other in the lax slice category $\mathbf{Set} // S$, which is defined in section 4.2 below. Poigné [Poi90] does mention that one category of order-sorted algebras is algebraic over the category denoted **Ext** below, but it is not clear whether all forms of order-sorted algebra can be described as monads.

3 Three Forms of Order-Sorted Algebra

Before commencing the proposed framework, we review details of some forms of order-sorted algebra. The aim of this section is to identify certain categories which arise in order-sorted algebra. We will use (S, \leq) , or just S , to denote a partially ordered set of sorts.

3.1 The Categories **Ext** and **Comp**

An (S, \leq) -sorted set is defined to be a family of sets A_s indexed by S such that $s_1 \leq s_2 \Rightarrow A_{s_1} \subseteq A_{s_2}$. If A is an S -sorted set, \overline{A} will denote $\bigcup_{s \in S} A_s$. For $a \in \overline{A}$, the set of all sorts of a , i.e. $\{s \in S \mid a \in A_s\}$, will be denoted by $\mathbf{sorts}(a)$.

Morphisms between S -sorted sets can be defined in various ways. A **compatible morphism** $\varphi : A \rightarrow B$ is a family of functions $\varphi_s : A_s \rightarrow B_s$ such that if $s \leq t$, then φ_s and φ_t agree on A_s . This gives the category **Comp**, of (S, \leq) -sorted sets and compatible morphisms. A compatible morphism $\varphi : A \rightarrow B$ is **extensible** if whenever $a \in A_s \cap A_t$ then $\varphi_s(a) = \varphi_t(a)$. The category **Ext** of (S, \leq) -sorted sets and extensible morphisms is a subcategory of **Comp**.

3.2 Signatures and Algebras

An (S, \leq) -sorted signature Σ is a family of sets $\Sigma_{w,s}$ indexed by $S^* \times S$. This one notion of signature gives rise to two notions of algebra. These differ in the semantics given to overloaded function symbols. For example, suppose we have a function symbol $\sigma : v \rightarrow s$ and $\sigma : w \rightarrow t$. In **extensible overloading** we require that $A_{\sigma_{v,s}}$ and $A_{\sigma_{w,t}}$ agree on $A_v \cap A_w$. In **compatible overloading** we require that if $v \leq w$ and $s \leq t$ then $A_{\sigma_{v,s}}$ and $A_{\sigma_{w,t}}$ agree on A_v . Goguen and Diaconescu [GD94] use the terms ‘weak’ and ‘strong’ rather than ‘extensible’ and ‘compatible’ respectively.

A **compatible algebra** for Σ is an (S, \leq) -sorted set, A , and for each $\sigma \in \Sigma_{w,s}$, a function $A_{\sigma_{w,s}} : A_w \rightarrow A_s$ such that

$$\left. \begin{array}{l} \sigma \in \Sigma_{v,s} \cap \Sigma_{w,t} \\ (v, s) \leq (w, t) \\ \mathbf{a} \in A_v \end{array} \right\} \Rightarrow A_{\sigma_{v,s}}(\mathbf{a}) = A_{\sigma_{w,t}}(\mathbf{a}).$$

An **extensible algebra** for Σ is an (S, \leq) -sorted set, A , and for each $\sigma \in \Sigma_{w,s}$, a function $A_{\sigma_{w,s}} : A_w \rightarrow A_s$ such that

$$\left. \begin{array}{l} \sigma \in \Sigma_{v,s} \cap \Sigma_{w,t} \\ \mathbf{a} \in A_v \cap A_w \end{array} \right\} \Rightarrow A_{\sigma_{v,s}}(\mathbf{a}) = A_{\sigma_{w,t}}(\mathbf{a}).$$

By defining a homomorphism $h : A \rightarrow B$ to be a morphism in the appropriate category such that $h_t(A_{\sigma_{w,t}}(\mathbf{a})) = B_{\sigma_{w,t}}(h_w(\mathbf{a}))$, we obtain the categories **AlgComp** $_{\Sigma}$ and **AlgExt** $_{\Sigma}$.

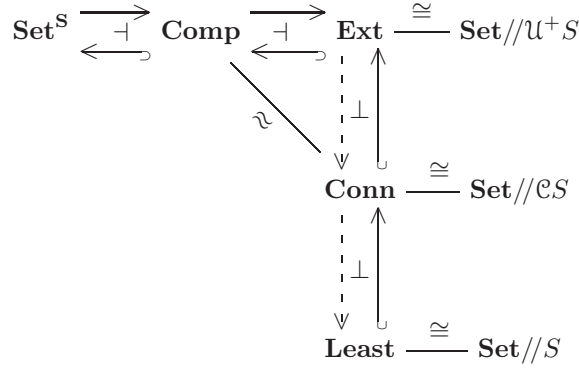
3.3 Least Sorts

A third category of order-sorted sets appears implicitly in the literature. An S -sorted set A **has least sorts** if for any $a \in \overline{A}$ the set $\text{sorts}(a)$ has a least element. The category **Least** has least sort S -sorted sets for its objects, and compatible morphisms between these. A compatible morphism between least sort S -sorted sets is necessarily extensible, so the least sort S -sorted sets determine the same full subcategory of **Ext** and of **Comp**.

The signatures considered in [MGS89] satisfy a technical condition which implies that every term has a least sort. Thus the category **Least** is the appropriate setting here.

4 Relating Categories of Order Sorted Sets

The previous section identified three categories where the objects are order-sorted sets which can be used as carriers for order-sorted algebras. The relationships between these categories, and other related ones, have not been identified before. The following diagram summarizes the results reported in this section.



In the diagram, $\xrightarrow{\quad \dashv \quad}$ indicates an inclusion having a left adjoint. In two cases, indicated by dashed arrows, the left adjoint exists only when S satisfies certain conditions. These conditions are detailed in theorems 4 and 5. The symbol \cong denotes an isomorphism of categories, and \approx an equivalence.

Informally, the diagram can be interpreted as follows. It is well known that sets, or other objects, indexed by some kind of structure admit two types of descriptions: one in a pre-sheaf fashion and one in a bundle like way. For order-sorted sets these two extremes appear in the top line of the diagram at the left hand end and the right hand end respectively. The right-hand column of the diagram shows that three different categories of order-sorted sets can be described in a bundle-like way as lax slice categories over different categories.

4.1 Relationships between Categories

The poset (S, \leq) can be seen as a category, \mathbf{S} , having objects the elements of S , and there being a morphism from s to t iff $s \leq t$. With this viewpoint, \mathbf{Comp} is the full subcategory of the functor category $\mathbf{Set}^{\mathbf{S}}$, determined by those $F : \mathbf{S} \rightarrow \mathbf{Set}$ where if $s \leq t$ then the function $Fs \rightarrow Ft$ is an inclusion.

Theorem 1 *The inclusion of \mathbf{Comp} in $\mathbf{Set}^{\mathbf{S}}$ has a left adjoint.*

Proof The left adjoint, $\Gamma : \mathbf{Set}^{\mathbf{S}} \rightarrow \mathbf{Comp}$ is defined on an object F by taking the colimiting cocone $i : F \rightarrow \text{colim } F$ and defining $(\Gamma F)s = \{i_s(x) \in \text{colim } F \mid x \in Fs\}$. It is routine to verify that Γ is left adjoint to the inclusion. \square

Definition 1 *An S -sorted set, A , has **connected sorts** if for all $a \in \overline{A}$, $\text{sorts}(a)$ is a connected subset of S . The category of S -sorted sets having connected sorts and compatible morphisms will be denoted \mathbf{Conn} .*

A compatible morphism between S -sorted sets having connected sorts is necessarily extensible. Thus there are inclusions $\mathbf{Conn} \hookrightarrow \mathbf{Ext} \hookrightarrow \mathbf{Comp}$. The first of these is full but not the second.

For any object A of \mathbf{Comp} , ΓA has connected sorts. Using this fact, we can obtain the following result.

Theorem 2 *There is an equivalence between \mathbf{Comp} and \mathbf{Conn} .* \square

By considering the composite $\mathbf{Comp} \xrightarrow{\cong} \mathbf{Conn} \hookrightarrow \mathbf{Ext}$, we can establish the next theorem.

Theorem 3 *The inclusion of \mathbf{Ext} into \mathbf{Comp} has a left adjoint.* \square

Next we turn to conditions under which the inclusions $\mathbf{Conn} \hookrightarrow \mathbf{Ext}$, and $\mathbf{Least} \hookrightarrow \mathbf{Conn}$ have left adjoints. Recall that an **upper set** in S is a subset $X \subseteq S$ for which $x \in X$ and $x \leq s \in S$ imply $s \in X$. The smallest upper set containing X is denoted X^\uparrow . The set of all connected upper sets which contain X will be denoted $\mathbf{upconn}(X)$. If the set $\mathbf{upconn}(X)$ has a least element it will be denoted $X^{\text{c}\uparrow}$. When the set $\mathbf{upconn}(X)$ has a least element for every X such that $\emptyset \neq X \subseteq S$, we will say that S is **up-connected**.

Theorem 4 *The inclusion $\mathbf{Conn} \hookrightarrow \mathbf{Ext}$ has a left adjoint if and only if S is up-connected.*

Proof Suppose S is up-connected. Define $F : \mathbf{Ext} \rightarrow \mathbf{Conn}$ on objects by $\overline{FA} = \overline{A}$, and for all $a \in \overline{A}$, $\mathbf{sorts}_{FA}(a) = (\mathbf{sorts}_A(a))^{\text{c}\uparrow}$. Thus for each s , $A_s \subseteq (FA)_s$.

To define F on morphisms, suppose $\varphi : A \rightarrow B$ is extensible, and define $F\varphi$ by $\overline{F\varphi} = \overline{\varphi}$. To justify this we need to check that if $a \in (FA)_s$, then $\varphi(a) \in (FB)_s$. Note that in general if $X \subseteq Y$ then $X^{\text{c}\uparrow} \subseteq Y^{\text{c}\uparrow}$. Thus from $\mathbf{sorts}_A(a) \subseteq \mathbf{sorts}_B(\varphi(a))$, we get $\mathbf{sorts}_{FA}(a) \subseteq \mathbf{sorts}_{FB}(\varphi(a))$ as required. It is straightforward to verify the remaining details.

Conversely, suppose S is not up-connected. If there is a left adjoint, F , we will have $\mathbf{Ext}(A, K) \cong \mathbf{Conn}(FA, K)$ for any objects A of \mathbf{Ext} and K of \mathbf{Conn} . Let $X \subseteq S$ have the property that the intersection of all the elements of $\mathbf{upconn}(X)$ is not connected. The upward closure of X , X^\uparrow , cannot be connected. Let A be the S -sorted set where $\overline{A} = \{a\}$ and $\mathbf{sorts}(a) = X^\uparrow$. Thus $A_s = \emptyset$ for $s \notin X^\uparrow$.

Suppose first that $\mathbf{upconn}(X)$ is non-empty. For any $Y \in \mathbf{upconn}(X)$ define Y^\dagger to be the object of \mathbf{Conn} where $\overline{Y^\dagger} = \{a\}$ and $\mathbf{sorts}(a) = Y$. There is exactly one morphism in \mathbf{Ext} from A to Y^\dagger , so $\mathbf{Conn}(FA, Y^\dagger)$ also contains exactly one morphism. Thus if $Y_s^\dagger = \emptyset$, we must have $(FA)_s = \emptyset$. Since this holds for all $Y \in \mathbf{upconn}(X)$, $(FA)_s$ can only be non-empty when s belongs to the intersection of all the elements of $\mathbf{upconn}(X)$. Thus either $\overline{FA} = \emptyset$, or $\{s \in S \mid (FA)_s \neq \emptyset\}$ is not connected. Now, as FA is an object of \mathbf{Conn} , if $\overline{FA} \neq \emptyset$ then \overline{FA} contains two distinct elements. In either case we can conclude that $\mathbf{Ext}(A, FA) = \emptyset \neq \mathbf{Conn}(FA, FA)$.

If $\mathbf{upconn}(X)$ is empty, we can find $s, t \in X$ lying in distinct connected components of S . As FA is an object of \mathbf{Conn} , $(FA)_s \cap (FA)_t = \emptyset$. Thus there

can be no extensible morphism from A to FA , so again we have $\mathbf{Ext}(A, FA) \not\cong \mathbf{Conn}(FA, FA)$. \square

The next result is relevant to order-sorted unification. Viewing S as a category, $X \subseteq S$ determines a diagram in this category and a limit for this diagram is exactly a meet for X . Next we see how X gives rise to a diagram in \mathbf{Least} . Regarding X as a category and X^{op} being its opposite, the functor $X^* : X^{op} \rightarrow \mathbf{Least}$ is defined by making $X^*(x)$ the S -sorted set having just the element x having least sort x .

A cocone on X^* is (up to isomorphism) a set containing a lower bound for each connected component of X . A colimit for X^* is a set consisting of a meet for each connected component of X .

Theorem 5 *The inclusion $\mathbf{Least} \hookrightarrow \mathbf{Conn}$ has a left adjoint if and only if every connected subset of S has a meet.*

Proof Suppose S has connected meets. Define $\Delta : \mathbf{Conn} \rightarrow \mathbf{Least}$ by $\overline{\Delta A} = \overline{A}$ and $\text{sorts}_{\Delta A}(a) = (\bigwedge \text{sorts}_A(a))^\uparrow$. It is straightforward to check that Δ has the appropriate properties.

Conversely, if there is a left adjoint then \mathbf{Least} has colimits since \mathbf{Set}^S has colimits and these are preserved by the left adjoints defined earlier. But then for any connected set X we have a colimit for X^* and hence a meet for X . \square

In [MGS89] conditions are given under which an order-sorted signature has a minimal set of unifiers. In the single sorted case there is a well known [RS87] description of a most general unifier as a coequalizer. In [Ste92] I showed how one form of order-sorted unification could be described as a generalization of the notion of coequalizer. Theorem 5 leads to conditions under which \mathbf{Least} may lack coequalizers, and with further work, should provide an account of the generalizations of coequalizers which describe minimal sets of order-sorted unifiers.

4.2 Description as Lax Slice Categories

The categories \mathbf{Ext} , \mathbf{Conn} , and \mathbf{Least} all admit alternative descriptions as lax slice categories. If (R, \leq) is any poset, then we can construct the lax slice category $\mathbf{Set} // R$. An object of this category consists of a set, X , and a function $\alpha : X \rightarrow R$. Such an object will be denoted X_α . A morphism $f : X_\alpha \rightarrow Y_\beta$ is a function $f : X \rightarrow Y$ such that $\beta f x \leq \alpha x$ for all $x \in X$. The three categories just mentioned are all (up to isomorphism) categories of this form for suitable choice of the poset R .

It is straightforward to verify that there is an isomorphism $\mathbf{Least} \rightarrow \mathbf{Set} // S$, which sends a least-sorts S -sorted set, A , to \overline{A}_α where α takes each $a \in \overline{A}$ to its least sort. To describe \mathbf{Ext} as a lax slice category, note that the set of sorts of an arbitrary S -sorted set, A , will be a non-empty upper set of S . Let $\mathcal{U}^+ S$ denote the poset of non-empty upper sets of S ordered oppositely to inclusion. Then

there is an isomorphism of categories $\mathbf{Ext} \cong \mathbf{Set} // \mathcal{U}^+ S$. Using \mathcal{CS} to denote the subposet of $\mathcal{U}^+ S$ determined by the connected non-empty upper sets, we obtain the isomorphism $\mathbf{Conn} \cong \mathbf{Set} // \mathcal{CS}$.

Note that the poset of all upper sets of S ordered oppositely to inclusion is actually the free meet semilattice on the poset S . Thus \mathbf{Ext} can be described as the category of least sort S^\wedge -sorted sets, where S^\wedge is the completion of S to allow for non-empty meets of sorts. There are also some interesting connections with sheaves and topology. Recall that a poset S can be given the Alexandroff topology by taking the open sets to be the upper sets of S . The functor category \mathbf{Set}^S is known [Gol84, p366] to be the category of sheaves over this topological space. The full picture of how this relates to the usual connection between sheaves and bundles needs some further work.

5 The Connection with Partial Algebras

This section shows how extensible algebras are a kind of partial algebra. Some basic facts about the category of sets and partial functions which we need are reviewed.

5.1 Extensible Algebras as Partial Algebras

An order sorted signature Σ can be forgotten to an unsorted signature $|\Sigma|$. An extensible Σ -algebra, A , can be forgotten to a partial $|\Sigma|$ -algebra with carrier $\bar{A} = \bigcup_{s \in S} A_s$. From this viewpoint, the elements of S serve to name subsets of \bar{A} , and the partial order in S specifies certain inclusions between these named subsets. The extensible Σ -algebras are thus partial $|\Sigma|$ -algebras where the domains of the operations are related by certain specified inclusions.

5.2 The Category of Sets and Partial Functions

Recall that an ordered category is a category enriched in the category of posets and monotone maps. Thus the morphisms between any two objects form a poset, and composition of morphisms respects this order. A key example of an ordered category is \mathbf{PFn} , the category of sets and partial functions. Given partial functions f and g from X to Y we put $f \leq g$ iff for all $x \in X$ if $f(x)$ is defined then $g(x)$ is defined, and $f(x) = g(x)$. In diagrams it is common to indicate that $f \leq g$ by drawing an arrow \Rightarrow from f to g . In \mathbf{PFn} a subset, $A \subseteq X$ is just a morphism $A : X \rightarrow X$ such that $A \leq \text{id}_X$.

The next observation plays an important role in expressing the sorting information in an order-sorted signature in a categorical way. The condition that f be a partial function defined on a superset of A and producing results in B from arguments in A is equivalent to the existence of a 2-cell as in the following diagram, where $\{*\}$ is a one-element set, and the morphisms to this set labelled $*$

are total functions.

$$\begin{array}{ccc}
 X & \xrightarrow{f} & Y \\
 A \downarrow & \Rightarrow & \downarrow B \\
 X & \xrightarrow{*} \{*\} \xleftarrow{*} & Y
 \end{array}$$

5.3 Example of Theory as Category

This simple example explains the basic idea behind the construction of the the next section. The aim is that given a signature Σ , we should be able to construct a category \mathbb{T} . The category should have certain structure, and functors from \mathbb{T} to \mathbf{PFn} which preserve the structure should be essentially the same as extensible algebras for Σ . This category is sometimes called the **classifying category** of Σ [Cro93].

Suppose we have sorts $B \leq A$, and a function symbol f with sortings $f : A \times A \rightarrow A$, and $f : B \times B \rightarrow B$. It appears we need some form of ordered category, which includes the following diagrams.

$$\begin{array}{ccc}
 \begin{array}{ccc}
 \xrightarrow{\text{id}_1} & & \\
 A \uparrow & \uparrow & \\
 1 \xrightarrow{\quad} 1 & & \\
 B \uparrow & \uparrow & \\
 \xrightarrow{\quad} & &
 \end{array} & \Rightarrow &
 \begin{array}{ccc}
 1 \times 1 \xrightarrow{!} 0 \xleftarrow{!} 1 & & \\
 \uparrow A \times A & \Rightarrow & A \uparrow \\
 1 \times 1 \xrightarrow{f} 1 & & \\
 \downarrow B \times B & \Rightarrow & B \downarrow \\
 1 \times 1 \xrightarrow{!} 0 \xleftarrow{!} 1 & &
 \end{array}
 \end{array}$$

If we consider what structure the category should support, it is clear that, unlike the single sorted case, 0 cannot be a terminal object and \times cannot be a product. The category \mathbf{PFn} does have a terminal object, but it is the empty set. There are also products in \mathbf{PFn} , but again these are not the structure we need. The product of A and B in \mathbf{PFn} is actually $A + (A \times B) + B$, where \times and $+$ are the product and coproduct respectively in \mathbf{Set} .

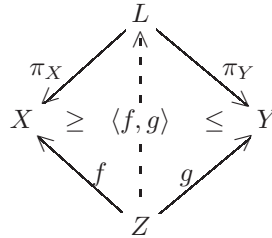
6 The Classifying Category of an Order-Sorted Theory

6.1 Ordered Categories with Total Local Products

We shall see that the appropriate structure on an extensible order-sorted theory *qua* category is that of an ordered category with total local products. In this section we review the definitions, which are due to Jay [Jay90]. The definitions assume the context of an ordered category \mathbb{O} .

Definition 2 A morphism $\delta : X \rightarrow X$ in \mathbb{O} is a **deflation** if $\delta^2 = \delta \leq id_X$. A morphism $f : Y \rightarrow Z$ is **total** if for every $g : X \rightarrow Y$, and for every deflation $\delta : X \rightarrow X$, the equality $fg\delta = fg$ implies $g\delta = g$.

Definition 3 A *local product* of objects X, Y is an object L and morphisms, π_X, π_Y as in the diagram below. Furthermore, given any f, g as in the diagram, there is a morphism $\langle f, g \rangle$ which is maximal among all morphisms $k : Z \rightarrow L$ such that $\pi_X k \leq f$ and $\pi_Y k \leq g$.



Definition 4 An object L of \mathbb{O} is a *local terminal object* if for any object X there is a morphism $\ell : X \rightarrow L$, and $k : X \rightarrow L \Rightarrow k \leq \ell$.

These forms of local limit are very weak notions. In **PFn** both \emptyset and 1 are locally terminal, and any subset of $X \times Y$ is a local product. However, total local limits are unique up to isomorphism.

Definition 5 A local product is *total* if the projections are total and whenever f and g are total, then $\pi_X \langle f, g \rangle = f$ and $\pi_Y \langle f, g \rangle = g$. A local terminal object is *total* if $\ell : X \rightarrow L$ is always total.

In **PFn**, the total product is $X \times Y$ as in **Set** with $\langle f, g \rangle$ defined on the intersection of domains of definition of f and g . Any one element set is a total terminal object.

6.2 Construction of the Classifying Category from a Signature

In this section the construction of a category from a signature Σ will be considered. The use of presentations consisting of a signature together with equations will be dealt with in the following section.

Essentially, the category we seek will be the free ordered category with total local products, generated diagrams derived from Σ as in the example in section 5.3. However, an explicit syntactic construction can be given, just as for the single sorted and many-sorted cases [Cro93]. Although order-sorted algebra can be seen as an extension of many-sorted algebra, several aspects of the following construction are closer to the single-sorted than the many-sorted case.

In the single sorted case, the objects of the category are the natural numbers $0, 1, 2, \dots$, and a morphism $m \rightarrow n$ is an equivalence class of n -tuples of terms constructed from m variables. In the case of extensible order-sorted algebra the main complication arises from morphisms $m \rightarrow 0$. Since such morphisms are to be interpreted as partial functions, it is not the case that there is a unique morphism $m \rightarrow 0$ which can be represented by the 0-tuple of terms.

First we construct a signature $\tilde{\Sigma}$ which extends the unsorted signature $|\Sigma|$ by the addition of two kinds of function symbol. Each $f : s_1 \times \dots \times s_n \rightarrow s$ in

Σ , where $n \geq 0$, gives rise to $f : n \rightarrow 1$ in $\tilde{\Sigma}$. The signature $\tilde{\Sigma}$ has, function symbols which represent subsets. If S^\wedge denotes the completion of S to allow for meets of non-empty finite subsets of S , then $\tilde{\Sigma}$ has, for each $s \in S^\wedge$ a function symbol $s : 1 \rightarrow 1$.

The signature $\tilde{\Sigma}$ also contains for each $n \geq 0$, a function symbol $*^n : n \rightarrow 0$. The significance of the symbols $*^n$ is that when the category we are constructing is interpreted in \mathbf{Pfn} , with 1 interpreted as the set H , the morphism $*^n : n \rightarrow 0$ will be interpreted as the unique total function from H^n to the one element set H^0 . Non-total functions to H^0 will arise by composition with $*^n$.

Terms in $\tilde{\Sigma}$ require variables of two types: 0 and 1. It is assumed that variables z, z_1, z_2, \dots , of type 0, and variables x, x_1, x_2, \dots , of type 1 are available. Other terms are formed according to the following rules. In these rules k will be a constant in Σ , and f will either be a function symbol in Σ (the case of $n = 1$), or $*^m$, in which case $n = 0$.

$$\frac{f : m \rightarrow n \quad t_i : 1 \quad i = 1, \dots, m \quad m > 0}{f(t_1, \dots, t_m) : n} \quad \frac{k : 0 \rightarrow 1 \quad t : 0}{k(t) : 1}$$

The appearance of terms which contain variables such as $k(z)$ when k is a constant in Σ may appear strange. This construction is necessary to cope with partially defined functions. For example, suppose that 1 is interpreted as the set H , the sort s is interpreted as $A \subseteq H$, and the constant k as $h \in H$. The term $k(*^1(s(x)))$ is interpreted as the function from H to H sending elements of A to h , and undefined outside A .

Substitution is defined in the usual way, though only terms of type i can be substituted for variables of i , where i can be 0 or 1.

Certain equations between terms are required. Any term containing two sub-terms $A(x_1)$ and $B(x_1)$ is equal to the term with both replaced by $A \wedge B(x_1)$, where $A \wedge B$ is the meet of A and B in S^\wedge . It is also necessary to have $A(B(x_1)) = A \wedge B(x_1)$, thus intersection of subsorts appears as composition in the category. In addition to these equalities, and substitution instances of them, certain equalities needed in the unsorted case have to be imposed. The reader should be able to supply the details by following the account of the unsorted case in [Cro93, section 3.8].

Using this equivalence, a morphism $m \rightarrow n$ is an equivalence class of expressions of the form $\Gamma; \mathbf{t}$, where Γ is a context, consisting of a list of variables, and \mathbf{t} is a list of terms. When $m = 0$, Γ will be a single variable of type 0, and when $m > 0$, Γ will be a list of m distinct variables of type 1. When $n = 0$, \mathbf{t} will be a single term of type 0, and when $n > 0$, \mathbf{t} will be an n -tuple of terms of type 1. In both cases, the terms in \mathbf{t} are built from the variables in Γ . Composition of morphisms is defined using substitution.

The order on the hom sets is generated by adding the sort constraints. For a function symbol $f : s_1 \times \dots \times s_n \rightarrow s$ in Σ we require

$$*^n(s_1(x_1), \dots, s_n(x_n)) \leq *^1(s(f(x_1, \dots, x_n))).$$

Note that this is just the general case of one of the diagrams in the example in section 5.3. For a constant $k : s$ in Σ we require $*^0(z) \leq *^1(s(k(z)))$. And for

each subsort relation $s_1 \leq s_2$ between elements of S^\wedge we require $s_1(x) \leq s_2(x)$. Further relations are generated by substitution.

The above construction yields an ordered category with total local products which is denoted $Cl(\Sigma)$. In $Cl(\Sigma)$ the object n is the n -fold total local product of 1 with itself.

6.3 Presentations Including Equations

To introduce equations, we need to recall the construction of order-sorted terms from a signature Σ . Given Σ , and an S -sorted set of variables, X , we can construct an extensible algebra, having as carrier the S -sorted set of terms $T_\Sigma(X)$. Details can be found in [Gog78] and in [Poi90, p235]. An equation is then an expression of the form $[X]t =_s t'$ where $t, t' \in (T_\Sigma(X))_s$ for some $s \in S$, and every variable in t or t' occurs in X . An algebra A in \mathbf{AlgExt}_Σ is said to **satisfy** the equation $[X]t =_s t'$, if $h_s(t) = h_s(t')$ for every homomorphism $h : T_\Sigma(X) \rightarrow A$. The category of all extensible algebras satisfying every equation in a presentation P will be denoted \mathbf{AlgExt}_P .

A set of equations of this form can be represented by parallel pairs in the category $Cl(\Sigma)$. By constructing the quotient category we obtain the classifying category $Cl(P)$ for an order-sorted presentation P .

6.4 Extensible Models of $Cl(P)$

Given an order-sorted presentation $P = (\Sigma, E)$, we can construct its classifying category $Cl(P)$ as outlined above. In this section we define models of $Cl(P)$ as certain functors to \mathbf{PFn} , and compare the category of these models with the category \mathbf{AlgExt}_P .

Definition 6 *Let \mathbb{T} be an ordered category with total local products. An **extensible model** of \mathbb{T} is a lax ordered functor to \mathbf{PFn} which preserves the total local products. A **morphism of extensible models** is a lax natural transformation between the functors. The category of extensible models and morphisms will be denoted $\mathbf{ModExt}_\mathbb{T}$.*

An operation symbol can be interpreted in a model by a partial function which is defined more widely than it is required to be. This accounts for the weakened sense of equivalence in the next theorem. Possibly this result is weaker than expected, which might suggest that further work could find an improved notion of functorial model.

Theorem 6 *The categories $\mathbf{ModExt}_{Cl(P)}$ and \mathbf{AlgExt}_P are equivalent in the following weakened sense. There are functors $A : \mathbf{ModExt}_{Cl(P)} \rightarrow \mathbf{AlgExt}_P$, and $M : \mathbf{AlgExt}_P \rightarrow \mathbf{ModExt}_{Cl(P)}$ where AM is the identity on \mathbf{AlgExt}_P . The composite MA is related to the identity on $\mathbf{ModExt}_{Cl(P)}$ by natural transformations $\eta : I \rightarrow MA$ and $\theta : MA \rightarrow I$, where I is the identity functor on $\mathbf{ModExt}_{Cl(P)}$. These transformations satisfy the properties that $\eta\theta = id_{MA}$, and $\theta\eta \leq id_I$, where id_F denotes the identity natural transformation on the functor F .*

7 Conclusions and Further Work

This paper has advocated, and presented some initial results in, a new research programme for understanding the various forms of order-sorted algebra. Specifically, an analysis of the various categories of order-sorted sets has been presented, and an enriched form of algebraic theory to model extensible order-sorted algebra has been identified.

Much more work is required to complete the programme. The next step should be a treatment of compatible algebras along the lines of that for extensible algebras. Another area for investigation is a categorical treatment of order-sorted unification and rewriting which extends the single-sorted case developed in [RS87].

Acknowledgements

I am grateful to Joseph Goguen for his interest in this work and his suggestions. The anonymous referees also made useful comments. The categorical diagrams have been drawn using Paul Taylor's macros.

References

- [Acz91] P. Aczel. Term declaration logic and generalized composita. In *6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam, July 1991*, pages 22–30. IEEE Computer Society, 1991. 399
- [AG94] A. J. Alencar and J. A. Goguen. Specification in OOZE with examples. In K. Lano and H. Haughton, editors, *Object-Oriented Specification Case Studies*, chapter 8, pages 158–183. Prentice-Hall, 1994. 396
- [AR94] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press, 1994. 399
- [BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000. 397
- [BM94] C. Beierle and G. Meyer. Run-time type computations in the Warren abstract machine. *Journal of Logic Programming*, 18(2):123–148, 1994. 397
- [Cro93] R. L. Crole. *Categories for Types*. Cambridge University Press, 1993. 405, 406, 407
- [EG94] M. Erwig and R. H. Güting. Explicit graphs in a functional model for spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):787–804, 1994. 397
- [GD94] J. A. Goguen and R. Diaconescu. An Oxford survey of order-sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994. 397, 400
- [GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992. 397, 399

- [Gog78] J. A. Goguen. Order sorted algebras: Exceptions and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Report 14, University of California at Los Angeles. Computer Science Department, December 1978. [397](#), [408](#)
- [Gol84] R. Goldblatt. *Topoi, The Categorical Analysis of Logic*, volume 98 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984. First edition 1979. [404](#)
- [Hax95] A. E. Haxthausen. Order-sorted algebraic specifications with higher-order functions. In V. S. Alagar and M. Nivat, editors, *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, July 1995*, volume 936 of *Lecture Notes in Computer Science*, pages 133–151. Springer-Verlag, 1995. [397](#)
- [HS95] A. C. Hearn and E. Schrufer. A computer algebra system based on order-sorted algebra. *Journal of Symbolic Computation*, 19(1–3):65–77, 1995. [396](#)
- [Jay90] C. B. Jay. Extending properties to categories of partial maps. Technical Report ECS-LFCS-90-107, Laboratory for Foundations of Computer Science, University of Edinburgh, February 1990. [399](#), [405](#)
- [Kea75] O. Keane. Abstract Horn theories. In F. W. Lawvere, C. Maurer, and G. C. Wraith, editors, *Model Theory and Topoi*, volume 445 of *Lecture Notes in Mathematics*, pages 15–50. Springer-Verlag, 1975. [399](#)
- [MGS89] J. Meseguer, J. A. Goguen, and G. Smolka. Order sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989. [399](#), [400](#), [403](#)
- [MT93] K. Meinke and J. V. Tucker, editors. *Many-sorted Logic and its Applications*. Wiley, 1993. [396](#)
- [Obe62] A. Oberschelp. Untersuchungen zur Mehrsortigen Quantoren Logik. *Mathematische Annalen*, 145:297–333, 1962. [397](#)
- [Poi90] A. Poigné. Parameterization for order-sorted algebraic specification. *Journal of Computer and System Sciences*, 40:229–268, 1990. [397](#), [399](#), [408](#)
- [Qia94] Z. Qian. Another look at parameterization for order-sorted algebraic specifications. *Journal of Computer and System Sciences*, 49:620–666, 1994. [397](#)
- [Rob94] E. Robinson. Variations on algebra: monadicity and generalisations of equational theories. Technical Report 6/94, School of Cognitive and Computing Sciences, University of Sussex, April 1994. [397](#)
- [RS87] D. E. Rydeheard and J. G. Stell. Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms. In D. H. Pitt et al., editors, *Category Theory and Computer Science, Edinburgh, 1987*, volume 283 of *Lecture Notes in Computer Science*, pages 114–139. Springer-Verlag, 1987. [403](#), [409](#)
- [SS89] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture notes in Artificial Intelligence*. Springer-Verlag, 1989. [397](#), [399](#)
- [Ste91] J. G. Stell. Unique-sort order-sorted theories – a description as monad morphisms. In S. Kaplan and M. Okada, editors, *Proceedings of 2nd International Workshop on Conditional and Typed Rewriting Systems, Montreal, June 1990*, volume 516 of *Lecture Notes in Computer Science*, pages 389–400. Springer-Verlag, 1991. [399](#)
- [Ste92] J. G. Stell. *Categorical Aspects of Unification and Rewriting*. PhD thesis, University of Manchester, 1992. [403](#)
- [Wal92] U. Waldmann. Semantics of order-sorted specifications. *Theoretical Computer Science*, 94:1–35, 1992. [397](#)

- [Wie91] R. J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and Object-Oriented Databases. Second International Conference, DOOD'91*, volume 566 of *Lecture Notes in Computer Science*, pages 431–452. Springer-Verlag, 1991. [396](#)

Guarded Transitions in Evolving Specifications

Dusko Pavlovic ^{*} and Douglas R. Smith ^{**}

Kestrel Institute, Palo Alto, California 94304, USA

Abstract. We represent state machines in the category of specifications, where assignment statements correspond exactly to interpretations between theories [7, 8]. However, the guards on an assignment require a special construction. In this paper we raise guards to the same level as assignments by treating each as a distinct category over a shared set of objects. A guarded assignment is represented as a pair of arrows, a guard arrow and an assignment arrow. We give a general construction for combining arrows over a factorization system, and show its specialization to the category of specifications. This construction allows us to define the fine structure of state machine morphisms with respect to guards. Guards define the flow of control in a computation, and how they may be translated under refinement is central to the formal treatment of safety, liveness, concurrency, and determinism.

1 Introduction

In previous work [8] we introduced *Evolving Specifications* (abbreviated to *especs*) as a framework for specifying, composing and refining behavior. The point of such a framework is, at the very least, to help us cross the path from ideas to running code. Programming languages are designed to support us at the final sections of that path. On one hand, *especs are evolving specifications*: diagrams of specs, displaying how the conditions, satisfied by the variables of computation, change from state to state. On the other hand, *especs are specification-carrying programs*: pieces of code, given with some global requirements and invariants, as well as annotated with some local conditions, *state descriptions*, satisfied at some states of computation and not at others. They can be construed as formalized comments, or Floyd-Hoare annotations, but made into the first-class citizens of code, i.e. available at runtime.

While such global and local specifications of the intent of computation are *hard to reconstruct* if the design records have been lost or thrown away, they are *easy to verify* if the design records are carried with the code.

^{*} Supported from the DARPA project “Specification-Carrying Software”, contract number F30602-00-C-0209, and the ONR project “Game Theoretic Framework for Reasoning about Security”, contract number N00014-01-C-0454.

^{**} Supported from the DARPA project “Specification-Carrying Software” contract number F30602-00-C-0209.

1.1 State Machines and Algebraic Specifications

Originally, state machines were introduced and studied (by Turing, Moore, Mealy, and many others) as abstract, mathematical models of computers. More recently, though, software engineering tasks reached the levels where *practical* reasoning in terms of state machines has become indispensable: designing reactive, hybrid, embedded systems seems unthinkable without the various state modeling tools and languages, like Esterel, or Statecharts. Verifying high assurance systems by model checking is based on such state machine models. Moreover, one could argue that the whole discipline of object oriented programming is essentially a method for efficient management of state in software constructs.

However, there seems to be a conceptual gap between state machines as theoretical versus practical devices. A notable effort towards bridging this gap are Gurevich's Abstract State Machines [5]: on one hand, they are a foundational paradigm of computation, explicitly compared with Turing machines; on the other hand, they have been used to present practically useful programming languages, capturing semantical features of C, Java, and others. However, the absence of powerful typing and structuring (abstraction, encapsulation, composition. . .) mechanisms makes them unsuitable for the development and management of large software systems.

We wish to investigate a representation of state machines in a framework for large-scale software specification development ("from-specs-to-code"). Previous work at Kestrel Institute has implemented the Specware/Designware framework for the development of functional programs that is based on a category of higher-order logical specifications, composition by colimit, and refinement by diagram morphisms [11, 12]. The current work builds on and extends this framework with behavioral specifications (especs), representing state machines as diagrams of specifications, and again using composition by colimit and refinement by diagram morphism. Related approaches to representing behavior in terms of a category of specifications include [2, 6].

The goal is to build a *practical* software development tool, geared towards large, complex systems, with reactive, distributed, hybrid, embedded features, and with high assurance, performance, reliability, or security requirements, all on a clean and simple semantical foundation.

1.2 Evolving Specifications

There are four key ideas underlying our representation of state machines as evolving specifications (especs). Together they reveal an intimate connection between behavior and the category of logical specifications. The first three are due to Gurevich [5].

1. *A state is a model* – A state of computation can be viewed as a snapshot of the abstract computer performing the computation. The state has a set of named stores with values that have certain properties.
2. *A state transition is a finite model change* – A transition rewrites the stored values in the state.

3. *An abstract state is a theory* – Not all properties of a state are relevant, and it is common to group states into abstract states that are models of a theory. The theory presents the structure (sorts, variables, operations), plus the axioms that describe common properties (i.e. invariants). We can treat states as static, mathematical models of a global theory thy_A , and then all transitions correspond to model morphisms. Extensions of the global theory thy_A provide local theories for more refined abstract states, introducing local variables and local properties/invariants.
4. *An abstract transition is an interpretation between theories* – Just as we abstractly describe a class of states/models as a theory, we abstractly describe a class of transitions as an interpretation between theories [7, 8]. To see this, consider the correctness of an assignment statement relative to a precondition P and a postcondition Q ; i.e. a Hoare triple $P \{x := e\} Q$. If we consider the initial and final states as characterized by theories thy_{pre} and thy_{post} with theorems P and Q respectively, then the triple is valid iff $Q[e/x]$ is a theorem in thy_{pre} . That is, the triple is valid iff the symbol map $\{x \mapsto e\}$ is an interpretation from thy_{post} to thy_{pre} . Note that interpretation goes in the *opposite* direction from the state transition.

The basic idea of specs is to use specifications (finite presentations of a theory) as state descriptions, and to use interpretations to represent transitions between state descriptions.

The idea that abstract states and abstract transitions correspond to specs and interpretations suggests that state machines are diagrams over Spec^{op} . Furthermore, state machines are composed via colimits, and state machines are refined via diagram morphisms [8].

1.3 Guards as Arrows

What's missing from the picture above is the treatment of guards on transitions. Interpretations between theories correspond exactly to (parallel) assignment statements, so something extra is needed to capture guards in this framework.

Let K and L be two states and $K \xrightarrow{g \vdash a} L$ a transition, consisting of the guard, *viz* predicate g , and the update a . Intuitively, it can be understood as the command **if** g **then** a , executed at the state K , and leading into L by a — whenever the guard condition g is satisfied. More precisely, it is executed in two steps:

- at the state K , the condition g is evaluated;
- if it is satisfied, the update a is performed.

Every guarded update $K \xrightarrow{g \vdash a} L$ thus factors in the form

$$(g \vdash a) = (g \vdash \text{id}) \cdot (\top \vdash a)$$

where $g \vdash \text{id}$ is a guard with a trivial update (with the identity id mapping all symbols of K to themselves), whereas $\top \vdash a$ is the unguarded update (with the condition \top always satisfied).

Going a step further, to compose two guarded commands

$$(g_1 \vdash a_1) \cdot (g_2 \vdash a_2) = (g_1 \vdash \text{id}) \cdot (\top \vdash a_1) \cdot (g_2 \vdash \text{id}) \cdot (\top \vdash a_2)$$

suggests the need for a switching operator to interchange the positions of a_1 and g_2 to obtain

$$(g_1 \vdash \text{id}) \cdot (g'_2 \vdash \text{id}) \cdot (\top \vdash a'_1) \cdot (\top \vdash a_2) = g_1 \cdot g'_2 \vdash a'_1 \cdot a_2$$

This gives rise to the following

Task: Given two classes of morphisms \mathbb{G} and \mathbb{A} over the same class of objects \mathcal{S} , construct the category $\mathcal{S}_{\mathbb{G} \vdash \mathbb{A}}$, where the morphisms will be the composites $g \vdash a$ of the elements of the two classes, which will be recovered as components of a factorization system.

The remainder of the paper introduces a general mathematical construction for $\mathcal{S}_{\mathbb{G} \vdash \mathbb{A}}$, and shows how to treat guarded transitions as a special case.

2 Construction

2.1 Simple Form

Let \mathbb{G} and \mathbb{A} be two categories over the same class of objects \mathcal{S} . We want to form the category $\mathcal{S}_{\vdash} = \mathcal{S}_{\mathbb{G} \vdash \mathbb{A}}$ with a factorization system where the \mathbb{G} -arrows will be the abstract epis and the \mathbb{A} -arrows the abstract monics. This means that \mathbb{G} and \mathbb{A} will appear as full subcategories of \mathcal{S}_{\vdash} , and every \mathcal{S}_{\vdash} -morphism will factorize as a composite of an \mathbb{G} -morphism followed by an \mathbb{A} -morphism, orthogonal to each other in the usual sense [1, 3].

The requirements induce the definition

$$\begin{aligned} |\mathcal{S}_{\mathbb{G} \vdash \mathbb{A}}| &= \mathcal{S} \\ \mathcal{S}_{\mathbb{G} \vdash \mathbb{A}}(K, L) &= \sum_{X \in \mathcal{S}} \mathbb{G}(K, X) \times \mathbb{A}(X, L) \end{aligned}$$

Notation. An arrow \mathcal{S}_{\vdash} , which is a triple $\langle X, g, a \rangle$, will usually be written in the form $(g \vdash a)$. The components $g \in \mathbb{G}$ and $a \in \mathbb{A}$ will sometimes be called guard and action, respectively.

Conversely, given an arrow $f \in \mathcal{S}_{\vdash}$, we'll denote by f^\bullet a representative of the \mathbb{G} -component, and by $\bullet f$ a representative of the \mathbb{A} -component. In summary,

$$\begin{aligned} f &= (g \vdash a) \text{ means that} \\ f^\bullet &= g \text{ and} \\ \bullet f &= a \end{aligned}$$

Towards the definition of the composition

$$\cdot : \mathcal{S}_-(A, B) \times \mathcal{S}_-(B, C) \longrightarrow \mathcal{S}_-(A, C)$$

note that its naturality with respect to the pre-composition in \mathbb{G} and the post-composition in \mathbb{A} means that it extends the composition in these two categories, i.e.

$$\begin{aligned} g' \cdot (g \vdash a) &= (g' \cdot g) \vdash a \\ (g \vdash a) \cdot a' &= g \vdash (a \cdot a') \end{aligned}$$

For the moment, these equations can be viewed as notational conventions. Later, when the composition is defined, they will be derivable as properties. Similarly, the left-hand side of

$$g' \cdot f \cdot a' = (g' \cdot f \bullet) \vdash (\bullet f \cdot a') \quad (1)$$

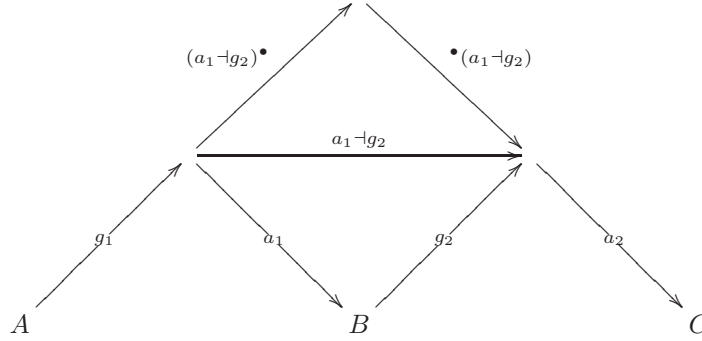
is just a convenient abbreviation of the right-hand side.

The composition can now be defined in the form

$$(g_1 \vdash a_1) \cdot (g_2 \vdash a_2) = g_1 \cdot (a_1 \dashv g_2) \cdot a_2$$

which is the abbreviated form of

$$(g_1 \vdash a_1) \cdot (g_2 \vdash a_2) = g_1 \cdot (a_1 \dashv g_2) \bullet \vdash \bullet (a_1 \dashv g_2) \cdot a_2 \quad (2)$$

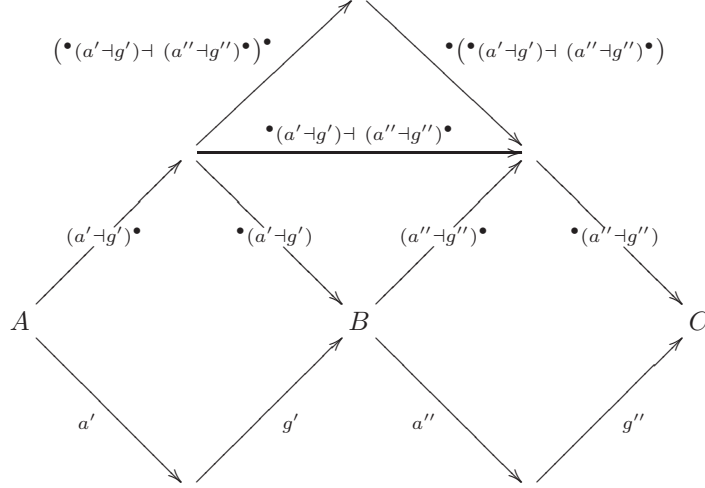


where the middle component comes from the family of *switching* functors

$$\dashv^{AB} : \sum_{Z \in \mathcal{S}} \mathbb{A}(A, Z) \times \mathbb{G}(Z, B) \longrightarrow \mathcal{S}_-(A, B)$$

natural for the pre-composition with the \mathbb{A} -morphisms to A and for the post-composition with the \mathbb{G} -morphisms out of B . Moreover, the switching is required to satisfy the following equations (simplified by (1))

$$\begin{aligned} a \dashv \text{id} &= \text{id} \vdash a \\ \text{id} \dashv g &= g \vdash \text{id} \\ a' \dashv g' \cdot (a'' \dashv g'') \bullet &= (a' \dashv g') \bullet \cdot (\bullet (a' \dashv g') \dashv (a'' \dashv g'')) \bullet \\ \bullet (a' \dashv g') \cdot a'' \dashv g'' &= (\bullet (a' \dashv g') \dashv (a'' \dashv g'')) \bullet \cdot \bullet (a'' \dashv g'') \end{aligned}$$



The first two of these equations ensure that $(\text{id} \vdash \text{id})$ play the role of the identities in \mathcal{S}_\vdash . The last two allow us to prove that the composition is associative.¹

$$\begin{aligned}
 ((g_1 \vdash a_1) \cdot (g_2 \vdash a_2)) \cdot (g_3 \vdash a_3) &\stackrel{(2)}{=} (g_1 \cdot (a_1 \vdash g_2)^\bullet \vdash \bullet(a_1 \vdash g_2) \cdot a_2) \cdot (g_3 \vdash a_3) \\
 &\stackrel{(2)}{=} \left\{ \begin{array}{l} g_1 \cdot (a_1 \vdash g_2)^\bullet \cdot (\bullet(a_1 \vdash g_2) \cdot a_2 \vdash g_3)^\bullet \\ \vdash \bullet(\bullet(a_1 \vdash g_2) \cdot a_2 \vdash g_3) \cdot a_3 \end{array} \right. \\
 &= \left\{ \begin{array}{l} g_1 \cdot (a_1 \vdash g_2)^\bullet \cdot (\bullet(a_1 \vdash g_2) \vdash (a_2 \vdash g_3)^\bullet)^\bullet \\ \vdash \bullet(\bullet(a_1 \vdash g_2) \vdash (a_2 \vdash g_3)^\bullet) \cdot (a_2 \vdash g_3)^\bullet \cdot a_3 \end{array} \right. \\
 &= \left\{ \begin{array}{l} g_1 \cdot (a_1 \vdash g_2 \cdot (a_2 \vdash g_3)^\bullet)^\bullet \\ \vdash \bullet(a_1 \vdash g_2 \cdot (a_2 \vdash g_3)^\bullet) \cdot (a_2 \vdash g_3)^\bullet \cdot a_3 \end{array} \right. \\
 &= (g_1 \vdash a_1) \cdot (g_2 \cdot (a_2 \vdash g_3)^\bullet \vdash \bullet(a_2 \vdash g_3) \cdot a_3) \\
 &= (g_1 \vdash a_1) \cdot ((g_2 \vdash a_2) \cdot (g_3 \vdash a_3))
 \end{aligned}$$

Proposition 1. *Given the categories \mathbb{G} and \mathbb{A} over the object class \mathcal{S} , the category \mathcal{S}_\vdash is universal for*

- categories \mathcal{K} , given with
- a factorization² (\mathbb{E}, \mathbb{M}) , and
- the functors

$$G : \mathbb{G} \rightarrow \mathbb{E} \hookrightarrow \mathcal{K} \text{ and}$$

$$A : \mathbb{A} \rightarrow \mathbb{M} \hookrightarrow \mathcal{K}$$

¹ The abbreviated form of (2) does not seem particularly useful here.

² For simplicity, we are taking an inessentially weaker notion of factorization than e.g. [3], omitting the requirement that both families contain all isomorphisms. Respecting this requirement would amount to an additional step of saturating the families.

that coincide on the objects
 – and such that for all composable actions a and guards g

$$A(a) \cdot G(g) = G(a \dashv g) \bullet \cdot A \bullet (a \dashv g)$$

Proof. The category \mathcal{S}_\vdash satisfies the above conditions. Indeed, the classes of arrows

$$\mathbb{E} = \{g \vdash \text{id} \mid g \in \mathbb{G}\}$$

$$\mathbb{M} = \{\text{id} \vdash a \mid a \in \mathbb{A}\}$$

form a factorization. This means that every \mathcal{S}_\vdash -arrow decomposes

$$(g \vdash a) = (g \vdash \text{id}) \cdot (\text{id} \vdash a)$$

The two families are orthogonal because every commutative square

$$\begin{array}{ccc} A_1 & \xrightarrow{g_1 \vdash a_1} & B_1 \\ g \vdash \text{id} \downarrow & \nearrow \tilde{g} \vdash \tilde{a} & \downarrow \text{id} \vdash a \\ A_2 & \xrightarrow{g_2 \vdash a_2} & B_2 \end{array}$$

can be filled with a unique diagonal, making both triangles commute. Indeed, the commutativity of the square means that

$$g_1 = g \cdot g_2 \text{ and}$$

$$a_1 \cdot a = a_2$$

so that we can simply take

$$\tilde{g} = g_2 \text{ and}$$

$$\tilde{a} = a_1$$

The switch functors are

$$(a \dashv g) = (g \vdash a)$$

so that the required condition holds

$$\begin{aligned} A(a) \cdot G(g) &= (\text{id} \vdash a) \cdot (g \vdash \text{id}) \\ &= (g \vdash a) \\ &= (a \dashv g) \\ &= G(a \dashv g) \bullet \cdot A \bullet (a \dashv g) \end{aligned}$$

The universality of \mathcal{S}_- now boils down to the fact that the functors $G : \mathbb{G} \rightarrow \mathbb{E} \hookrightarrow \mathcal{K}$ and $A : \mathbb{A} \rightarrow \mathbb{M} \hookrightarrow \mathcal{K}$ extend to a unique functor $H : \mathcal{S}_- \rightarrow \mathcal{K}$, preserving the factorizations. The object part of this functor is already completely determined by the object parts of G and A , which coincide. Since the factorization is required to be preserved, the arrow part will be

$$H(g \vdash a) = G(g) \cdot A(a)$$

The functoriality follows from the assumptions:

$$\begin{aligned} H(g_1 \vdash a_1) \cdot H(g_2 \vdash a_2) &= G(g_1) \cdot A(a_1) \cdot G(g_2) \cdot A(a_2) \\ &= G(g_1) \cdot G(a_1 \dashv g_2)^\bullet \cdot A^\bullet(a_1 \dashv g_2) \cdot A(a_2) \\ &= G(g_1 \cdot (a_1 \dashv g_2)^\bullet) \cdot A^\bullet(a_1 \dashv g_2) \cdot a_2 \\ &= H((g_1 \vdash a_1) \cdot (g_2 \vdash a_2)) \end{aligned}$$

2.2 Examples

Decompositions. Trivially, the construction can be applied to $\mathbb{A} = \mathbb{G} = \mathcal{C}$, for any category \mathcal{C} . The switch functors can also be trivial, and map $\langle Z, f, g \rangle \in \sum_{X \in \mathcal{C}} \mathbb{A}(K, X) \times \mathbb{G}(X, L)$ to the same triple $\langle Z, f, g \rangle$ but this time as element of $\sum_{X \in \mathcal{C}} \mathbb{G}(K, X) \times \mathbb{A}(X, L)$.

The morphisms $(f \vdash g) : K \rightarrow L$ in resulting category \mathcal{C}_- are simply the composable pairs of morphisms, leading from K to L in \mathcal{C} .

Adjoining a monoid of guards. Let $\langle \Gamma, \otimes, \top \rangle$ be a monoid, \mathcal{C} an arbitrary category, and let

$$\mathcal{C} \times \Gamma \xrightarrow{\otimes} \mathcal{C}$$

be a monoid action, also denoted \otimes , by abuse of notation. For every $g \in \Gamma$ there is thus a functor $(-) \otimes g : \mathcal{C} \rightarrow \mathcal{C}$, satisfying

$$\begin{aligned} A \otimes \top &= A \\ (A \otimes g_1) \otimes g_2 &= A \otimes (g_1 \otimes g_2) \end{aligned}$$

We want to adjoin the elements of Γ to \mathcal{C} as abstract epis, while keeping the original \mathcal{C} -arrows as the abstract monics. So take $\mathbb{A} = \mathcal{C}$, and define the hom-sets of \mathbb{G} by

$$\mathbb{G}(K, L) = \{g \in \Gamma \mid K \otimes g = L\}$$

The composition is clearly induced from Γ : if $K \otimes g_1 = L$ and $L \otimes g_2 = M$, then $K \otimes (g_1 \otimes g_2) = M$ makes $g_1 \otimes g_2$ into an arrow from K to M .

The \mathcal{C}_- construction now becomes

$$\mathcal{C}_-(K, L) = \sum_{g \in \Gamma} \mathcal{C}(K \otimes g, L)$$

The switch functors are

$$(a \dashv g) = (g \vdash a \otimes g)$$

$$\begin{array}{ccc}
 & K \otimes g & \\
 g \nearrow & & \searrow a \otimes g \\
 K & & L \otimes g \\
 a \searrow & & \nearrow g \\
 & L &
 \end{array}$$

so that the composition is

$$(g_1 \vdash a_1) \cdot (g_2 \vdash a_2) = g_1 \otimes g_2 \vdash (a_1 \otimes g_2) \cdot a_2$$

Objects as guards. An interesting special case of the above arises when \mathcal{C} is a strict monoidal category³. Its object class $|\mathcal{C}|$ is then a monoid, which comes with the action

$$\mathcal{C} \times |\mathcal{C}| \xrightarrow{\otimes} \mathcal{C}$$

The category \mathcal{C}_\vdash is formed as above, with the objects of \mathcal{C} as the guards.

If \mathcal{C} is the category \mathcal{N} of von Neumann natural numbers, i.e. $n = \{0, 1, 2, \dots, n-1\}$ with all functions as morphisms. This is, of course, equivalent to the category of finite sets, but the monoidal structures induced by the products, or the coproducts, can be made strict, using the arithmetic operations. The finiteness does not matter either, so the category \mathcal{O} of all ordinals would do just as well, but looks bigger.

With the cartesian products as the monoidal structure, i.e. $\otimes = \times$ and $\top = 1$, a guarded morphism $(g \vdash a) : k \longrightarrow \ell$ is a function $a : k \times g \longrightarrow \ell$. The numbers g, k, ℓ can be thought of as arities; besides the inputs of arity k , the function a thus also requires the inputs from the guard g . Its composite with $(h \vdash b) : \ell \longrightarrow m$ just accumulates the arguments $x : g$ and $y : h$

$$(g \vdash a) \cdot (h \vdash b)(u, x, y) = a(b(u, x), y)$$

If the monoidal structure is the coproducts, i.e. $\otimes = +$ and $\top = 0$, then $(g \vdash a) : k \longrightarrow \ell$ is a pair of functions $[a_0, a_1] : k + g \longrightarrow \ell$. The composite with $(h \vdash b) : \ell \longrightarrow m$, i.e. $[b_0, b_1] : \ell + h \longrightarrow m$ is the triple

$$(g \vdash a) \cdot (h \vdash b) = [a_0 \cdot b_0, a_1 \cdot b_0, b_1]$$

On the other hand, if we use the monoid $\langle \mathcal{N}, +, 0 \rangle$, a guarded morphism $(g \vdash a) : k \longrightarrow \ell$ will be a function $a : k \longrightarrow \ell + g$, which besides the outputs of arity ℓ may yield some outputs of arity g .

³ In fact, premonoidal [9] is enough.

2.3 Full Construction

Some natural examples require a more general construction. For instance, starting from a category \mathcal{S} with a factorization system (\mathbb{E}, \mathbb{M}) and taking $\mathbb{G} = \mathbb{E}$ and $\mathbb{A} = \mathbb{M}$, one would expect to get $\mathcal{S} \simeq \mathcal{S}_-$. However, one gets a category where the morphisms are the *chosen* factorizations, which is essentially larger than the original one (it may not be locally small).

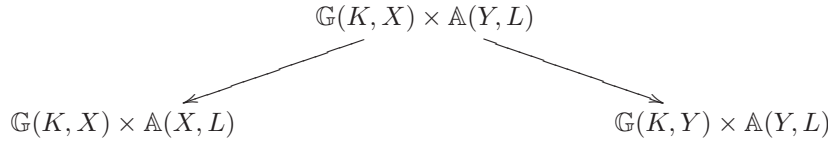
Moreover, looking back at the motivating example $\mathcal{S} = \text{Spec}$, one might like to develop the guard category using the *specification morphisms*, rather than interpretations as the class of updates. This suggests the task of generating from \mathcal{S} , \mathbb{A} and \mathbb{G} a category where the elements of a chosen class of morphisms \mathbb{D} will boil down to isomorphisms. In the case when \mathcal{S} is Spec , the class \mathbb{D} are the definitional extension.

So suppose now that we are given a class \mathcal{S} and three categories over it, \mathbb{A} , \mathbb{D} and \mathbb{G} , such that \mathbb{D} is a subcategory of both \mathbb{A} and \mathbb{G} . So we have the faithful functors $\mathbb{D} \hookrightarrow \mathbb{A}$ and $\mathbb{D} \hookrightarrow \mathbb{G}$, both bijective on objects.

For every pair of objects $K, L \in \mathcal{S}$ there is a diagram

$$\begin{aligned} \mathbb{D} \times \mathbb{D}^{op} &\longrightarrow \text{Set} \\ \langle X, Y \rangle &\longmapsto \mathbb{G}(K, X) \times \mathbb{A}(Y, L) \end{aligned}$$

The coend $\int_{X \in \mathbb{D}} \mathbb{G}(K, X) \times \mathbb{A}(X, L)$ of this diagram [10, IX.6] identifies elements of $\mathbb{G}(K, X) \times \mathbb{A}(X, L)$ and $\mathbb{G}(K, Y) \times \mathbb{A}(Y, L)$ along the spans



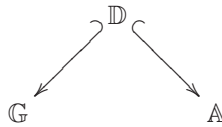
induced by post-composing with $d \in \mathbb{D}(X, Y)$ in \mathbb{G} and by pre-composing in \mathbb{A} .

The category $\mathcal{S}_- = \mathcal{S}_{\mathbb{D}}^{\mathbb{G} \dashv \mathbb{A}}$ is now defined

$$\begin{aligned} |\mathcal{S}_-| &= \mathcal{S} \\ \mathcal{S}_-(K, L) &= \int_{X \in \mathbb{D}} \mathbb{G}(K, X) \times \mathbb{A}(X, L) \end{aligned}$$

If the \mathbb{D} -arrows support the right calculus of fractions in \mathbb{G} and the left calculus of fractions in \mathbb{A} , the coends can be simplified by the usual equivalence relations [4, 1]. In any case, the composition follows from the universal property of the coends, and this general construction yields the following universal property of \mathcal{S}_- .

Proposition 2. *Given the categories*



over the same object class \mathcal{S} , the category \mathcal{S}_- is universal for

- categories \mathcal{K} , given with
- a factorization (\mathbb{E}, \mathbb{M}) , and
- the functors

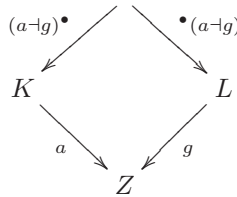
$$\begin{aligned} G : \mathbb{G} &\rightarrow \mathbb{E} \hookrightarrow \mathcal{K} \text{ and} \\ A : \mathbb{A} &\rightarrow \mathbb{M} \hookrightarrow \mathcal{K} \\ D : \mathbb{D} &\rightarrow \mathbb{I} \hookrightarrow \mathcal{K} \end{aligned}$$

- where \mathbb{I} is groupoid of isomorphisms of \mathcal{K} . These three functors coincide on the objects, and for all composable actions a and guards g holds
- satisfy

$$A(a) \cdot G(g) = G(a \dashv g) \bullet A \bullet (a \dashv g)$$

Examples

Spans. If \mathcal{C} is a category with pullbacks, we can take $\mathbb{G} = \mathcal{C}^{op}$, $\mathbb{A} = \mathcal{C}$ and $\mathbb{D} = \mathbb{I}$ the groupoid of all isomorphisms in \mathcal{C} . An abstract epi from K to L is now an ordinary \mathcal{C} -map from L to K . The switch functors can now be defined using pullbacks:



The above construction yields the category of matrices in \mathcal{C} , in contrast with the bicategory of spans, obtained by the earlier simple construction. E.g., for $\mathcal{C} = \text{Set}$, the morphisms of this category are ordinal matrices, with the matrix multiplication as the composition.

Opspans. Dually, when \mathcal{C} is a category with pushouts, we can take $\mathbb{G} = \mathcal{C}$ and $\mathbb{A} = \mathcal{C}^{op}$, and get a “strictified” version of the bicategory of opspans as \mathcal{C}_+ . In contrast with the span composition, which boils down to the matrix composition, the opspan composition corresponds to the equivalence relation unions.

Factorization systems Given a category \mathcal{C} with a factorization system (\mathbb{E}, \mathbb{M}) , we can now recover $\mathcal{C} = \mathcal{C}_+$ by taking \mathbb{D} to be the isomorphism groupoid \mathbb{I} of \mathcal{C} . This gives an equivalence of the 2-category of factorization systems, with the 2-category of the triples $\langle \mathbb{E}, \mathbb{M}, \mathbb{I} \rangle$.

3 Adjoining Guards to Spec

The general construction can now be specialized to obtain a suitable category of guarded transitions in state machines. We start from the class $\mathcal{S} = |\mathbf{Spec}|$, and want to build the category of guarded transitions from the categories \mathbb{A} of unguarded transitions, and \mathbb{G} of guards. The unguarded transitions will be the interpretations backwards, i.e.

$$\mathbb{A} = \mathbf{Spec}^{op}$$

But what to take as the guards? Probably the simplest idea is to take *all interpretations*:

$$\mathbb{G} = \mathbf{Spec}$$

With the switch functors induced by the pushouts, the resulting category of guarded interpretations is just the category of opspans in \mathbf{Spec} . A guarded transition $K \xrightarrow{g \dashv f} L$ is thus just an opspan $K \xrightarrow{g} M \xleftarrow{f} L$ in \mathbf{Spec} . The guard M is a separate spec, given with the interpretations g and f of K and L respectively, *and* possibly including some private variables, sorts and operations, equally invisible to K and to L .

As semantics of transitions, the opspans have the obvious shortcoming that they are completely symmetric: given an $K \xrightarrow{g} M \xleftarrow{f} L$, we have no way to tell whether it is a transition from K to L , or from L to K , since the guard M is symmetrically related, and can be equally unrelated to both. The opspan semantics is thus *not full*: not all opspans can be reasonably interpreted as transitions.

Intuitively, one might want to think of the guarded transition $K \xrightarrow{\Phi \dashv f} L$ as the command `if Φ then f` , executed at the state K , and leading into L by f — whenever the guard condition Φ is satisfied. The above symmetry is broken by the assumption that the guard Φ is evaluated at state K , and not L , or some third state. This is why, in general, such a transition is *irreversible*: once the state K is overwritten, it cannot in general be recovered, nor can Φ be re-evaluated.

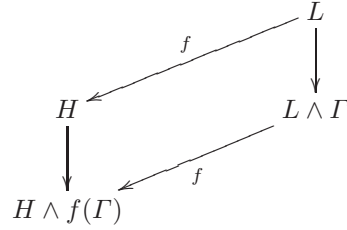
Following this intuition, that the guard Φ is evaluated at K , we restrict the opspan semantics by the requirement that Φ is expressible in the language \mathcal{L}_K of K :

$$\mathbb{G}(K, M) = \{\Phi \in \mathcal{L}_K \mid K \wedge \Phi = M\}$$

where $K \wedge \Phi$ denotes the spec that extends K with the axiom Φ . Indeed, it seems reasonable to require that all variables of Φ must be known and determined at the state K , where this condition is evaluated, causing the transition to fire, or not.

The switching functors are given by

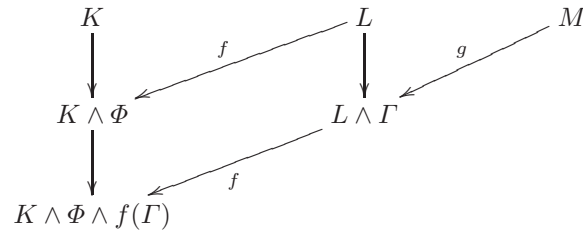
$$f \dashv \Gamma = f(\Gamma) \vdash f$$



so that arrow composition is given by the rule

$$\frac{K \xrightarrow{\Phi \vdash f} L \quad L \xrightarrow{\Gamma \vdash g} M}{K \xrightarrow{\Phi \wedge f(\Gamma) \vdash f \cdot g} M}$$

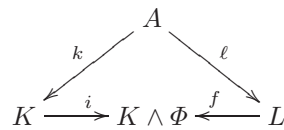
or diagrammatically



In summary, the category Spec_\vdash of specifications and guarded transitions will have specifications as its objects, while the hom-sets will be

$$\text{Spec}_\vdash(K, L) = \{(\Phi \vdash f) \mid \Phi \in \mathcal{L}_K \text{ and } f \in \text{Spec}(L, K \wedge \Phi)\}$$

In general, for a fixed spec A , we shall define A/Spec_\vdash to be the category with the specs inheriting A (*viz* the extensions $A \xrightarrow{k} K$) as objects⁴, arrows $\Phi \vdash f$ have the form



⁴ The state descriptions of a machine are not unrelated specs, but they share/inherit the common global spec, capturing whatever may be known about the global invariants and the intent of the program. The abstract states K and L thus come with the interpretations $k : A \rightarrow K$ and $\ell : A \rightarrow L$ of the globally visible signature and the invariants specified in A . The universe from which the states are drawn is thus the category A/Spec , of all specs inheriting A , rather than just Spec . While Spec is fibered over Lang by the functor mapping each spec K to its language \mathcal{L}_K , the category A/Spec is fibered over $\mathcal{L}_A/\text{Lang}$, mapping each interpretation $A \rightarrow K$ to the underlying language translation.

and the hom-sets will be

$$A/\text{Spec}_+(k, \ell) = \{(\Phi \vdash f) \mid \Phi \in \mathcal{L}_K \text{ and } f \in \text{Spec}(L, K \wedge \Phi) \text{ and } k \cdot i = \ell \cdot f\}$$

for $\mathbb{A} = A/\text{Spec}$ and $\mathbb{G}(k, \ell) = \{\Phi \in \mathcal{L}_K \mid K \wedge \Phi = L\}$.

Remark. Note that the construction of Spec_+ is just a fibered form of adjoining objects as guards: for each language Σ the semilattice $\langle |\text{Spec}_\Sigma|, \wedge, \top \rangle$ of all theories over Σ is adjoined to the fiber Spec_Σ as the monoid of guards.

The construction A/Spec_+ is, on the other hand, a fibered form of a slightly more general construction. The category A/Spec is fibered over the category $\mathcal{L}_A/\text{Lang}$ of languages interpreting \mathcal{L}_A . But this time, each fiber $(A/\text{Spec})_\sigma$ over $\sigma : \mathcal{L}_A \rightarrow \Sigma$ is assigned not its own monoid of objects, but again just the semilattice of theories over Σ .

4 Concluding Remarks

One remarkable fact arising from the above presentation is that an elementary question arising from a basic computational model — the question of guards in evolving specifications — already leads to an apparently novel categorical construction, which may be of independent mathematical interest. The general construction, outlined in Section 2.3, corresponds to a *double* calculus of fractions, combining left and right fractions in a common framework, with new universal properties. The details and the instances of this new construction, which appears to lead to some interesting groups and algebras, remain to be investigated in a separate paper.

On the *spec* side, from which it originates, the construction now guides the development of a suitable category for guarded actions, allowing us to treat guarded transition systems (*especs*) as diagrams over Spec^{op} . In accord with the ideas and structures presented in [8], we can then compose *especs* using colimits, and we can refine systems of *especs* using diagram morphisms, providing a solid mathematical background for interleaving and combining bottom-up and top-down software development in a unified framework.

The conditions under which guards are mapped under refinement directly affect key behavioral properties such as nondeterminism, safety, and liveness. These will be explored in a subsequent paper. Other future directions for this work include adding timing constraints and exploring resource-bounded computation, and modeling hybrid embedded systems with *especs*.

References

- [1] BORCEUX, F. *Handbook of Categorical Algebra 1: Basic Category Theory*, vol. 50 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, Cambridge, 1994. 414, 420
- [2] ERRINGTON, L. Notes on diagrams and state. Tech. rep., Kestrel Institute, 2000. 412

- [3] FREYD, P., AND KELLY, G. M. Categories of continuous functors I. *Journal of Pure and Applied Algebra* 2, 3 (1972), 169–191. 414, 416
- [4] GABRIEL, P., AND ZISMAN, M. *Calculus of Fractions and Homotopy Theory*, vol. 36 of *Ergebnisse der Mathematik und ihrer Grenzgebiete. New Series*. Springer-Verlag, New York, 1967. 420
- [5] GUREVICH, Y. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Boerger, Ed. Oxford University Press, 1995, pp. 9–36. 412
- [6] J. L.FIADEIRO, AND T.MAIBAUM. Interconnecting formalisms: supporting modularity, reuse and incrementality. In *Proc. 3rd Symposium on the Foundations of Software Engineering* (1995), G. Kaiser, Ed., ACM Press, pp. 72–80. 412
- [7] KUTTER, P. W. State transitions modeled as refinements. Tech. Rep. KES. U.96.6, Kestrel Institute, August 1996. 411, 413
- [8] PAVLOVIC, D., AND SMITH, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Automated Software Engineering Conference* (2001), IEEE Computer Society Press, pp. 157–165. 411, 413, 424
- [9] POWER, J., AND ROBINSON, E. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science* 7, 5 (1997), 453–468. 419
- [10] MACLANE, S. *Categories for the Working Mathematician*, vol. 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1971. 420
- [11] SMITH, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292. 412
- [12] SRINIVAS, Y. V., AND JÜLLIG, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422. 412

Revisiting the Categorical Approach to Systems*

Ant nia Lopes¹ and Jos Luiz Fiadeiro^{1,2}

¹ Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt

²ATX Software SA
Alameda Ant nio S rgio 7, 1A, 2795-023 Linda-a-Velha, Portugal
jose@fiadeiro.org

Abstract. Although the contribution of Category Theory as a mathematical platform for supporting software development, in the wake of Goguen's approach to General Systems, is now reasonably recognised, accepted and even used, the emergence of new modelling techniques and paradigms associated with the New-Economy suggests that the whole approach needs to be revisited. In this paper, we propose some revisions capitalising on the lessons that we have learned in using categorical techniques for formalising recent developments on Software Architectures, Coordination Technologies, and Service-Oriented Software Development in general.

1 Introduction

In the early 70s, J.Goguen proposed the use of categorical techniques in General Systems Theory for unifying a variety of notions of system behaviour and their composition techniques [11,12]. The main principles of this approach are the following:

- components are modelled as objects;
- system configurations are modelled as diagrams that depict the way in which the components of the system are interconnected;
- the behaviour of a complex system is given through the colimit of its configuration diagram.

These categorical principles have been used to formalise different mathematical models of system behaviour [13,24,25,14], their logical specifications [2,7] and their realisations as parallel programs [5].

This categorical approach has also been used as a platform for comparing different modelling approaches and for giving semantics to the gross modularisation of complex systems (e.g., [26,4]). Grounded on the principle that a design formalism can be

* This research was partially supported by Funda o para a Ci ncia e Tecnologia through project POSI/32717/00 (FAST).

expressed as a category, we have established an algebraic characterisation of the notions of compositionality [8] and coordination [9], and formalised architectural principles in software design [10].

More recently, in the scope of a joint research effort with ATX Software – a Portuguese IT company with a very strong R&D profile – we have been engaged in the development of coordination technologies for supporting the levels of agility that are required on systems by the new ways in which companies are doing business, which includes coping with the volatility of requirements and the new paradigms associated with the Internet (B2C, B2B, P2P, etc). Our research has focused on a set of primitives centred around the notion of coordination contract – with which OO languages such as the UML can be extended to incorporate modelling techniques that have been made available in the areas of Software Architectures, Parallel Program Design and Reconfigurable Distributed Systems.

The coordination technologies that we have been developing are largely language independent – in the sense that they can be used together with a variety of modelling and design languages. Our experience in formalising this independence through the use of categorical techniques made us realise that the general approach as outlined above is far too simplistic and restrictive. On the one hand, it does not allow one to capture the typical restrictions that apply to the way components can be interconnected (e.g. [19]). These restrictions are of crucial importance because properties such as compositionality usually depend on the class of diagrams that represent correct configurations. On the other hand, as soon as more sophisticated models of behaviour are used, e.g. those in which non-determinism needs to be explicitly modelled, different mechanisms become necessary to support vertical and horizontal structuring. Hence, we cannot expect that the same notion of morphism can be used to support "component-of" relationships as required for horizontal structuring, and "refined-by" relationships as required by vertical structuring.

In this paper, we review the categorical approach in order to support the two new aspects motivated above, including a new characterisation of the notion of compositionality. We use a simple program design formalism –CommUnity, to illustrate the applicability of our proposal. Although we could have used instead a sophisticated and powerful formalism, the simplicity of CommUnity is ideal to present what we have found to be important features of the new breed of design formalisms that need to be accommodated in the categorical approach.

2 Capturing Constraints on Configurations

The basic motto of the categorical approach to systems design is that morphisms can be used to express interaction between components. More concretely, when a design formalism is expressed as a category *DESC*, a configuration of interconnected components can be expressed as a diagram in *DESC*. The semantics of the

configuration, i.e. the description of the system that results from the interconnections, can be obtained from the diagram by taking its colimit.

In the languages and formalisms that are typically used for the configuration of software systems, the interconnection of components is usually governed by specific rules, as happens with physical components, say in hardware or mechanical systems. A simple example can be given in terms of languages with I/O communication. Typically in these languages output channels cannot be connected to each other (see for instance [21,1]). Hence, it may happen that not every diagram represents a correct configuration. For instance, if *DESC* is not finitely cocomplete, then not every configuration of components is "viable", in the sense that it gives rise to a system (has a semantics). Clearly, in this situation not every configuration of components is correct.

However, the correctness of a configuration diagram cannot always be reduced to the existence of a colimit. There are situations in which the colimit exists (the category may even be cocomplete) but, still, the diagram may be deemed to be incorrect according to the rules set up by the configuration language. In situations like this, the rules that establish the correctness of configurations are not internalised in the structure of the category and have to be expressed as restrictions on the diagrams that are considered to be admissible as configurations. In the situations we have to deal with in practice, such correctness criteria are reflected on properties of the category that are related to properties of the development approach, such as the separation between computation and coordination, and compositionality. A specific example will be presented in section 3.

Definition 1 (Configuration criterion).

A configuration criterion *Conf* over *DESC* is a set of finite diagrams in *DESC* s.t. if $dia \in Conf$ then *dia* has a colimit.

In order to illustrate the notion of configuration criterion, we use a very simple interface definition language. Further on, we will use a program design language that is built over this interface language.

We consider that the interaction between a component and its environment is modelled by the simultaneous execution of actions and by exchanging data through shared channels. We will distinguish between input and output channels. We shall also assume a fixed set *S* of sort symbols for typing the data that can be exchanged through these channels.

*An interface description θ is a triple $\langle O, I, \Gamma \rangle$ where *O* and *I* are disjoint *S*-indexed families of sets and Γ is a finite set. The union of *O* and *I* is denoted by *V*.*

The families *O* and *I* model, respectively, the output and the input channels, and the set Γ models the actions that a given system makes available through its public interface. For each sort *s*, O_s (resp. I_s) is the set of output (resp. input) channels that take data in *s*.

An interface morphism $\sigma: \theta_1 \rightarrow \theta_2$ is a pair $\langle \sigma_{ch}, \sigma_{ac} \rangle$ where $\sigma_{ch}: V_1 \rightarrow V_2$ is a total function s.t. (1) $\sigma_{ch}(V_{1s}) \subseteq V_{2s}$, for every $s \in S$ and (2) $\sigma_{ch}(O_1) \subseteq O_2$ and $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ is a partial mapping.

An interface morphism σ from θ_1 to θ_2 supports the identification of a way in which a system with interface θ_1 is a component of another system, with interface θ_2 . The function σ_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The typing of the channels has to be preserved and the output channels of the component have to be mapped to output channels of the system. Notice, however, that input channels of the component may be identified with output channels of the system. This is because the result of interconnecting an input of θ_1 with an output of another component of θ_2 results in an output for the whole system. This is different from most approaches in that communications are normally internalised and not made available in the interface of the resulting system. We feel that hiding communication is a decision that should be made explicitly and not as a default, certainly not as a part of a minimal operation of interconnection which is what we want to capture (see [25] for a categorical characterisation of hiding).

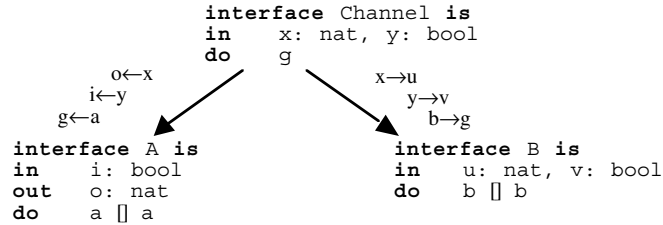
*Interface descriptions and interface morphisms constitute a category which we will denote by **INTF**.*

Interface morphisms can be used to establish synchronisation between actions of different components as well as the interconnection of input channels of one component with output channels of other components. However, they can also be used to establish the interconnection of an output channel of one component with an output channel of another component, a situation that in our formalism is ruled out.

In order to express that, in the interface description formalism we are presenting, output channels of an interface cannot be connected with output channels of the same or other interfaces we define which diagrams represent correct configurations.

A diagram $\mathbf{dia}: \mathbf{D} \rightarrow \mathbf{INTF}$ is a configuration diagram iff if $(\mu_n: \mathbf{dia}(n) \rightarrow \langle I, O, \Gamma \rangle)_{n \in |\mathbf{D}|}$ is a colimit of \mathbf{dia} , then for every $v \in O$, there exists exactly one n in $|\mathbf{D}|$ s.t. $\mu_n^{-1}(v) \cap O_{\mathbf{dia}(n)} \neq \emptyset$ and, for such n , $\mu_n^{-1}(v) \cap O_{\mathbf{dia}(n)}$ is a singleton.

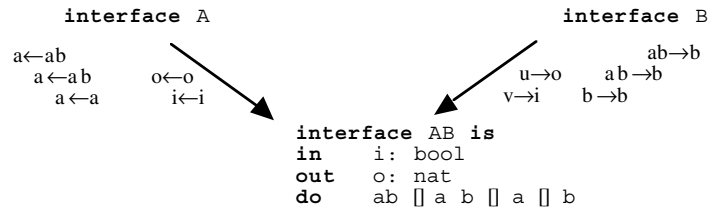
That is to say, in a configuration diagram each output channel of the system can be traced back to one, and only one, output channel of one, and only one, component.



An example of a configuration diagram is given above. This diagram defines a configuration of a system with two components whose interfaces are *A* and *B*. To make explicit the connections between the two components, we use a third interface *Channel*. The reason we call this third interface *Channel* is that it can be viewed as a natural extension of the interconnection services that are made available through the

interfaces (input and output channels and actions), i.e. a kind of structured or complex channel. Using the analogy with *hardware*, the channel and the two morphisms act as a "cable" that can be used to interconnect components.

More concretely, the diagram defines that the input channel u of B is connected with the output channel o of A , and that the input channels i of A and v of B are identified, i.e., in the resulting system, A and B input from the same source. Furthermore, the configuration establishes that A and B synchronise on actions a and b . Indeed, the colimit of the diagram depicted above returns



The action ab models the simultaneous execution of a in A and b in B . In the same way, action $a b$ models the simultaneous execution of a in A and b in B . However, because A and B do not have to synchronise on actions a and b , the interface AB has also two actions — a and b — modelling the isolated execution of a in A and b in B .

3 Horizontal vs Vertical Structuring

In the categorical approach that we have described in the previous section, morphisms are used for modelling horizontal structuring, i.e. to support the process of structuring complex systems by interconnecting simpler components. As we mentioned, a morphism identifies how its source is a component of its target.

However, for supporting software development, other relationships need to be accounted for that reflect notions of refinement between different levels of abstraction, e.g. [22,23,16]. In some cases, the morphisms used for capturing interconnections between components can also be used for expressing refinement. For instance, this is the case in the algebraic specification of abstract data types [2] and in the specification formalism of reactive systems presented in [8]. However, this is not always the case.

On the one hand, interconnection morphisms may serve the purpose of expressing refinement but may be too weak to represent the refinements that can occur during software development. For instance, this is the case in the algebraic approach proposed in [20] and refined in SPECWARE [23]. In this approach, interconnection morphisms are theorem-preserving translations of the vocabulary of a source specification into the terms of a target specification, whereas refinement morphisms are interconnection morphisms from the source specification to a conservative/definitional extension of the target specification (a refinement morphism from A to B is a pair of interconnection morphisms $A \rightarrow AasB \leftarrow B$).

On the other hand, morphisms used for interconnecting components may not be suitable for expressing refinement. In particular, this happens when a system is not necessarily refined by a system of which it is a component. For instance, in CSP [15] the functionality of a system is not necessarily preserved when it is composed with other systems. A process $a.P \parallel b.Q$, that is ready to execute a or b , has no longer this property if it is composed with a process that is only ready to execute a . Because the notion of refinement in CSP (based on the readiness semantics) requires that readiness be preserved by refinement, $a.P \parallel b.Q$ is not refined by $(a.P \parallel b.Q) \parallel a.R$.

Because of this, the integration of both dimensions – horizontal (for structuring) and vertical (for refinement) – may require that two different categories be considered.

Definition 2 (Design Formalism).

A design formalism is a triple $\langle c\text{-DESC}, Conf, r\text{-DESC} \rangle$ where $c\text{-DESC}$ and $r\text{-DESC}$ are categories and $Conf$ is a configuration criterion over $c\text{-DESC}$ s.t. 1. $|c\text{-DESC}| = |r\text{-DESC}|$ and 2. $Isom(c\text{-DESC}) \subseteq Isom(r\text{-DESC})$.

The objects of $c\text{-DESC}$ and $r\text{-DESC}$, which are the same, identify the nature of the descriptions that are handled by the formalism. The morphisms of $r\text{-DESC}$ identify the vertical structuring principles, i.e., a morphism $\eta: S \rightarrow S$ in $r\text{-DESC}$ expresses that S refines S , identifying the design decisions that lead from S to S . The morphisms of $c\text{-DESC}$ identify the horizontal structuring principles, that is to say, diagrams in $c\text{-DESC}$ can be used to express how a complex system is put together through the interconnection of components. The description of the system that results from the interconnection is the description returned by the colimit of the configuration diagram and, hence, it is defined up to an isomorphism. Condition 2 ensures that the notion of refinement is "congruent" with the notion of isomorphism in $c\text{-DESC}$. That is, descriptions that are isomorphic with respect to interconnections refine, and are refined exactly by, the same descriptions. Finally, as explained in the previous section, the relation $Conf$ defines the correct configurations.

In order to illustrate that different categories may need to be used to formalise the horizontal and vertical structuring principles supported by a formalism, we use a language for the design of parallel programs –CommUnity.

CommUnity, introduced in [5], is similar to Unity [3] in its computational model but has a different coordination model. More concretely, the interaction between a program and the environment in CommUnity and Unity are dual: in Unity it relies on the sharing of memory and in CommUnity in the sharing (synchronisation) of actions.

A design in CommUnity is of the following form:

```

design P is
in      I
out     O
init    Init
do       $\coprod_{g \in \Gamma} g: [L(g), U(g) \rightarrow \coprod_{v \in \mathcal{D}(g)} v: \in F(g, v)]$ 

```

The sets of input and output channels, respectively, I and O , and the set of actions Γ constitute the interface of design P . $Init$ is its initialisation condition. For an action g ,

- $D(g)$ represents the set of channels that action g can change.
- $L(g)$ and $U(g)$ are two conditions s.t. $L(g) \supset U(g)$ that establish an interval in which the enabling condition of any guarded command that implements g must lie. $L(g)$ is a lower bound for enabledness in the sense that it is implied by the enabling condition (its negation establish a *blocking* condition) and $U(g)$ is an upper bound in the sense that it implies the enabling condition. When $L(g)=U(g)$ the enabling condition is fully determined and we write only one condition.
- for every channel v in $D(g)$, $F(g,v)$ is a non-deterministic assignment: each time g is executed, v is assigned one of the values denoted by $F(g,v)$.

Formally,

A design is a pair $\langle \theta, \Delta \rangle$ where θ is an interface and Δ , the body of the design, is a 5-tuple $\langle \text{Init}, D, F, L, U \rangle$ where:

Init is a proposition over the output channels;

- D assigns to every action a set of output channels;
 F assigns to every action g a non-deterministic command (when $D(g)=\emptyset$, the only available command is the empty one which we denote by *skip*);
 L and U assign to every action g a proposition.

An example of a CommUnity design is the following bank account, where $CRED$ is a given negative number, modelling the credit limit.

```

design BankAccount is
in   amount: nat
out  bal: int
init bal=0
do   deposit: [true  $\rightarrow$  bal:=bal+amount]
      [] withdraw: [bal-amount $\geq$ CRED, bal-amount $\geq$ 0 $\rightarrow$ bal:=bal-amount]

```

This design has only two actions modelling the deposit and the withdrawal of a given amount. This amount is transmitted to the component via the input channel *amount*. Deposits are always available for clients. Withdrawals are definitely refused if the requested amount is over the credit limit ($bal-amount < CRED$) and are definitely accepted when the balance is sufficient to cover the requested amount, i.e. the credit facility is not necessary ($bal-amount \geq 0$). In all the other situations, i.e. when the credit facility is necessary ($0 > bal-amount \geq CRED$), it was left unspecified whether withdrawals are accepted or refused. This form of underspecification (also called allowed non-determinism), as we will see, can be restricted during a refinement step.

In the previous section we have seen how the interconnection of components can be established at the level of interfaces. The corresponding configurations of designs are expressed in the following category of designs.

A design morphism $\sigma: P_1 \rightarrow P_2$ is an interface morphism from θ_1 to θ_2 s.t.:

1. For all $g \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$;
2. For all $v \in O_1$, $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$;
3. For all $g_2 \in \Gamma_2$, if $\sigma_{ac}(g_2)$ is defined then, for all $v_1 \in D_1(\sigma_{ac}(g_2))$,
 $\models (F_2(g_2, \sigma_{ch}(v_1)) \subseteq \sigma(F_1(\sigma_{ac}(g_2), v_1)))$;

4. $\models (I_2 \supset \underline{\sigma}(I_1))$;
5. For every $g_2 \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\models (L_2(g_2) \supset \underline{\sigma}(L_1(\sigma_{ac}(g_2))))$;
6. For every $g_2 \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\models (U_2(g_2) \supset \underline{\sigma}(U_1(\sigma_{ac}(g_2))))$.

*Designs and design morphisms constitute a category **c-DSGN**.*

A design morphism $\sigma: P_1 \rightarrow P_2$ identifies P_1 as a component of P_2 . Conditions 1 and 2 mean that the domains of channels are preserved and that an action of the system that does not involve an action of the component cannot transmit in any channel of the component. Conditions 3 and 4 correspond to the preservation of the functionality of the component design: (3) the effects of the actions can only be preserved or made more deterministic and (4) initialisation conditions are preserved. Conditions 5 and 6 allow the bounds that the design specifies for enabling condition of the action to be strengthened but not weakened. Strengthening of the two bounds reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. In other words, the enabling condition of a joint action is given by the conjunction of the enabling conditions of the corresponding actions in the involved components.

The correctness of a configuration of interconnected designs depends uniquely on the correctness of the underlying interconnection of interfaces (as defined in the previous section). Let **Int** be the forgetful functor from **c-DSGN** to **INTF**.

A diagram $\mathbf{dia}: \mathbf{I} \rightarrow \mathbf{c-DSGN}$ is a configuration diagram iff $\mathbf{dia}; \mathbf{Int}: \mathbf{I} \rightarrow \mathbf{INTF}$ is an interface configuration diagram.

It is interesting to notice that the category **c-DSGN** is not finitely cocomplete, namely O/O connections lead to design diagrams that may have no colimit. However, the existence of colimits is ensured for correct configurations.

In order to illustrate how a system design can be constructed from the design of its components, let us consider that every deposit of an amount that exceeds a given threshold must be signaled to the environment. This can be achieved by coupling the *BankAccount* with an *Observer* design.

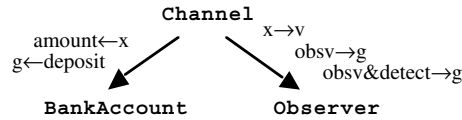
```

design Observer is
in   v: nat
out  sg: bool
init  $\neg$ sg
do   obsv&detect: [ $\neg$ sg  $\wedge$  v>NUMBER  $\rightarrow$  sg:=true]
      [] obsv: [ $\neg$ sg  $\wedge$  v $\leq$ NUMBER  $\rightarrow$  skip]
      [] reset: [sg  $\rightarrow$  sg:=false]

```

The *Observer* design can be used to "observe" the execution of a given action, namely to detect if $v > \text{NUMBER}$ holds when that action is executed. The design is initialised so as to be ready to execute *obsv&detect* or *obsv* depending on whether $v > \text{NUMBER}$ holds or not. The assignment of *obsv&detect* sets the output channel *sg* to true, thus signaling to the environment the execution of the "observed" action under the condition $v > \text{NUMBER}$. The environment may acknowledge the reception of the signal through the execution of the action *reset*.

The diagram below gives the configuration of the required system. This diagram defines that the action that is "observed" by *Observer* is *deposit* and identifies the input channel v of the *Observer* with input channel *amount* of the *BankAccount*. That is to say, in this system the *Observer* signals the deposits of *amounts* greater than *NUMBER*.



It remains to define the notion of refinement supported by CommUnity.

A refinement morphism $\sigma: P_1 \rightarrow P_2$ is an interface morphism from θ_1 to θ_2 s.t. conditions 1 to 5 of the definition of design morphism hold and

6. $\sigma_{ch}(I_1) \sqsubseteq I_2$;
7. σ_{ch} is injective;
8. For every $g \in \Gamma_2$, $\sigma_{ac}^{-1}(g) \neq \emptyset$;
9. For every $g_1 \in \Gamma_1$, $\not\models (\underline{\sigma}(U_1(g_1))) \supset \bigvee_{\sigma_{ac}(g_2)=g_1} U_2(g_2)$.

A refinement morphism σ from P_1 to P_2 supports the identification of a way in which P_1 is refined by P_2 . Each channel of P_1 has a corresponding channel in P_2 and each action g of P_1 is implemented by the set of actions $\sigma_{ac}^{-1}(g)$ in the sense that $\sigma_{ac}^{-1}(g)$ is a menu of refinements for action g . The actions for which σ_{ac} is left undefined (the new actions) and the channels that are not in $\sigma_{ch}(V_1)$ (the new channels) introduce more detail in the description of the design.

Conditions 6 to 8 express that refinement cannot alter the border between the system and its environment. More precisely, input channels cannot be made local by refinement (6), different channels cannot be collapsed into a single one (7) and every action has to be implemented (8). Condition 9 states that conditions $U(g)$ can be weakened but not strengthened. Because condition 5 allows L to be strengthened, the interval defined by the conditions L and U of each action, in which the enabling condition of any guarded command that implements the action must lie, is required to be preserved or reduced during refinement. Finally, conditions 3 and 4 state that the effects of the actions of the more abstract design, as well as the initialisation condition, are required to be preserved or made more deterministic.

Preservation of required properties and reduction of allowed non-determinism are intrinsic to any notion of refinement, and justify the conditions that we have imposed on refinement morphisms. Clearly, the morphisms that we used for modelling interconnections do not satisfy these properties.

In order to illustrate refinement, consider another design of a bank account — *BankAccount2*. In this design, the history of the client's use of the credit facility is taken into account for deciding if a withdrawal is accepted or not. It is not difficult to see that

$$\begin{array}{ll}
 \sigma_{ch} \quad bal \rightarrow bal & \sigma_{ac} \quad deposit \rightarrow deposit \\
 \quad \quad \quad amount \rightarrow amt & \quad \quad n_withd \rightarrow withdraw \\
 & \quad \quad s_withd \rightarrow withdraw \\
 & \quad \quad reset \rightarrow
 \end{array}$$

define a refinement morphism from *BankAccount* to *BankAccount2*.

```

design BankAccount2 is
in   amt: nat
out  bal, count: int
init bal=0^count=0
do   deposit: [true → bal:=bal+amt]
      [] n_withd: [bal-amt≥0→bal:=bal-amt]
      [] s_withd: [bal-amt≥CRED^count≤MAX→bal:=bal-amt||count:=count+1]
      [] reset: [true,false→count:=0]

```

The conditions under which withdrawals are available to clients, that was left unspecified in *BankAccount*, is completely defined in *BankAccount2*. The action *withdraw* of *BankAccount* is implemented in *BankAccount2* by *n_withd* and *s_withd*, where *n* and *s* stand for normal and special, respectively. Special withdrawals are counted and, as soon as their number reaches a given maximum, they are no longer accepted. Special withdrawals are accepted again if this restriction is lifted, which is modelled by the execution of the new action *reset*. Notice that *BankAccount2* also contains allowed non-determinism –the upper bound of enabledness of *reset* is false and, hence, it was not made precise in which situations the restriction is lifted.

4 Compositionality

Compositionality is a key issue in the design of complex systems because it makes it possible to reason about a system using the descriptions of their components at any level of abstraction, without having to know how these descriptions are refined in the lower levels (which includes their implementation). Compositionality is usually described as the property according to which the refinement of a composite system can be obtained by composing refinements of its components. However, when component interconnections are explicitly modelled through configurations, the refinement of the interconnections has to be taken into account. In these situations, the compositionality of a formalism can only be investigated w.r.t. a given notion of refinement of interconnections. Such notion, together with the refinement principles defined by *r-DESC*, establishes the refinement of configurations. Configurations and their refinement principles can be themselves organised in a category as follows.

Definition 3 (Configuration Refinement Criterion).

A configuration refinement criterion for a design formalism $\langle c-DESC, Conf, r-DESC \rangle$ is any category *CONF* that satisfies the following conditions:

1. The objects of *CONF* are the finite diagrams in *c-DESC*;
2. If $\eta: dia \rightarrow dia$ is a morphism in *CONF* and *I* and *I* are the shapes of, respectively, *dia* and *dia*, then η is an $|I|$ -indexed family of morphisms $\eta_i: dia(i) \rightarrow dia(i)$ in *r-DESC*.

Because diagrams are functors, it may seem that diagram morphisms should be merely natural transformations (as they are, for instance, in SPECWARE). However, this would be very restrictive because it would enforce that the shape of diagrams that define configurations be preserved during refinement.

Once we have fixed such a configuration refinement criterion *CONF*, compositionality can be formulated as the property according to which we can pick arbitrary refinements of the components of a system *Sys*, and interconnect these more concrete descriptions with arbitrary refinements of the connections used in *Sys*, and obtain a system that still refines *Sys*.

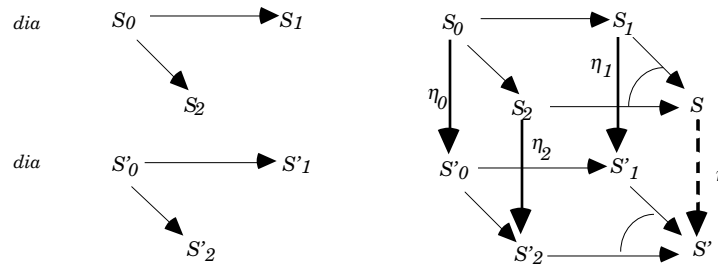


Figure 1

More precisely, if *dia* and *dia'* define the configuration of, respectively, *Sys* and *Sys'*, and $\eta: \mathbf{dia} \rightarrow \mathbf{dia}'$ is a morphism in *CONF*, then η defines, in a unique way, a refinement morphism in *r-DESC* from *Sys* to *Sys'* whose design decisions are compatible with the design decisions that were taken in the refinement of the components (see figure 1). That is to say, there exists a unique refinement morphism η in *r-DESC* from *Sys* to *Sys'* s.t. η is compatible with each of the morphisms $(\eta_i: \mathbf{dia}(i) \rightarrow \mathbf{dia}'(i))_{i \in |I|}$.

Once again, it is not possible to define the meaning of "compatibility of design decisions" independently of the context. As happens with the notion of refinement of interconnections, the meaning of "compatibility of design decisions" must be fixed *a priori*.

The compatibility of η and η_i depends on the morphisms that identify $\mathbf{dia}(i)$ and $\mathbf{dia}'(i)$ as components of, respectively, *Sys* and *Sys'*. In this way, we consider that the compatibility of design decisions is abstracted as a 4-ary relation between *Mor(r-DESC)*, *Mor(c-DESC)*, *Mor(r-DESC)* and *Mor(c-DESC)*.

Definition 4 (Criterion of Compatibility of Design Decisions).

A criterion of compatibility of design decisions for a design formalism $\langle \mathbf{c-DESC}, Conf, \mathbf{r-DESC} \rangle$ is a 4-ary relation *Crt* between *Mor(r-DESC)*, *Mor(c-DESC)*, *Mor(r-DESC)* and *Mor(c-DESC)* such that

1. If $Crt(\eta_i, \mu_i, \eta, \mu_i)$ then $dom(\eta_i) = dom(\mu_i)$, $dom(\mu_i) = cod(\eta_i)$, $dom(\eta) = cod(\mu_i)$ and $cod(\eta) = cod(\mu_i)$;
2. If $Crt(\eta_i, \mu_i, \eta, \mu_i)$ and $\kappa: cod(\eta) \rightarrow S''$ is a morphism in *c-DESC* and also in

r-DESC, then $Crt(\eta_i, \mu_i, (\eta; \kappa), (\mu_i; \kappa))$;

3. If $dia: I \rightarrow c-DESC$ is a configuration diagram, $(\mu_i: dia(i) \rightarrow S)_{i \in |I|}$ is a colimit of dia and $Crt(\eta_i, \mu_i, \eta, \mu_i)$, $Crt(\eta_i, \mu_i, \eta, \mu_i)$ for every $i \in |I|$, then $\eta = \eta$.

The idea is that $Crt(\eta_i: S_i \rightarrow S, \mu_i: S_i \rightarrow S, \eta: S \rightarrow S, \mu_i: S_i \rightarrow S)$ expresses that the design decisions defined by η agree with the decisions defined by η_i , taking into account the morphisms that identify S_i as a component of S and S'_i as a component of S' (see Fig. 2).

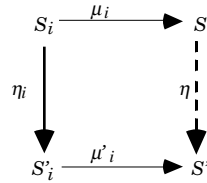


Figure 2

Condition 1 states the obvious restrictions on the domains and codomains of the four morphisms. Condition 2 ensures that the criterion is consistent w.r.t. the composition of morphisms that are simultaneously refinement and interconnection morphisms. Condition 3 means that if the refinement of the components of a system Sys is given by $(\eta_i: S_i \rightarrow S)_{i \in |I|}$, and Sys is a system having $(S_i)_{i \in |I|}$ as components, then there exists a unique way in which Sys may be a refinement of Sys .

Definition 5 (Compositional Formalism).

Let $Fd = \langle c-DESC, Conf, r-DESC \rangle$ be a design formalism, $CONF$ a configuration refinement criterion for Fd and Crt a criterion of compatibility of design decisions for Fd . Fd is compositional w.r.t. $CONF$ and Crt iff, for every morphism $\eta: dia \rightarrow dia$ in $CONF$, there exists a unique morphism $\eta: S \rightarrow S$ in $r-DESC$ s.t. $Crt(\eta_i, \mu_i, \eta, \mu_i)$, for every $i \in |I|$, where $(\mu_i: dia(i) \rightarrow S)_{i \in |I|}$ and $(\mu_i: dia(i) \rightarrow S)_{i \in |I|}$ are colimits of, respectively, dia and dia .

The notion of compositionality had been investigated and characterised in [8] for the situation in which the interconnection and refinement morphisms coincide. More concretely, when

- $c-DESC = r-DESC$;
- a refinement morphism η from the diagram $dia: I \rightarrow c-DESC$ to the diagram $dia: I \rightarrow c-DESC$ exists iff I is a subcategory of I and, in this case, η is an $|I|$ -indexed family $(\eta_i: dia(i) \rightarrow dia(i))_{i \in |I|}$ of morphisms such that $dia(f); \eta_j = \eta_i; dia(f)$, for every morphism $f: i \rightarrow j$ in I ;
- the criterion of compatibility of design decisions Crt is defined by $Crt(\eta_i, \mu_i, \eta, \mu_i)$ iff $\mu_i; \eta = \eta_i; \mu_i$;

compositionality is just a consequence of the universal property of colimits.

Figure 3 shows an example with the interconnection of two components through a channel. In this case, if $\sigma_i; \eta_i = \eta_0; \sigma_i$, for $i=1,2$, then the universal property of colimits ensures that there exists a unique morphism η from S to S such that $\mu_i; \eta = \eta_i; \mu_i$, for

$i=1,2$. In this case, notice that the interconnection of the two components defined by $\langle \sigma_1, \sigma_2 \rangle$ and the refinement morphisms $\langle \eta_1, \eta_2 \rangle$ define an interconnection of the more concrete descriptions S_1 and S_2 , that is given by $\langle \sigma_1; \eta_1, \sigma_2; \eta_2 \rangle$. This new configuration is, by definition, a refinement of the initial one (where η_0 is the identity).

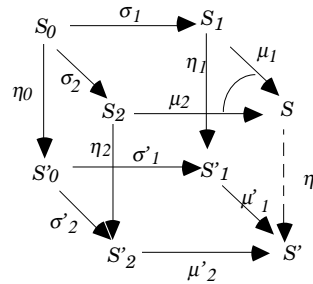


Figure 3

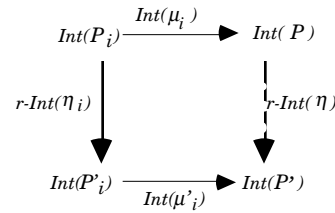


Figure 4

For illustrating the characterisation of compositionality we have proposed, we use CommUnity again. We define informally in which conditions a configuration diagram $dia : I \rightarrow c\text{-DSGN}$ is a refinement of another configuration diagram $dia : I \rightarrow c\text{-DSGN}$, given that the refinement of the components is defined by $(\eta_i: dia(i) \rightarrow dia'(i))$.

Basically, the morphisms of $CONF^P$ define that the system architecture, seen as a collection of components and "cables", cannot change during a refinement step ($I=I'$). However, the "cables" used to interconnect the more abstract designs can be replaced by "cables" with more capabilities. More precisely, the replacement of an interconnection dia by dia' must satisfy the following conditions:

- The interconnection dia' must be consistent with dia , i.e., the i/o communications and synchronisations defined by dia have to be preserved.
- The diagram dia' cannot establish the instantiation of any input channel that was left unplugged in dia . That is to say, the input channels of the composition are preserved by refinement.
- The diagram dia' cannot establish the synchronisation of actions that were defined as being independent in dia .

For the compatibility of design decisions Crt^P we simply consider the commutability of the diagram in $INTF$ depicted in Figure 4.

The design formalism CommUnity is compositional w.r.t. $CONF^P$ and Crt^P .

The formal definitions as well as the proof of this result can be found in [18].

It is important to notice that the adoption of different configuration refinement criteria gives rise to different notions of compositionality. For instance, the criterion $CONF^P$ we have defined for CommUnity prevents the addition of new components to the system (*superposition*) during a refinement step, even if they are just "observers", i.e., they do not have effects in the behaviour of the rest of the system. However, it is also possible to prove the compositionality of CommUnity w.r.t. a configuration refinement criterion that

allows the addition of new components to the system. For simplicity, we have decided not to present this other criterion (which is more difficult to express because it requires further conditions over the new components and the new "cables").

5 Conclusions

In this paper, we proposed a revision of Goguen's categorical approach to systems design in order to make it applicable to a larger number of design formalisms. The new aspects of the revised approach are the ability to represent the specific rules that may govern the interconnection of components and the ability to support the separation between horizontal and vertical structuring principles. Furthermore, we proposed a characterisation of compositionality of refinement with respect to composition in this revised categorical framework. The explicit description of component interconnections leads us to a definition of compositionality that depends on a notion of refinement of interconnections and on the meaning of "compatibility of design decisions".

The work reported in this paper was motivated by our experience in using the categorical platform to express sophisticated formalisms, namely formalisms for architectural design [17]. For instance, we have perceived that the property according to which a framework for system design supports the separation between computation and coordination (which is a good measure of the ability of a formalism to cope with the complexity of systems) is, in general, false if *ad hoc* interconnections are permitted. In a similar way, interconnections of designs can be synthesised from interconnections of interfaces in coordinated frameworks only if interconnections are severely restricted. From this experience, the need for expressing the correctness of components configurations emerged as reported in the paper.

Further work is going on which explores the impact of this revision on the use of the categorical platform to map and relate different formalisms. As showed in [6], the categorical platform can be used to formalise the integration of different formalisms. This integration is important because it supports the use of multiple formalisms in software development and promotes reuse.

References

1. R.Alur and T.Henzinger, "Reactive Modules", in *Proc. LICS 96*, 207-218.
2. R.Burstall and J.Goguen, "Putting Theories Together to Make Specifications", in *Proc. 5th IJCAI*,1045-1058, 1977.
3. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley, 1988.
4. T.Dimitrakos, "Parametrising (algebraic) Specification on Diagrams", *Proc. 13th Int.Conference on Automated Software Engineering*, 1998.

5. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, 28:111-138, 1997.
6. J.L.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed), *Proc. 3rd Symp. on Foundations of Software Engineering*, 72-80, ACM Press, 1995.
7. J.L.Fiadeiro and T.Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification", in *Formal Aspects of Computing* 4(3), 1992, 239-272.
8. J.L.Fiadeiro, "On the Emergence of Properties in Component-Based Systems", in M.Wirsing and M.Nivat (eds), *AMAST 96*, LNCS 1101, Springer-Verlag 1996, 421-443.
9. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination", in A.Haeberer (ed), *AMAST 98*, LNCS 1548, Springer-Verlag.
10. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in M.Bidoit and M.Dauchet (eds), *TAPSOF 97*, LNCS 1214, 505-519, Springer-Verlag, 1997.
11. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trapp (eds), *Advances in Cybernetics and Systems Research*, 121-130, Transcripta Books, 1973.
12. J.Goguen and S.Ginalli, "A Categorical Approach to General Systems Theory", in G.Klir(ed), *Applied General Systems Research*, Plenum 1978, 257-270.
13. J.Goguen, "Sheaf Semantics for Concurrent Interacting Objects", *Mathematical Structures in Computer Science* 2, 1992.
14. M.Gro e-Rhode, "Algebra Transformations Systems and their Composition", in E.Astesiano (ed), *FASE 98*, LNCS 1382, 107-122, Springer-Verlag, 1998.
15. C.A.Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
16. K.Lano and A.Sanchez, "Design of Reactive Control Systems for Event-Driven Operations", in J.Fitzgerald, C.Jones and P.Lucas (eds), *Formal Methods Europe 1997*, LNCS 1313, 142-161, Springer-Verlag, 1997.
17. A.Lopes and J.L.Fiadeiro, "Using Explicit State to Describe Architectures", in J.Finance (ed), *FASE 99*, LNCS 1577, 144-160, Springer-Verlag, 1999.
18. A.Lopes, *Non-determinism and Compositionality in the Specification of Reactive Systems*, PhD Thesis, University of Lisbon, 1999.
19. J.Magee and J.Kramer, "Dynamic Structures in Software Architecture", in *4th Symposium on Foundations of Software Engineering*, ACM Press 1996, 3-14.
20. T.Maibaum, P.Veloso and M.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?", in H.Ehrig, C.Floyd, M.Nivat and J.Thatcher (eds) *TAPSOF 85*, LNCS 186, 1985, 214-230.
21. Z.Manna and A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
22. D.Smith, "Constructing Specification Morphisms", *Journal of Symbolic Computation* 15 (5-6), 571-606, 1993.
23. Y.Srinivas and R.J.lli, "Specware :Formal Support for Composing Software", in B.M.ller (ed) *Mathematics of Program Construction*, LNCS 947, 399-422, Springer-Verlag, 1995.
24. G.Winskel, "A Compositional Proof System on a Category of Labelled Transition Systems", *Information and Computation* 87:2-57, 1990.
25. G.Winskel and M.Nielsen, "Models for Concurrency", *Handbook of Logic in Computer Science*, S.Abramsky, D.Gabbay and T.Maibaum (eds), Vol.4, 1-148, Oxford University Press 1995.
26. V.Wiels and S. Easterbrook, "Management of Evolving Specifications using Category Theory", *Proc. 13th Int.Conference on Automated Software Engineering*, 1998.

Proof Transformations for Evolutionary Formal Software Development

Axel Schairer and Dieter Hutter

German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
{schairer,hutter}@dfki.de

Abstract. In the early stages of the software development process, formal methods are used to engineer specifications in an explorative way. Changes to specifications and verification proofs are a core part of this activity, and tool support for the evolutionary aspect of formal software development is indispensable.

We describe an approach to support evolution of formal developments by explicitly transforming specifications and proofs, using a set of predefined basic transformations. They implement small and controlled changes both to specifications and to proofs by adjusting them in a predictable way. Complex changes to a specification are achieved by applying several basic transformations in sequence. The result is a transformed specification and proofs, where necessary revisions of a proof are represented by new open goals.

1 Introduction

It is widely accepted by now that formal methods can increase the dependability of software systems and the trust that users of the system can reasonably have. Although, often, formal methods have been associated with verifying the correctness of program code with respect to a specification, which was assumed to be given and adequate, this has changed in recent years. There is a shift of the focus of formal methods towards earlier phases in the development process with the aim of formally engineering the specifications themselves. It is argued that the benefit of using formal methods, e.g. being able to analyze the requirements specification and establish properties it implies, before the system is designed and implemented, by mechanized calculations rather than by subjective inspection and review, is particularly helpful in this context, cf. [18]. The shift towards using formal methods in earlier phases is reflected, e.g., in the requirements of IT security evaluation criteria like Common Criteria [3] and ITSEC [12], where formal security models are required for, e.g., assurance levels EAL5 and E4, respectively, for which no formal design specifications or verified program code are required.

In this context, formal methods are used to incrementally construct and validate security models, i.e. specifications together with their properties. Changes to the specification, properties, and proofs are a core part of the development

activity rather than an accident that should have been avoided, and proper tool support for an evolutionary formal software development is indispensable in practice. Necessary tool support includes management of change for large structured specifications, which formalize both an abstract system model and requirements, and management of change on the level of the proofs that are associated with the system model and its properties.

In this paper, we describe an approach to cope with changes to specifications by transforming a given specification and associated proofs. Transformations correspond to atomic changes on the specification and extend to the associated proofs in a way that is simple and easy to understand by the user. This is achieved by tailoring each transformation to the kind of change it is associated with and using the additional information available then. Since transformations intentionally do not depend on heuristic search and are deterministic, they provide a solid tool which is easy to control and understand, and with which the user can develop specifications together with proofs in an evolutionary manner.

The rest of this paper is organized as follows. After describing evolutionary formal software development in Sect. 2, Sect. 3 introduces transformations on developments. Sect. 4 then describes a concrete example set of basic transformations and their effects. In Sect. 5, we compare our approach to related work. Finally, we describe future work and draw conclusions in Sect. 6.

2 Evolutionary Formal Software Development

The state of a formal software development is represented by a specification, i.e. a system model and its desired properties, together with (partial) proofs for the postulated properties. Typically, such specifications are formulated in a specification language that allows for the structured presentation of the specification, e.g. CASL [11] or VSE-SL [1]. Proof obligations are derived mechanically from the specification when, e.g., one theory is postulated to be satisfied by another theory, see e.g. [2]. A mechanized reasoning module is then used to discharge of the proof obligations, i.e. constructing proofs. The state of a development changes over time, as the specification is edited and proofs are constructed. A typical mode of development is to tackle proof obligations, and use the insight gained from failed proof attempts to correct or extend the specification, e.g. [20, Sect. 7].

On a different scale, complex systems are built in a stepwise manner. First a simple model and its properties are studied, and over time additional complexity is added by integrating features into the system model and accordingly modifying the properties. Obviously, many of the old proofs need to be reconsidered in the light of the changes and need to be adjusted to fit the new specification.

As an example we consider the case study [15] on modeling reliable broadcast using the technique of fault transformations described in [8]. First a network of processes running concurrently is formalized and verified. Processes, connected by directed channels, run a local non-deterministic program to deal with arriving messages and to resend them. It is verified that only messages that have been

broadcast to the network are delivered to processes, and that they are delivered to each process at most once. Throughout the rest of the paper, we will refer to simplified excerpts of a reconstruction of the example in [15].

The system model used in [15] is that of a state transition system described by an initial state and possible state transitions that the processes can take, called actions. The state transition system is explicitly specified using axiomatic specifications of abstract datatypes. The proof obligation corresponding to the safety property roughly reads

$$\forall tr : Trace. admissible(tr) \Rightarrow safety(res(tr)) \quad (1)$$

where $res(tr)$ is the resulting state after the actions in tr have been taken and $safety$ is defined by

$$\forall s : State. safety(s) \Leftrightarrow \quad (2)$$

$$\left\{ \begin{array}{l} (\forall p : Proc. p \in Procs(s) \Rightarrow delivered(p, s) \subseteq broadcast(s)) \\ \wedge (\forall p : Proc. p \in Procs(s) \Rightarrow nodups(delivered(p, s))) \end{array} \right.$$

I.e. each admissible sequence of states ends in a state satisfying the safety property. The proof is by induction over the length of traces. In the step case, the proof is split by a case distinction over possible actions, and for each action it is shown that it conserves the safety property.

In a second step, fault assumptions are added to the specification: processes are no longer assumed to work reliably, rather each process can either be up or down. This is done by adding parts to the specification related to the added functionality, but it also involves changing parts of the existing specification, cf. [15, Sect. 4.1]. In particular, the representation of processes in the state is changed to hold additional information about whether the respective process is up or down. Also, an additional crash action is added. It can be executed by running processes, and its effect is to crash the process. All other actions are restricted to be executable by running processes only, thereby strengthening the actions' preconditions. Obviously, after these changes the safety property given by (1) and (2) does no longer hold and the verification proofs are no longer valid. According to the methodology of failure transformations, the property is replaced by a weaker property: in (2), $delivered(p, s) \subseteq broadcast(s)$ is weakened to $isup(p) \Rightarrow delivered(p, s) \subseteq broadcast(s)$, where a definition of $isup(p)$ has to be added as well. As a consequence of these changes, the proof for (1) now has an additional case for the crash action. Also, it remains to be shown that the weaker safety property is sufficient for the induction to go through in the cases for all other actions (the induction hypothesis is weakened, too, so this is not obvious). This is done by changing the subproofs for these actions to take the additional preconditions into account. The proof idea and the overall structure are essentially unchanged.

There are, however, a number of technical consequences that are trivial to hand-wave over in an informal description of the changes but which are hard to carry out efficiently and reliably in mechanized theorem provers. Examples for

this include changing existing symbols of the signature and induction schemata. Obviously, this requires changing proofs.

In this paper, we offer transformations that cope with changes described above, taking care of technical details in a way that allows changes occurring in evolutionary formal development to be carried out as a matter of routine.

3 Development Transformations

We view the process of formal evolutionary software development as a sequence D_1, \dots, D_n of self-contained formal *development states*. The correctness of the final development state D_n does not depend on the correctness of previous states. D_i is transformed into D_{i+1} by a *transformation*. We require that each D_i is statically well-formed in the sense that, e.g. no syntax or static type errors are present in the specification, and that each D_i associates a valid (although maybe partial) proof to each of its proof obligations. Transformations are, therefore, required to conserve these properties. Note that *any* change to a specification such that it is again statically well-formed and associating trivial open proofs to all proof obligations is a transformation. It is uninteresting, however, because all effort spent on previous proofs is lost. For some changes to the specification this may be the best we can do, but for many changes found in practice, we can do much better, as was demonstrated in the previous section, where virtually all of the proofs could be kept, albeit on the proviso that additional open goals in the new proofs had to be tackled. In the example of the previous section, it was found that a new subproof for the crash action was needed and that there were gaps in the subproofs for other actions related to the strengthened preconditions.

Of course, specific transformations depend on the concrete choice of concepts a specification language offers, the relationship between specifications and proof obligations, and the notion of proofs. This paper concentrates on the problem of adjusting formal proofs to changed specifications. Since theorem provers do usually not make use of the structure information incorporated into the specification we restrict ourselves to basic unstructured specifications and assume that proof obligations are generated for lemmata, which are explicitly given as part of the specification. However, in order to come up with all inclusive system for evolutionary formal software development it will be necessary to develop techniques to propagate the changes of a local part of the specification to other potentially affected part. We have done first steps in this direction, however, this is not inside the focus of this paper.

A specification consists of a signature, axioms, and postulated lemmata or theorems. A signature Σ is determined by a set of type, function, and predicate definitions. A type definition defines τ to be an uninterpreted type, and

$$\tau = c_1 : \tau_{1,1} \times \dots \times \tau_{1,k_1} \rightarrow \tau \mid \dots \mid c_n : \tau_{n,1} \times \dots \times \tau_{n,k_n} \rightarrow \tau$$

defines τ to be a type generated by the constructor functions c_i . $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau_n$ defines the function f with the given arity, and similarly $p : \tau_1 \times \dots \times \tau_n$ for the predicate p . Axioms and lemmata are closed first order formulae over Σ .

The semantics of a specification is the class of all Σ -models which satisfy the generatedness constraints and all axioms and lemmata, cf. [14].

For each conjectured lemma A a proof obligation $\Gamma \vdash A$ is generated, where Γ is the set of axioms and lemmata that occur in the specification before A . It asserts that A is implied by the conjunction of formulae in Γ . For the example of Sect. 2, *Trace*, *State*, *admissible*, or *safety* are given by the signature, and their definitions, e.g. (2), by axioms, whereas the property (1) is given last as a lemma.

Changes to the specification can have the following consequences on the proof obligations.

- The language for the assumptions and the formula to be proven can be changed by adding or removing signature definitions, or by changing existing ones.
- Additional proof obligations can be generated or proof obligations can disappear by adding or removing lemmata.
- Assumptions can be added or removed from proof obligations by adding or removing axioms or lemmata.
- Formulae in proof obligations can change.
- Induction schemata can be changed by changing the definition of generated types.

In general, changing the specification will have more than one of the effects mentioned above.

It is not feasible to provide separate specialized transformations for each particular domain transformation, e.g. for introducing one particular fault transformation as was described in Sect. 2. Instead, we provide a set of *basic transformations* that are specific to the specification language, the way proof obligations are generated, and the notion of proof, but not specific to an application domain. Basic transformations can be used in sequence to achieve the effect of a domain specific transformation. Each basic transformation maps specification and proofs to a new consistent state, though potentially new open goals will occur in proofs and some parts of old proofs will be missing. In this new state, another transformation can be applied, or the proofs can be inspected and changed as usual with the theorem prover.

4 Basic Transformations

As the result of working through examples, we have come up with a set of basic transformations on specifications given in Fig. 1. In the rest of this section we will describe some of the transformations in more detail and show how they are employed in the case study introduced in Sect. 2.

We use an analytic sequent calculus for first order logic, cf. [7] with equality and induction rules. The representation for proofs is a tree of nodes. Each node carries a sequent $\Gamma \vdash \Delta$ which means that the conjunction of formulae in the set Γ (antecedent) implies the disjunction of the formulae in the set Δ (succedent).

- *Add type, add function, add predicate* – add a definition to the signature
- *Remove type, remove function, remove predicate* – remove a definition from the signature
- *Rename signature item* – consistently rename a signature item throughout the whole specification
- *Add axiom, add lemma* – add an axiom or lemma to the collection of axioms and lemmata
- *Remove axiom, remove lemma* – remove an axiom or lemma from the collection of axioms and lemmata
- *Move axiom, move lemma* – move an axiom or lemma from one position in the collection of axioms and lemmata to another
- *Add constructor* – add a new constructor function to the list of constructors of an existing generated type
- *Remove constructor* – remove a constructor from the list of constructors of a generated type
- *Add argument, remove argument* – change the arity of a function or predicate by adding a new argument or by removing an argument from the definition
- *Swap arguments* – reorder the arity of a function or predicate by swapping two of its arguments
- *Replace occurrence* – replace the occurrence of a formula or term in an axiom or lemma by another formula or term
- *Wrap* (special case of *replace occurrence*) – replace a sub-formula A of an axiom or lemma by another formula B that has A wrapped somewhere inside
- *Unwrap* (special case of *replace occurrence*) – replace a sub-formula B of an axiom or lemma that has A wrapped inside by A

Fig. 1. Example set of basic transformations

The root node carries the proof obligation. Each node has associated with it a justification that indicates how the sequent of the node can be inferred from the sequents of its n children (where n may be 0). Justifications can be calculus rules or, for nodes without children, a special justification *open*, i.e. proofs may be partial. For valid proofs, we require for each node N with justification r and children nodes N_1, \dots, N_n that

$$\frac{S_{N_1} \quad \dots \quad S_{N_n}}{S_N} r$$

(where S_N is the sequent of node N) is a valid application of proof rule r . Note that proof rules may refer to signature items, formulae, or terms, e.g. for each generated type, there is a proof rule according to the induction schema for the type. We now describe aspects of sample basic transformations in groups corresponding to transformations' possible consequences given at the end of Sect. 2.

4.1 Extending/Restricting the Signature

A group of transformations, i.e. *add type*, *add function*, and *add predicate* and the corresponding *remove* transformations change the specification in a way that

is not reflected in the proof obligations. *Add* transformations are applicable only if the name of the signature item, e.g. a type, to be added is new in the relevant name space of the specification signature. Name conflicts with bound variables in axioms or lemmata of the specification can be avoided by α -renaming. The result of the transformation on the signature, terms and formulae, therefore, is an inclusion homomorphism. Semantically, for each model M of the new specification, its reduct with respect to the inclusion homomorphism is a model of the old specification. Existing proofs can be retained in the new specification: none of the proof steps in any of the old proofs can be invalidated by applying the homomorphism to the goals in the proof, unless name conflicts with Eigenvariables introduced in the proof arise. These can be avoided by naming the conflicting Eigenvariables away, either by implicit or explicit α -renaming of the Eigenvariables, depending on whether the Eigenvariables are implicitly or explicitly bound in the proof tree. In the example, new datatypes for dealing with crashed processes are added and functions and predicates are defined to operate on them, e.g. an enumeration type $UpDown = up \mid down$ and a predicate $isup : Proc \rightarrow UpDown$ are added.

The *remove* transformations are applicable only if the signature item to be removed is not used in the specification, i.e. a type τ can only be removed if none of the functions or predicates mentions τ in its arity, and if none of the axioms or lemmata quantifies over τ . The same restriction applies to formulae and terms introduced in the proof, e.g. no formula introduced by a cut rule may quantify over τ if the transformation is to be applicable. In this case, the inclusion homomorphism from the new signature to the old signature maps every formula or term in the axioms, lemmata, and proofs to itself. The old proofs are, therefore, proofs for the new proof obligations. In the example, several constants that were introduced by other transformations (cf. Sect. 4.2) were removed from the specification when they were no longer needed.

The transformation *rename signature item* is applicable when the new name of, e.g., a type does not already occur in the relevant name space. In this case, the renaming corresponds to an isomorphism between the old and the new signature, and similarly between the old and new specification. Semantically, this does not have any effects, since the concrete names of signature items do not matter for the meaning of terms and formulae. Modulo α -renaming of bound variables and Eigenvariables, the image of a proof under the isomorphism, i.e. the result of applying the isomorphism to terms and formula in the proof, is again a proof. In the example, signature items were renamed after they had been changed to address fault tolerance, e.g. type *Action* was renamed to *CrashAction*.

4.2 Changing Existing Signature Entries

The transformation *add argument* changes the signature of the specification by changing the arity of a function or predicate. This means that if, e.g., $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau$ is changed to $f : \tau_1 \times \dots \times \tau_n \times \tau_{n+1} \rightarrow \tau$, terms constructed with the function f will no longer be well-formed. There are also consequences concerning induction schemes when f is a constructor function,

cf. Sect. 4.5 below. For now, assume that f is a non-constructor function. Then replacing each term of the form $f(t_1, \dots, t_n)$ with a term $f(t_1, \dots, t_n, t)$, where t is an arbitrary but fixed term of type τ_{n+1} , in the specification, proof obligations, and proofs ensures all formulae are again syntactically correct. We add a new constant $a : \tau_{n+1}$ and use it for the term t . The transformed proof obligations correspond to the new specification and the transformed proofs are valid. This can be shown by induction over the tree structure of a proof, using the fact that whenever two formulae were equal before the transformation, they will again be equal afterwards. In the example, the arity of the function $mkproc$ was changed by adding an argument of type $UpDown$, representing information on whether the process is up or down. See Sect. 4.5 for more details.

4.3 Adding/Removing Proof Obligations and Assumptions

The Transformations *add axiom*, *add lemma*, *remove axiom*, and *remove lemma*, have direct and obvious consequences on the proof obligations. For each lemma there is exactly one proof obligation, so adding or removing lemmata adds or removes proof obligations. When a new proof obligation $\Gamma \vdash B$ is generated, no old proof is available for the new proof obligation, so a trivial open proof with root goal $\Gamma \vdash B$ and the justification *open* is associated with the new goal. When an old proof obligation is removed, the proof for it is removed together with the proof obligation. On the other hand, adding or removing axioms or lemmata adds or removes formulae from the assumptions of proof obligations for other lemmata that occur lexically after the axiom or lemma that was added or removed. In the fault tolerance example, defining axioms for *isup* are added, e.g.

$$\forall i : PID, b : BMsg, m : MMSet. isup(mkproc(i, b, m, up)) .$$

Let A be this formula. The proof obligation for (1) is changed by adding the new axiom A to the assumptions of the proof obligation. Throughout the whole proof ξ , it can be added to the antecedent of each node's goal sequent by the obvious transformation defined inductively over the size of the tree, without making any of the proof steps invalid (modulo α -renaming of Eigenvariables). In particular, each open goal $\Gamma \vdash \Delta$ of ξ is transformed into $\Gamma, A \vdash \Delta$, which means that proofs to be constructed for open goals can use the newly introduced axiom.

Conversely, if A were to be removed from the antecedent of a proof obligation $\Gamma, A \vdash B$, the part of its associated proof which does not refer to A can be kept by removing A from the antecedents of the goals. If A is the formula above, in the proof

$$\frac{\begin{array}{c} \vdots \xi_2 \\ \Gamma_1, \forall i, b, m. isup(mkproc(i, b, m, up)), isup(mkproc(i_1, b_1, m_1, up)) \vdash \Delta_1 \end{array}}{\Gamma_1, \forall i, b, m. isup(mkproc(i, b, m, up)) \vdash \Delta_1} \forall : l$$

$$\begin{array}{c} \vdots \xi_1 \\ \Gamma, \forall i, b, m. isup(mkproc(i, b, m, up)) \vdash B \end{array}$$

which does not use A in ξ_1 , the steps in ξ_1 remain valid steps if A is removed from the antecedent of all goals. The $\forall : l$ step is not a valid proof step without A , however. Instead of throwing the proof consisting of this step and ξ_2 away, A is introduced by a cut rule and can then be used in the proof consisting of the $\forall : l$ step and ξ_2 .

$$\frac{\Gamma_1 \vdash \Delta_1, A \quad \frac{\Gamma_1, A, \text{isup}(\text{mkproc}(i_1, b_1, m_1, up)) \vdash \Delta_1}{\Gamma_1, A \vdash \Delta_1} \forall : l}{\Gamma \vdash B} \text{cut}$$

In general, the result is a new proof with additional goals of the form $\Gamma' \vdash \Delta'$, A at each point in the proof tree, where A was used in the original proof. In particular, for proofs that do not use A at all, no additional goals are created. The transformations *move axiom* and *move lemma* are similar: moving an axiom adds it to some assumptions and removes it from others; moving a lemma additionally corresponds to adding or removing formulae from the assumptions of its proof obligation according to its new position in the specification relative to the old one.

4.4 Changing Formulae

The transformation *replace occurrence* replaces an occurrence of a formula in the specification by another one. As an example consider (2) where the subformula $\text{delivered}(p, s) \subseteq \text{broadcast}(s)$, call it $A(p, s)$, is replaced by the formula $\text{isup}(p) \Rightarrow A(p, s)$. The old proof for (1) uses the definition in (2) several times to expand the definition of *safety* for each subproof resulting from a case split over all possible actions. All these proof steps are still valid if only instances of $A(X, Y)$ are replaced by the corresponding instances of $\text{isup}(X) \Rightarrow A(X, Y)$ throughout the proof, since the steps do not depend on the concrete form of the formula. However, at some point, the concrete structure of A is used in the old proof, e.g. the top-level formula $A(p_1, s_1)$ is unified with another formula of the form $\tau_1 \subseteq \tau_2$ for some terms τ_1, τ_2 . This step is not valid if $A(p_1, s_1)$ is replaced by the new formula. At this point in the proof, the cut rule can be used to introduce the old formula as a top-level formula, resulting in an additional open goal of the form $\Gamma, \text{isup}(p_1) \Rightarrow A(p_1, s_1) \vdash \Delta, A(p_1, s_1)$. In this particular case, however, since it is known that $A(p, s)$ has been replaced by $\text{isup}(p) \Rightarrow A(p, s)$, a more specific transformation, i.e. *wrap connective* is applicable.

Wrap Connective. If a formula occurrence A is replaced by $A \circ B$ or $B \circ A$, where ‘ \circ ’ is a connective, and A has the same polarity in the new formula, the transformation *wrap connective* is applicable.¹ In the example, parts of a

¹ The polarity of A is conserved in $A \circ B$ and $B \circ A$ for $\circ \in \{\wedge, \vee\}$, and in $B \Rightarrow A$, but not in $\neg A$ or $A \Rightarrow B$.

simplified proof for (1) looks like the following, where parts of the proof for the base case of the induction and for all but one specific action a have been omitted.

$$\frac{\dots \quad \frac{\Gamma' \vdash A(p, \text{res}(tr.a)) \quad \dots}{\vdots \xi_2}}{\vdots \xi_1}}{\Gamma, \forall s. \text{safety}(s) \Leftrightarrow \forall p. A(p, s) \vdash \text{ad}(tr.a) \Rightarrow \text{safety}(\text{res}(tr.a))}$$

When $A(p, s)$ is replaced by $\text{isup}(p) \Rightarrow A(p, s)$, ξ_1 is transformed to ξ'_1 as described for the general *replace occurrence*. Then, the top-level formula $\text{isup}(p) \Rightarrow A(p, s)$ is decomposed by inserting a $\Rightarrow: r$ rule such that ξ_2 can be used if it is transformed to ξ'_2 as described for the case of adding assumptions above. This yields the following transformed proof.

$$\frac{\dots \quad \frac{\frac{\Gamma'', \text{isup}(p) \vdash A(p, \text{res}(tr.a))}{\vdots \xi'_2}}{\Gamma'' \vdash \text{isup}(p) \Rightarrow A(p, \text{res}(tr.a))} \Rightarrow: r \quad \dots}{\vdots \xi'_1}}{\Gamma, \forall s. \text{safety}(s) \Leftrightarrow \forall p. (\text{isup}(p) \Rightarrow A(p, s)) \vdash \text{ad}(tr.a) \Rightarrow \text{safety}(\text{res}(tr.a))}$$

Because the proof is by induction, a similar formula also occurs in the antecedent of the sequent as the induction hypothesis. There, instead of adding $\text{isup}(p)$ to the assumptions, the proof is split by a $\Rightarrow: l$ rule and a new open goal is produced. Note that the new open goal for the case of each action is mentioned in [15, p. 483] as being there ‘because of the additional condition in the guard which was added’ to the actions. Thus, the new open goal produced by the applications of the *wrap connective* transformation corresponds perfectly to the new proof that the authors of the case study had to construct after they had transformed the system in one big step and reproduced the proofs manually, cf. in particular their Fig. 6 with an overview of their new proof tree.

Wrap Quantifier. A specialized basic transformation for wrapping formulae into quantifiers, *wrap quantifier*, is similar to *wrap connective*. When formula $A(a)$ (where a is a constant) is replaced by, e.g., $\forall x. A(x)$, then again, the part ξ_1 of the proof in which the form of A does not matter is transformed by replacing the formula occurrence. Where $A(a)$ was used in the old tree, a proof step is introduced that eliminates the quantifier. If this is on the left hand side, a witness for the variable x can be chosen, and with the choice of a the rest of the proof using $A(a)$ can be reused essentially unchanged. If the \forall -quantifier is eliminated on the right hand side, the resulting formula is $A(a')$, where a' is a new Eigenvariable. In this case the old proof can be reused if it is transformed as if by the general case of *replace occurrence*, where $A(a)$ is replaced by $A(a')$, and a new open goal will be generated that reads $\Gamma, A(a') \vdash \Delta, A(a)$.

4.5 Changing Induction Schemes

Add constructor adds a constructor to type τ by making a non-constructor function $f : \tau_1 \times \dots \times \tau_n \rightarrow \tau$ a constructor function. Consequently, it changes the induction scheme for τ , but does not change the proof obligations that are generated. If inside a proof the induction rule for τ is used, the proof has to be changed to respect the new induction schema. Applying induction over τ splits the proof in m branches, where m is the number of constructors and the i th branch is the proof for the case of the i th constructor c_i . This means that the transformation adds a new child to the goal at which the induction was applied, and the child goal is the obligation to be shown for the new constructor f .

$$\frac{\dots \quad \Gamma \vdash \Delta, \dots \Rightarrow A(c_i(\dots)) \quad \dots \quad \Gamma \vdash \Delta, \dots \Rightarrow A(f(\dots))}{\Gamma \vdash \Delta, \forall x : \tau. A(x)}$$

Thus, the application of the induction rule is again a sound proof step. The new goal is left open. Since the old constructors are not changed, the subproofs above them are still valid (unless, of course, another induction over τ takes place there, in which case the transformation is applied recursively). The transformation, therefore, manages to keep all of the old proofs but produces one new open goal for each induction step over τ . Similarly, *remove constructor* removes a constructor and the corresponding cases from induction steps. In the example, a new action is added by changing the definition of type *Action* from

$$Action = B_1 : PID \times Msg \rightarrow Action \mid \dots \mid B_5 : PID \rightarrow Action$$

to

$$Action = B_1 : PID \times Msg \rightarrow Action \mid \dots \mid B_5 : PID \rightarrow Action \\ \mid Crash : PID \rightarrow Action .$$

The subproofs in the proof for (1) of the actions B_1 to B_5 are kept and a new case for the action *Crash* is added as an open goal. This corresponds to the fact that the subproof for *Crash* had to be constructed from scratch in the case study.

The transformation *add argument*, if applied to a constructor function c , also changes the induction schema for the target type of c . To ease the presentation of the transformation, consider the special case where $c : \tau_1 \rightarrow \tau$ is the only constructor for $\tau \neq \tau_1$. An application of the induction rule for τ before and after adding an argument of type $\tau_2 \neq \tau$ to c produces a proof step of the form

$$\frac{\Gamma \vdash \Delta, \forall y : \tau_1. A(c(y))}{\Gamma \vdash \Delta, \forall x : \tau. A(x)} \quad \vdots \xi \quad \text{and} \quad \frac{\Gamma \vdash \Delta, \forall y : \tau_1, z : \tau_2. A(c(y, z))}{\Gamma \vdash \Delta, \forall x : \tau. A(x)} \quad \vdots \xi'$$

respectively. The subproof ξ can be transformed by a combination of the transformations described for *wrap quantifier* (this takes care that the quantifier

binding z is removed before A is used in the proof) and *add argument* for non-constructor functions (this takes care that formulae with terms constructed by c are again well-formed).

In the example, the function $mkproc : PID \times BMsg \times MMSet \rightarrow Proc$ constructs process terms, where the arguments represent the process ID and the local state of the process. For each of the arguments corresponding to a local variable there is an update function, e.g.

$$\forall p : Proc, b : BMsg. updatebbuf(p, b) = mkproc(procpid(p), b, procD(p)) . \quad (3)$$

In the main proof for (1), lemmata are used which state that, e.g., *updatebbuf* does not change the other slots of a process, e.g.

$$\forall p : Proc, b : BMsg. procD(p) = procD(updatebbuf(p, b)) . \quad (4)$$

These lemmata are proved using induction (i.e. a case split) over p . The arity of $mkproc$ is changed to $PID \times BMsg \times MMSet \times UpDown \rightarrow Proc$ and the new lemma reads

$$\forall p, b. updatebbuf(p, b) = mkproc(procpid(p), b, procD(p), a) .$$

The induction produces open goals of the form

$$\Gamma, updatebbuf(p, b) = mkproc(\dots, a) \vdash \Delta, updatebbuf(p, b) = mkproc(\dots, a')$$

where a' is an Eigenvariable introduced for the universally bound variable resulting from the changed induction scheme. In the example these goals can be closed by subsequent *wrap quantifier* and *change occurrence* transformations.

4.6 Completeness and Adequacy

To assess the usefulness of a set of basic transformations, natural questions to ask are whether the set is complete in the sense that every well-formed specification can be transformed into an arbitrary well-formed specification using only transformations from the set, and whether the transformations are adequate in that a reasonable portion of old proofs is conserved.

The question of completeness can be answered formally, but in a totally uninteresting way: A subset of the basic transformations given above, i.e. *add* and *remove* transformations for signature items, axioms and lemmata suffice to reduce any specification to the empty specification, and also suffice to build any specification from scratch. In the course, all proof effort would be lost, however.

Adequateness can hardly be shown formally, as a characterization of the parts of proofs that are expected to be kept for an arbitrary overall transformation, does not seem to be possible in general. We have derived the set of basic transformations described above using the experience from development case studies and projects, and have found that they provided adequate support. We take this as an indication that the approach is worthwhile even though no more formal assessment of its adequateness is possible.

We expect that using the approach in practice over time, we will need to provide additional basic transformations for cases, in which the existing set is found not to be adequate, e.g. particularly useful compositions of basic transformations that can, when considered as a whole, be given better proof support.

4.7 Supporting Basic Transformations

In order to support the basic transformations described above, we employ the paradigm of explicitly representing and storing proofs and transforming them directly in a way similar to [5, 6]. Another well-known alternative would be to transform proofs by replay, i.e. by replaying the rules or tactics that were used to construct the original proof. This can be implemented by storing the original rules or tactics in an explicit proof [13], [17], [16] or by storing the proof as its conclusion and a tactic script [4] or tactic execution sequence [19]. Since our approach relies on the fact that proofs are changed in a predictable way we have found it easier to support the transformations using the the paradigm of explicit transformations rather than replaying transformed proof scripts.

We require proofs to be fully expansive, i.e. to explicitly justify each inference step taken in the proof. We view a proof as represented by a proof pattern that abstracts from the concrete formulae and terms used in the proof, together with a substitution for the variables in the schema, such that applying the substitution on the schema produces the concrete proof for the concrete proof obligation.

Given this view on proofs, many transformations can be carried out directly by changing the substitution as described in [5], i.e. *rename signature item* or *add axiom* correspond to changing the substitution in a suitable way and the change is automatically propagated throughout the proof. Similarly, *add argument* for a non-constructor simply requires changing the instantiation such that all relevant terms get an additional argument. Other transformations are carried out by transforming the proof pattern and the substitution in parallel. In any case, correctness of the resulting proofs is checked by the underlying proof engine after the proof has been transformed. This means that basic transformations need not be part of the trusted proof engine.

5 Related Work

The idea used in this paper of restricting the editing process to a set of predefined transformations is inspired by the work in [21]: a similar idea was used for the special case of editing terminating and well-typed ML programs using transformations on the program's synthesis proof, where, however, no user-generated proofs were involved.

Program development by successive application of correctness preserving transformations to an initially given specification has been investigated extensively, e.g. in the EU project PROSPECTRA [9]. An initial requirements specification is gradually transformed using correctness preserving transformations, which check for applicability conditions. This results in a waterfall-like process,

where the given initial specification is assumed to be correct and each step assumes there to be a well-defined, pre-conceived relationship between the specification before and after the transformation. However, there is no provision for changing the initial specification once a number of transformations has already been applied to it.

Other techniques have been proposed to cope with changes in definitions or theorems. Proof scripts that have been assembled when creating the original proof can be replayed, cf. e.g. [4]. In this case, however, proof scripts are patched and changed manually, whereas our approach transforms the proofs without manual intervention.

In contrast to a heuristic replay of proofs on changed proof obligations, e.g. [13], [17], [16], our approach allows transformations to produce proofs that could not have been produced by replaying the tactics that were originally applied. This allows us to change the overall proof structure in a reliable and predictable way.

Our approach is similar to the work in [5] in that an explicit proof is transformed, not by replaying the tactic steps but by changing proofs explicitly. However, we additionally consider transformations that change existing signature items and proof rules, i.e. inductions schemes. Also, we support patches for the case in which a formula is, e.g., wrapped into another one, in which case the technique described in [5] would throw away the subproof which uses the changed formula.

Management of change in the large [10], [2] is orthogonal to our approach in the sense that it is concerned with questions as to whether whole proofs remain valid after a change has taken place without the need to change these. In fact, we envision an integration such that only proofs which are invalidated by the more static and efficient analysis on the level of development graphs and whole proofs need to be subject to the proof transformations suggested in this paper.

6 Conclusions and Future Work

We have described an approach for coping with changes to specifications and proofs which are commonplace in evolutionary formal software development. It works by explicitly transforming whole specifications and proofs step by step using predefined basic transformations. These transformations change the specification in a restricted way and use the knowledge of the exact change to extend the transformation to proofs. Transformations, if they are applicable, transform a well-formed development into another well-formed development, although new open goals in proofs might be produced. Basic transformations allow to change the signature and the set of axioms and lemmata, but also allow to change existing parts of the signature and subformulae of axioms and lemmata. They alter the proof structure in a predictable way. We have shown how the approach can be used to carry out changes of a specification and the associated proofs of a case study reported in the literature. Many other examples of changes that occur

regularly in case studies done in the VSE system which we have looked at can be handled in a similar way.

In the future, we would like to extend our approach to cope with structured specifications and more complicated mappings from specifications to proof obligations. We hope to be able to cope with the latter by macros formed from several basic transformations applied after each other. For the former we hope that changes to one part of the structured specification can be propagated throughout the whole specification in a preprocessing step to make basic transformations on proofs similar to the ones described in this paper applicable.

References

- [1] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: formal methods meet industrial needs. *Int. Journal on Software Tools for Technology Transfer*, 3(1), 2000. 442
- [2] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In *Proc. Int. Workshop Algebraic Development Techniques (WADT)*, 2000. 442, 454
- [3] Common criteria for information technology security evaluation (CC), 1999. Also ISO/IEC 15408: IT – Security techniques – Evaluation criteria for IT security. 441
- [4] P. Curzon. The importance of proof maintenance and reengineering. In *Proc. Int. Workshop on Higher Order Logic Theorem Proving and Its Applications*, 1995. 453, 454
- [5] A. Felty and D. Howe. Generalization and reuse of tactic proofs. In *Proc. Int. Conf. Logic Programming and Automated Reasoning (LPAR)*, 1994. 453, 454
- [6] A. Felty and D. Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In *Proc. Int. Conf. Automated Deduction (CADE)*, 1994. 453
- [7] M. Fitting. *First Order Logic and Automated Theorem Proving*. Springer, 1996. 445
- [8] F. C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science*, 5(10):668–692, 1999. 442
- [9] B. Hoffmann and B. Krieg-Brückner, editors. *Program Development by Specification and Transformation*. Springer, Berlin, 1993. 453
- [10] D. Hutter. Management of change in structured verification. In *Proc. Automated Software Engineering (ASE)*, 2000. 454
- [11] Common Framework Initiative. The common algebraic specification language summary, March 2001. Available from <http://www.brics.dk/Projects/CoFI>. 442
- [12] Information technology security evaluation criteria (ITSEC), 1991. 441
- [13] T. Kolbe. *Optimizing Proof Search by Machine Learning Techniques*. PhD thesis, FB Informatik, TH Darmstadt, 1997. 453, 454
- [14] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley and Teubner, Chichester, Stuttgart, 1996. 445
- [15] H. Mantel and F. C. Gärtner. A case study in the mechanical verification of fault tolerance. *JETAI*, 12:473–487, 2000. 442, 443, 450
- [16] E. Melis and A. Schairer. Similarities and reuse of proofs in formal software verification. In *Proc. European Workshop on Case Based Reasoning (EWCBR)*, 1998. 453, 454

- [17] W. Reif and K. Stenzel. Reuse of proofs in software verification. In *Foundation of Software Technology and Theoretical Computer Science*, 1993. 453, 454
- [18] J. Rushby. Security requirements specifications: How and what? In *Symposium on Requirements Engineering for Information Security (SREIS)*, 2001. 441
- [19] A. Schairer, S. Autexier, and D. Hutter. A pragmatic approach to reuse in tactical theorem proving. *Electronic Notes in Theoretical Computer Science*, 58(2), 2001. 453
- [20] G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2000. 442
- [21] J. Whittle, A. Bundy, R. Boulton, and H. Lowe. An ML editor based on proofs-as-programs. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, 1999. 453

Sharing Objects by Read-Only References

Mats Skoglund

Department of Computer and Systems Sciences (DSV)
Stockholm University/Royal Institute of Technology
{matte}@dsv.su.se

Abstract. Exporting objects by reference can be problematic since the receivers can use the received reference to perform state changing operations on the referenced object. This can lead to errors if the referenced object is a subobject of a compound object not anticipating the change. We propose an extension of a type system with a read-only construct that can be used to control access to state changing methods. We formulate and prove a read-only theorem for read-only references stating that a read-only reference cannot be used to perform operations that change its referenced object state.

1 Introduction

In object-oriented programming, data abstraction and encapsulation mechanisms contribute to hide implementation details of a package, module or class. It should be possible to change the implementation of the abstraction's representation without the need to adapt the clients to these changes as long as the abstraction's public protocol is kept unchanged and the new implementation produces the same results [2].

However, protecting the representation of a compound object with so called *visibility based* protection mechanisms [14] such as *e.g.* private variables can easily be circumvented by *e.g.* exporting a reference to a part of the representation as the return value of a method. Since usually all references have the same rights to an object, uncontrolled export of references can be risky. The receiver of a reference to a subobject of a compound object may invoke any method in the referenced object's protocol, even methods that change the state of the target object, *i.e.* the method's *self* object, and thus change the state of the compound object. This may lead to that internal invariants of the compound object get broken and may perhaps lead to a system crash [4, 17]. Also, aliases are created when objects are exported by reference and aliases are a known source of errors that can lead to programs that are hard to maintain, debug, test and reason about [13]. This is a part of the *representation exposure* problem and has been the target for intensive research and many proposals for controlling the flow of references across abstractions' boundaries have been presented, *e.g.* [1, 10, 15, 16, 17, 19]. Some of these proposals define some kind of *read-only* construct as a part of, or even as the basis for, their respective approach [15, 16, 17].

A *read-only reference* is typically defined as a reference that can only read the object to which the reference refers [5]. Read-only references make it possible to export subobjects by reference without risking the exported reference being used to modify its referenced object and thereby modifying the state of the compound object of which the subobject is a part.

Unfortunately, the hitherto proposed read-only constructs are more restrictive on ordinary programming than perhaps necessary for their purpose, as will be discussed in the related work section of this paper. In this paper we propose a mode system that defines a transitive read-only construct without some of the limitations on ordinary programming as some of the prior read-only constructs suffer from. The proposed read-only construct can be used to safely export subobjects by reference from a compound object to untrusted receivers without risking the exported reference being used to change the transitive state of the compound object. The system is an extension of a type system and works by annotations on variables, formal parameters, methods and method returns. We formulate and prove a read-only theorem stating that read-only references can only be used to read the values of the objects to which they refer.

1.1 Outline

In Section 1.2 we informally define some terms used throughout this paper. Section 2 presents our key concept and the informal semantics of the mode system. In Sections 3, 4 and 5 the mode system is formally described. In Section 6 we formulate and prove some read-only properties. Section 7 discusses the result, Section 8 contains related work and Section 9 concludes.

1.2 State and State Modification

The *local state* of an object can be defined as the values of the object's member variables [15]. However, with a compound object it is sometimes more desirable to reason about the *transitive state* and its protection. This is since protection mechanisms aiming at protecting the local state of an object, such as *e.g.* C++'s `const` construct, can easily be circumvented and thus the abstraction's representation is not protected from changes, only its interface object.

Informally we use the definition of the transitive state for an object X found in [11], *i.e.* "the set of objects consisting of the objects referred to by the self-reference, *i.e.* X itself, and the transitive closure of objects referred to by X ". We do not consider temporary, free or global objects to be included in the transitive state, only static references held in member variables.

This definition of transitive state will be refined and described more formally in Section 6.

The local state can be modified by operations updating the value of a member variable, *e.g.* `this.x := y`.¹ The transitive state of an object o can be modified by changing the local state of any object in the transitive state of o .

¹ We assume that member variables are only accessible via `this`

2 The Mode System

We describe the key concepts by a notion of *modes on references*. A reference may have either the mode *write* or *read*. The mode of a reference depends on the kind of methods invoked on it. A reference with the mode *write*, a *write reference*, is in our system similar to an ordinary reference in e.g. Java, *i.e.* any method in the referenced object's protocol may be invoked on it. Methods invoked on write references is said to execute in a *write context*.

A *read reference*, on the other hand, is restricted in terms of what kind of methods may be invoked on it. A method invoked on a read reference is said to execute in a *read context* and is not allowed to change its target object's state.

2.1 A Method Call on a Read Reference

A method whose arguments do not contain write references to the target object's transitive state that is invoked on a read reference does not change the transitive state of the target object and does not return a reference that can be used to change the target object's transitive state.

However, objects are allowed to be created and references to objects that are *not* part of the target object's transitive state may be changed or returned as write. For example, a fresh object, *i.e.* a newly created object, is allowed to be returned by a write reference. This makes it possible to *e.g.* create object factories where the factory objects themselves cannot be changed while the objects returned from them can be freely manipulated by the receivers. Modifications of global objects are also allowed in methods invoked on read references. Thus, these methods are not side-effect free using the terminology of [16].

2.2 Flexibility

In our system, a method may return a write reference to its target object's transitive state when invoked on a write reference and return a read reference when invoked on a read reference. This allows for a flexible programming model since some clients are permitted to change objects returned from a method and some are not, depending on the reference used to invoke the method.

2.3 Annotations and Informal Semantics

Our system supporting the read and write reference concepts described above works by mode annotations on methods, variables, method returns and formal parameters to methods. A method must be declared as either **read** or **write**. Local variables and method returns must be declared as either **read**, **write** or **context**. A member variable must be declared as **read** or **context**. A formal parameter to a method must be declared as either **read** or **write**.

A variable declared as read (write), a *read (write) variable*, always contains a read (write) reference. In addition to the static treatment of read and write

variables, we have the more flexible *context* variables. A variable declared as context is treated differently depending on the mode of the reference used to invoke the method using it. A context variable is treated as a write variable in a write context and as a read variable otherwise. It is thus possible to change the state of an object referred to by a context variable if the change is initiated from a write reference but not if initiated from a read reference.

A method declared to return read (write) always returns read (write) references. A method declared to return context returns a reference with the same mode as the mode of the context, *i.e.* the mode of the reference the method was invoked on.

We restrict access to methods from read references depending on the declared mode of the method. A method declared as write, a *write method*, may only be invoked on write references. Our system will reject a method declared as read if the method performs some of the following.

- Changes the target object’s local state.
- Invokes a write method on the target’s transitive state.
- Returns a write reference to the target’s transitive state.
- Passes a write reference to the target’s transitive state as an argument to a method.

These methods must instead be declared as write. All other methods are *read methods* and should be declared as read to assure that the programmer does not declare a method as write if it is not state changing. This is not essential to achieve correct static treatment of modes but it is an improvement over prior proposed read-only constructs such as *e.g.* [15] where it is possible to wrongfully declare a non state-changing method as state-changing. It is easy to fulfil this condition in any method where a field is accessible by assigning it to itself. However, this kind of circumvention would probably not be necessary since read methods may be invoked on any reference.

All reference transfers; assignments, parameter passing and method returns are controlled with respect to modes. For example, assigning a write variable from a read variable is not allowed. The rules for reference transfers can be found in the type rules in Section 4.

Note that the concepts of read and write references are only abstract entities since the mode system is statically checkable and need no run-time representations of modes.

2.4 A Brief Example

The example in Figure 1, somewhat contrived for pedagogical reasons, is written in a Java-like language extended with mode annotations.

A `ThermEvent` object is created with a `Therm` object passed as read to the constructor and stored in the `th` variable. The `ThermEvent` object itself is cast to read and assigned to the read variable `te`. Then that `ThermEvent` object is sent to all listeners by the invocation of the `sendThermEvent` method. The receivers

```

class Main{
  void main():write{
    write Therm aTherm;
    write Temp newTemp;
    ...
    read ThermEvent te := read
      new ThermEvent(read aTherm);
    listeners.sendThermEvent(te); //send reads
    ...
    aTherm.setTemp(newTemp); // valid
    te.getTherm().setTemp(newTemp); // fails
    read Temp t :=
      te.getTherm().getTemp(); // valid
    t.setValue(newTemp.getValue()); // fails
    read Value v := newTemp.getValue(); //valid
    aTherm.getTemp().setValue(v); // valid
  }
}
class Value{ ...}

class Therm{
  context Temp temp;
  void setTemp(write Temp tp):write{
    temp.setValue(tp.getValue());
  }
  context Temp getTemp():read{ return temp; }
}

class Temp{
  read Value getValue():read{ ...}
  void setValue(read Value v):write{ ... }
}

class ThermEvent{
  read Therm th;
  ThermEvent(read Therm t):write{ th := t; }
  void setTherm(read Therm t):write{
    th := t;
  }
  read Therm getTherm():read{ return th; }
}

```

Fig. 1. An event example

receives a read reference to the `ThermEvent` object and thus cannot change the `Therm` object included in the `ThermEvent` object, nor any other object contained in the transitive state of the `ThermEvent` object.

The `aTherm.setTemp(newTemp)` statement is valid since `aTherm` is declared as write and thus contains a write reference to the `Therm` object. However, in the next line, trying to change the same `Therm` object via the reference obtained from the `ThermEvent` object by `te.getTherm().setTemp(newTemp)` fails since `getTherm` returns a read reference to the `Therm` object.

The statement `read Temp t := te.getTherm().getTemp()` is valid since `t` is declared as read and `getTemp` returns a read reference since it is declared to return context but is invoked on the read reference returned from the invocation of `getTherm`. Then, `t.setValue(newTemp.getValue())` fails since the `setValue` method is a write method invoked on a read reference.

The statement `aTherm.getTemp().setValue(v)` is valid since `aTherm` is write, `getTemp` is a method declared to return context and thus returns a write reference when invoked on `aTherm` and thus the write method invocation of `setValue(v)` is allowed.

3 Formal Description of the Mode System

We present our mode system in the context of a Java-like language which is an extension of a subset of ClassicJava [8].

The abstract syntax for our Java-like language is shown in Fig. 2. To simplify the presentation we include only those elements relevant for the presentation of the mode system. Methods are public and fields (member variables) are private and can only be accessible via `this`, similar to [3]. All expressions evaluate to values. A value is either a reference (object identifier) or null. The value of the

```

P =      defn* e
defn =   class c { field* meth* }
meth =  m c md(arg):(read|write){local* e}
field = (read|context) c fd
arg =   (read|write) c lv
e =     e; e | read e | new c | var | this | null | e.md(e) | var := e
local = m c lv
lv =    a local variable name or a parameter name
var =   lv | this.fd
c =     a class name
fd =    a field name
md =    a method name
m =     read | write | context

```

Fig. 2. Abstract syntax for the Java-like language

last expression in a method is the value returned from the method. We allow only one single parameter to a method and the parameter is treated as a local variable. We also treat **this** as a local variable with the restriction that it cannot be assigned to and that it is always considered to be declared as context. We assume that all member variables' names are distinct and that formal parameter names do not clash with the names of local variables declared in the method body. We do not have any constructors and fields of newly created objects are initialised to null.

Also, for simplicity of the presentation, the language does not include subtyping or inheritance. The only relevant difference regarding modes when considering subtyping and inheritance similar to Java's [9] is that the dynamic binding of methods requires a strategy for the modes of overridden methods. A simple strategy is to require every overriding method to have the same mode as the overridden; both for the method itself and for the mode of the method return. An extension of the mode system including subtyping and inheritance has been presented elsewhere [18].

For types and modes we have a notion of type/mode pairs, written (t, m) , where t is a type and m denotes a mode. We have an environment Γ for local variables and parameters to methods: $\Gamma ::= \epsilon | \Gamma, lv : (t, m)$ where (t, m) is the type/mode pair for a variable lv , where t is the declared type of lv , *i.e.* its class, and m its declared mode. We write $\Gamma\{lv : (t, m)\}$ to denote that Γ has been extended with the variable lv with the type/mode pair (t, m) .

We use overline notation to denote lists, *e.g.* we write $\Gamma\{\overline{lv} : (\overline{t}, \overline{m})\}$ to denote that Γ has been extended with multiple variables with their respective type/mode pairs. Note that \overline{m} is different from m . Also, we have dictionaries for local variables and methods for every class, similar to [7]. These dictionaries can be determined from the program.

$$\mathcal{M}^c \stackrel{def}{=} \{fd \mapsto (t, m)\}^*$$

A map from field names to type/mode pairs.

$$\mathcal{MD}^c \stackrel{def}{=} \{md \mapsto ((t_{arg}, m_{arg}) \rightarrow (t_{ret}, m_{ret}), a, e, \{lv \mapsto (t, m)\}^*, mm)\}^*$$

A map from method names to a 5-tuple consisting of the method's type, *i.e.* its parameter type/mode pair together with its return type/mode pair, the argument name, the method body and the mappings for the local variables to their type/mode pairs and the declared mode of the method.

4 The Type System

The judgements for the type system are shown in Figure 3.

In addition to the standard P and Γ , the judgements for conversion (denoted with \rightsquigarrow) and expressions use a special context variable μ denoting the context an expression is evaluated in. In a write method the context is always write while the context may be either write or read in a read method.

The type rules are shown in Figure 4 and are pretty straightforward. A program is valid (*program*) if all class definitions in the program $defn_1 \dots defn_n$ are valid with respect to the program P and if the expression following the class definitions e can be assigned a type/mode pair with an empty environment and in an initial write context.

A class is valid (*class*) when all its fields' type/mode pairs are valid and when all methods in the class, $meth_1 \dots meth_n$, are valid given an environment containing only **this** declared as context.

A valid type/mode pair (*type/mode*) is a pair where the type is defined in the program P and the mode is either of *read*, *write*, or *context*. We write $P = P', def, P''$ to denote that P contains def , similar to [6].

The rule (*conv*) describes valid conversion between type/mode pairs and is used for reference transfers in assignments, method calls and method returns.

A method is valid (*meth*) if its expression e can be assigned a type/mode pair with an environment containing **this** (from the rule (*class*)), the parameter to the method (**a**) and the local variables declared in the method body (\overline{lv}) and with the context variable μ assigned the value of the declared mode of the method. Also, if the expression can be assigned a type/mode pair in a read context then the method must be declared read. This is since we want all methods to be correctly declared by the programmer. That a state changing method is required to be declared write is handled in the rules (*fd set*) and (*call*).

Mode casting (*read*) sets the resulting mode to read and is only included for clarity. A field can only be accessed via **this** and the resulting type/mode pair is the type/mode declared for the field in the field dictionary.

When assigning to a field, (*fd set*), the mode of the context must be write. The resulting mode of object creation (*new*) is always write.

To invoke a method (*call*) the method must be present in the method dictionary of the target objects class. It is only allowed to invoke a write method if the mode of the target is write or context in a write context. A read method may be invoked regardless of the mode of the target.

$\vdash P : (t, m)$	P is well-formed with type t and mode m
$P \vdash defn$	$defn$ is well-formed
$P, \Gamma \vdash meth$	$meth$ is well-formed
$P, \mu \vdash (t, m) \rightsquigarrow (t', m')$	The conversion from (t, m) to (t', m') is well-formed
$P, \Gamma, \mu \vdash e : (t, m)$	e is well-formed with type t and mode m
$P \vdash (t, m)$	(t, m) is well-formed

Fig. 3. The judgements for the mode system

$$\begin{array}{c}
\text{(program)} \frac{P \vdash \text{def } n_i \text{ for } i \in \{1 \dots n\} \quad P, \emptyset, \mathbf{write} \vdash e : (t, m)}{\vdash P : (t, m)} \\
\text{Where } P = \text{def } n_1 \dots \text{def } n_n e \\
\\
\text{(class)} \frac{P \vdash (t_i, m_i) \text{ for } i \in \{1 \dots n\} \quad P, \{\mathbf{this} : (c, \mathbf{context})\} \vdash \text{meth}_j \text{ for } j \in \{1 \dots k\}}{P \vdash \mathbf{class } c\{m_1 t_1 fd_1 \dots m_n t_n fd_n \text{meth}_1 \dots \text{meth}_k\}} \\
\\
\text{(type/mode)} \frac{P = P', \mathbf{class } c\{\dots\}, P'' \quad m \in \{\mathbf{read}, \mathbf{write}, \mathbf{context}\}}{P \vdash (c, m)} \\
\\
\text{(conv)} \frac{P \vdash (t, m) \quad P \vdash (t', m') \quad t = t' \quad ((m = m') \text{ or } (m = \mathbf{context} \text{ and } m' = \mu) \text{ or } (m = \mathbf{write} \text{ and } \mu = \mathbf{write} \text{ and } m' = \mathbf{context}))}{P, \mu \vdash (t, m) \mapsto (t', m')} \\
\\
\text{(meth)} \frac{P \vdash (t, m) \quad P \vdash (t'', m'') \quad P, \Gamma\{a : (t'', m''), \bar{lv} : (\bar{t}, \bar{m})\}, \mu \vdash e : (t', m') \quad \mu = mm \quad P, \Gamma\{a : (t'', m''), \bar{lv} : (\bar{t}, \bar{m})\}, \mathbf{read} \vdash e : (t', m') \Rightarrow \mu = \mathbf{read} \quad \mu \in \{\mathbf{read}, \mathbf{write}\} \quad P, \mu \vdash (t', m') \mapsto (t, m) \quad P \vdash (m_i, t_i) \text{ for } i \in \{1 \dots n\}}{P, \Gamma \vdash m t \text{ md}(m'' t'' a) : mm\{m_1 t_1 lv_1 \dots m_n t_n lv_n e\}} \\
\\
\text{(seq)} \frac{P, \Gamma, \mu \vdash e : (t, m) \quad P, \Gamma, \mu \vdash e' : (t', m')}{P, \Gamma, \mu \vdash e; e' : (t', m')} \quad \text{(read)} \frac{P, \Gamma, \mu \vdash e : (t, m)}{P, \Gamma, \mu \vdash \mathbf{read } e : (t, \mathbf{read})} \\
\\
\text{(lv)} \frac{lv \in \text{dom}(\Gamma) \setminus \mathbf{this} \quad (t, m) = \Gamma(lv)}{P, \Gamma, \mu \vdash lv : (t, m)} \quad \text{(this)} \frac{\mathbf{this} \in \text{dom}(\Gamma) \quad (t, m) = \Gamma(\mathbf{this}) \quad m = \mathbf{context}}{P, \Gamma, \mu \vdash \mathbf{this} : (t, m)} \\
\\
\text{(lv set)} \frac{P, \Gamma, \mu \vdash lv : (t', m') \quad lv \in \text{dom}(\Gamma) \setminus \mathbf{this} \quad ((P, \mu \vdash (t, m) \mapsto (t', m')) \text{ or } (m = \mathbf{write} \text{ and } \mu = \mathbf{write} \text{ and } m' = \mathbf{context} \text{ and } t = t'))}{P, \Gamma, \mu \vdash lv := e : (t, m)} \\
\\
\text{(fd)} \frac{fd \in \text{dom}(\mathcal{M}^t) \quad (t', m') = \mathcal{M}^t(fd) \quad P, \Gamma, \mu \vdash \mathbf{this} : (t, m)}{P, \Gamma, \mu \vdash \mathbf{this}.fd : (t', m')} \quad \text{(new)} \frac{P \vdash (t, m)}{P, \Gamma, \mu \vdash \mathbf{new } t : (t, \mathbf{write})} \\
\\
\text{(fd set)} \frac{P, \Gamma, \mu \vdash \mathbf{this}.fd : (t, m) \quad P, \Gamma, \mu \vdash e : (t', m') \quad \mu = \mathbf{write} \quad P, \mu \vdash (t', m') \mapsto (t, m)}{P, \Gamma, \mu \vdash \mathbf{this}.fd := e : (t', m')} \quad \text{(null)} \frac{P \vdash (t, m)}{P, \Gamma, \mu \vdash \mathbf{null} : (t, m)} \\
\\
\text{(call)} \frac{P, \Gamma, \mu \vdash e : (t, m) \quad P, \Gamma, \mu \vdash e' : (t', m') \quad m_{ret} \neq \mathbf{context} \Rightarrow m'' = m_{ret} \quad m_{ret} = \mathbf{context} \Rightarrow m'' = m \quad (m = \mathbf{context} \text{ and } \mu = \mathbf{write}) \text{ or } m = \mathbf{write} \text{ or } mm = \mathbf{read}}{P, \mu \vdash (t', m') \mapsto (t_{arg}, m_{arg}) \quad \mathcal{MD}^t(md) = \langle (t_{arg}, m_{arg}) \rightarrow (t_{ret}, m_{ret}), \Rightarrow, \Leftarrow, \Leftarrow, mm \rangle} \\
P, \Gamma, \mu \vdash e.md(e') : (t_{ret}, m'')
\end{array}$$

Fig. 4. The type system

If the declared return mode of the method is read or write then the resulting mode of a method call is the declared return mode. If the method is declared to return context the resulting mode is the same mode as the mode of the target.

5 Dynamic Semantics

The dynamic semantics is described with a big-step operational semantics. An evaluation is written $\langle e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}'$, where e is the expression being eval-

uated, Δ and Δ' are stacks, \mathcal{S} and \mathcal{S}' are stores and v is the resulting value that is either a reference or **null**.

Definition 1 (Store). A store \mathcal{S} is a mapping from references (object identifiers), written as r or o to objects. An object is a map from field names to values ($v ::= r|\text{null}$) and is denoted with \mathcal{F} :

- $\mathcal{S} \stackrel{\text{def}}{=} \{r \mapsto \mathcal{F}\}^*$ where \mathcal{F} is a mapping from field names to values:
- $\mathcal{F} \stackrel{\text{def}}{=} \{fd \mapsto v\}^*$ and $\text{dom}(\mathcal{F}) = \text{dom}(\mathcal{M}^c)$ where c is the class of r .

We have the following operations on stores:

- $\mathcal{S} \cup \{r \mapsto \mathcal{F}\}$ is the store obtained when a new reference r with a field map \mathcal{F} , i.e. $r \mapsto \mathcal{F}$, is added to \mathcal{S} where $r \notin \text{dom}(\mathcal{S})$.
- $\mathcal{S}[r \mapsto \mathcal{F}]$ is the result of updating the reference r in the store \mathcal{S} with the field map \mathcal{F} .
- $\mathcal{F}[fd \mapsto v]$ is the field map obtained when the field fd is updated with the value v in the field map \mathcal{F} .

Definition 2 (Stack). A stack frame δ is a mapping from local variable names and parameter names to their respective values: $\delta \stackrel{\text{def}}{=} \{lv \mapsto v\}^*$. We have the following operations on stack frames:

- $\delta(lv) \stackrel{\text{def}}{=} v$, where $lv \mapsto v \in \delta$
- $\delta[lv \mapsto v]$ is the stack frame where the variable lv has been updated with the value v

A stack Δ is a sequence of stack frames $\delta_1 \dots \delta_n$ with δ_n at the top. We have the following operations on stacks:

- $\Delta; \delta$ is the stack formed by pushing δ onto Δ
- $\Delta(lv) \stackrel{\text{def}}{=} \delta(lv)$ A lookup in the stack is defined as a lookup in the top stack frame where $\Delta \equiv \Delta'; \delta$

The reduction rules are shown in Figure 5. The rules are standard and show that modes are not involved in evaluation of the program and thus do not impose any run-time or memory overhead.

The rule for method call (*call*) requires some explanation. First evaluates the target and then the argument. Then the method body is evaluated with a new stack frame pushed on the stack consisting of **this** mapped to the target object, the parameter mapped to the actual argument and the local variables mapped to **null**. After the evaluation of the expression in the method body the top stack frame is discarded. Only the top stack frame is used in the evaluation of the method body, which can be shown by induction [7]. The result of the expression in the method body is the result of the method call. Trying to invoke a method on **null** yields an error (*ncall*).

$$\begin{array}{c}
\text{(seq)} \frac{\langle e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}' \quad \langle e', \Delta', \mathcal{S}' \rangle \rightarrow v', \Delta'', \mathcal{S}''}{\langle e; e', \Delta, \mathcal{S} \rangle \rightarrow v', \Delta'', \mathcal{S}''} \\
\text{(read)} \frac{\langle e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}'}{\langle \text{read } e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}'} \quad \text{(lv)} \frac{v = \Delta(lv)}{\langle lv, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta, \mathcal{S}} \\
\text{(this)} \frac{v = \Delta(\mathbf{this}) \quad v \neq \text{null}}{\langle \mathbf{this}, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta, \mathcal{S}} \\
\text{(fd)} \frac{r = \Delta(\mathbf{this}) \quad v = \mathcal{S}(r)(fd)}{\langle \mathbf{this}.fd, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta, \mathcal{S}} \quad \text{(lv set)} \frac{\langle e, \Delta; \delta, \mathcal{S} \rangle \rightarrow v, \Delta; \delta'', \mathcal{S}' \quad \delta' = \delta''[lv \mapsto v]}{\langle lv := e, \Delta; \delta, \mathcal{S} \rangle \rightarrow v, \Delta; \delta', \mathcal{S}'} \\
\text{(fd set)} \frac{\langle \mathbf{this}, \Delta, \mathcal{S} \rangle \rightarrow r, \Delta, \mathcal{S} \quad \langle e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}'}{\langle \mathbf{this}.fd := e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}'[r \mapsto \mathcal{F}[fd \mapsto v]]} \text{ Where } \mathcal{S}'(r) = \mathcal{F} \\
\text{(new)} \frac{r \notin \text{dom}(\mathcal{S})}{\langle \mathbf{new } t, \Delta, \mathcal{S} \rangle \rightarrow r, \Delta, \mathcal{S} \cup \{r \mapsto \{fd \mapsto \text{null} \mid fd \in \text{dom}(\mathcal{M}^t)\}} \\
\text{(call)} \frac{\langle e, \Delta, \mathcal{S}_0 \rangle \rightarrow v', \Delta', \mathcal{S}_1 \quad v' \neq \text{null} \quad \langle e', \Delta', \mathcal{S}_1 \rangle \rightarrow v'', \Delta'', \mathcal{S}_2}{\langle e'', \Delta''; \{\mathbf{this} \mapsto v'\} \cup \{a \mapsto v''\} \cup \{\overline{lv} \mapsto \text{null}\}, \mathcal{S}_2 \rangle \rightarrow v, \Delta''; \delta, \mathcal{S}_3} \\
\text{Where } v' \text{ has type } c, \mathcal{MD}^c(md) = \langle -, a, e'', l, - \rangle, \\
\overline{lv} = \text{dom}(l) \text{ and } \text{dom}(\delta) = \{\mathbf{this}\} \cup a \cup \overline{lv}. \\
\text{(ncall)} \frac{\langle e, \Delta, \mathcal{S} \rangle \rightarrow v, \Delta', \mathcal{S}' \quad v = \text{null}}{\langle e.md(e'), \Delta, \mathcal{S} \rangle \rightarrow \text{Error}} \quad \text{(null)} \frac{}{\langle \mathbf{null}, \Delta, \mathcal{S} \rangle \rightarrow \text{null}, \Delta, \mathcal{S}}
\end{array}$$

Fig. 5. The dynamic semantics

6 Read-Only Properties

To formalise the read-only properties we first define some terms and help predicates.

Definition 3 (Read and Write References).

A read (write) reference is the result of an expression returning a value where the mode of the expression is:

- read (write) in a read or write context or
- context in a read (write) context.

An object's transitive state is intuitively a graph where objects are nodes, fields are edges and the field names are labels on the edges.

Definition 4 (Transitive State).

Given an object o in the store, i.e. $o \in \text{dom}(\mathcal{S})$ the transitive state of o with

respect to the store \mathcal{S} , denoted as $\mathcal{TS}_o^{\mathcal{S}}$, is the smallest graph satisfying the following:

1. $o \in \mathcal{TS}_o^{\mathcal{S}}$.
2. if $v \neq \text{null}$ and $v \in \mathcal{TS}_o^{\mathcal{S}}$, with $\mathcal{F} = \mathcal{S}(v)$ then for each $fd \mapsto v' \in \mathcal{F}$, $v' \in \mathcal{TS}_o^{\mathcal{S}}$ and $v \xrightarrow{fd} v' \in \mathcal{TS}_o^{\mathcal{S}}$.

The transitive state of an object o with respect to a store \mathcal{S} , i.e. $\mathcal{TS}_o^{\mathcal{S}}$, is equal to the transitive state for the same object with respect to another store \mathcal{S}' if the set of references are equal in both graphs and if every field contains the same values in both state graphs.

Definition 5 (Equal Transitive States).

$\mathcal{EQ}(o, \mathcal{S}, \mathcal{S}')$ is true iff $\forall v \in \mathcal{TS}_o^{\mathcal{S}}, v \in \mathcal{TS}_o^{\mathcal{S}'}$ and $\forall r \xrightarrow{fd} v' \in \mathcal{TS}_o^{\mathcal{S}}, r \xrightarrow{fd} v' \in \mathcal{TS}_o^{\mathcal{S}'}$.

Note that \mathcal{EQ} is transitive with respect to stores, i.e. if $\mathcal{EQ}(o, \mathcal{S}, \mathcal{S}')$ and $\mathcal{EQ}(o, \mathcal{S}', \mathcal{S}'')$ then $\mathcal{EQ}(o, \mathcal{S}, \mathcal{S}'')$.

The transitive state of an object o in a store \mathcal{S} , i.e. $\mathcal{TS}_o^{\mathcal{S}}$, is changed when a field in an object in the transitive state of o is updated. To change the transitive state of the object o we must update a field in an object r where $r \in \mathcal{TS}_o^{\mathcal{S}}$.

When extending the store with a new object, the transitive states of all original objects in the store are unchanged.

Since write references to objects are allowed to be used to change their referenced object's transitive states we define a starting configuration without any write references to a specified object, the "protected" object. We then prove our read-only properties in such a starting configuration without risking interference of any original write references to our "protected" object.

We formalise this starting configuration by first introducing *safe paths*. There are only safe paths from one object to the transitive state of another object, the protected object, if every path from the first object to objects in the transitive state of the protected object goes through a field declared as read. An example of an object having safe paths to the transitive state of another object is shown in Fig. 6.

Definition 6 (Safe Paths).

The predicate $\mathcal{SP}(o, o', \mathcal{S})$ is true if $\forall r \in \text{dom}(\mathcal{S})$ s.t. $r \neq o, r \in \mathcal{TS}_o^{\mathcal{S}}$ and $r \in \mathcal{TS}_{o'}^{\mathcal{S}}$ there is at least one $r' \xrightarrow{fd} r'' \in \mathcal{TS}_o^{\mathcal{S}}$ s.t. $\mathcal{M}^t(fd) = (_, \text{read})$ in every path from o to r , where t is the class of r' .

We use safe paths to define *safe values*. A value is a safe value with respect to an object o if the value is null or the value is a read reference or there are only safe paths from the value to o 's transitive state and the value itself is not in the o 's transitive state.

Definition 7 (Safe Value).

The predicate $\mathcal{SV}(v, o, \mathcal{S}, m, \mu)$ is true if some of the following holds:

1. $v = \text{null}$.

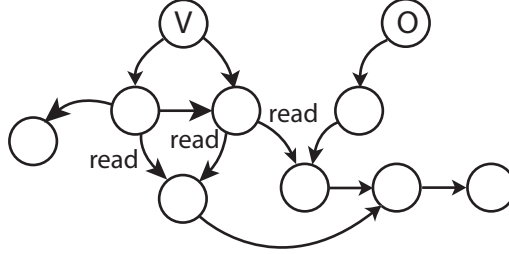


Fig. 6. All paths from v to values included in the transitive states of both v and o goes through at least one field declared as read

2. $m = \mathbf{read}$ or ($m = \mathbf{context}$ and $\mu = \mathbf{read}$).
3. $v \neq \mathbf{null}$ and $\mathcal{SP}(v, o, \mathcal{S})$ and $v \notin \mathcal{TS}_o^{\mathcal{S}}$.

A stack is safe if all variables in the stack contain safe values with respect to the protected object.

Definition 8 (Safe Stack).

The predicate $\mathcal{SS}(o, \delta, \mathcal{S}, \Gamma, \mu)$ is true if $\forall lv \in \text{dom}(\delta)$, $\mathcal{SV}(\delta(lv), o, \mathcal{S}, m, \mu)$ for $\Gamma(lv) = (_, m)$.

Our starting configuration where no write references to a protected object are present is simply a configuration with a safe stack with respect to the protected object. For example, at the beginning of a method invoked on a read reference where the parameter to the method is declared as read we have a configuration with a safe stack with respect to the target object.

With this starting configuration we formulate a read-only invariant for references. This invariant states that if we evaluate an expression with a safe stack with respect to an object and a store, then the transitive state of the object with respect to that store is the same after the evaluation of an expression as it was before. The resulting stack after the evaluation is also a safe stack and the result of the evaluation is a safe value with respect to o . Furthermore, the evaluation does not create any new unsafe paths from any object in the store to the transitive state of the protected object.

Theorem 1 (Generalised Read-Only Theorem).

if $\langle e, \Delta; \delta, \mathcal{S} \rangle \rightarrow v, \Delta; \delta', \mathcal{S}'$ and $P, \Gamma, \mu \vdash e : (t, m)$ and given an object $o \in \text{dom}(\mathcal{S})$ such that $\mathcal{SS}(o, \delta, \mathcal{S}, \Gamma, \mu)$, then:

1. $\mathcal{EQ}(o, \mathcal{S}, \mathcal{S}')$.
2. $\mathcal{SS}(o, \delta', \mathcal{S}', \Gamma, \mu)$.
3. $\mathcal{SV}(v, o, \mathcal{S}', m, \mu)$.
4. $\forall r \in \text{dom}(\mathcal{S})$ s.t. $\mathcal{SP}(r, o, \mathcal{S})$, $\mathcal{SP}(r, o, \mathcal{S}')$.

Proof. The proof is by structural induction on $\langle e, \Delta; \delta, \mathcal{S} \rangle \rightarrow v, \Delta; \delta', \mathcal{S}'$. □

7 Discussion

We have shown a statically checkable mode system where read references can be used to protect the transitive states of objects from modifications. We have a notion of *read methods* that is methods which, when invoked on read references, do not change the transitive states of their target objects unless write references to the target objects' transitive states can be obtained from the arguments to the methods. However, in a read method we allow creation of new objects and modification of objects that are not part of the target object's transitive state, *e.g.* global objects. This allows for a more flexible programming model than some earlier proposed read-only constructs, *e.g.* functional methods in [16] or clean expression in [17]. This is more thoroughly discussed in Section 8.

A read method cannot change the transitive state of its *self* object when invoked on a read reference unless a write reference to a part of the state of the target object can be obtained from the parameter to the method. Then it is possible to invoke write methods on that reference that perhaps change the state of the target object. This flexibility could be viewed as both a benefit and as a drawback. It is a benefit since allowing write references as incoming parameters makes it possible to *e.g.* perform in-place updates. Since the write references were already present and reachable in the method performing the call the target object was already exposed for modification from that object and thus it is not necessarily a problem allowing these changes within the method. This feature can, however, be viewed as a drawback if the programmer accidentally passes a writable alias to the target object's state as an argument and is not aware of the possible consequences. By requiring all parameters to read methods be declared as read this drawback would easily be eliminated. However, some flexibility of the read methods would then also be lost.

The syntactical overhead imposed by the mode system is high since the programmer must annotate every method, formal parameter, variable and method return with a mode. This burden can be reduced by introducing default modes where the annotations are left out. For example, it should be possible to write programs without any protection using only write and context annotations. Thus, all local variables, formal parameters, methods and method returns could be silently treated as write and all fields could be treated as context unless annotated with read. It would then be possible to write programs annotating only those parts of the program where the read-only protection is desired.

The syntactical overhead could also be reduced further by replacing the casting construct with implicit conversion from write to read references in cases where the correct treatment of modes can be statically ensured.

8 Related Work

In C++ it is possible to declare a variable to be a pointer to a *const* object. The referenced object is thus protected from changes to its *local state* from that variable. It is, however, still possible to change the object's transitive state via

the same variable. It is, for example, possible to invoke a method on the variable that returns a non-const reference to an object referred to by the protected object. Methods that change their target objects and thus the transitive state of the const protected object can then be invoked on the returned reference. Another drawback with const is that it can be cast away. This makes it possible to remove the protection and invoke methods that change the state of an object. Other ways to circumvent const protection are described in *e.g.* [20].

In [12] a read construction is proposed for achieving side-effect free functions. A variable annotated with the *read* label cannot be assigned to and can only be exported as read. Also, a read expression cannot be assigned from. As pointed out in [17] these restrictions are limiting on ordinary programming with object-oriented programming languages.

Addressing encapsulation issues, [17] proposes Flexible Alias Protection that make use of read-only constructs named *clean expressions* and *clean messages* respectively. The definition of a clean message is that it is made up only of clean expressions. A clean expression may either read values of objects or send clean messages. Unfortunately this has some practical restrictions as even pointed out in [17]. For example, it is impossible to change *any* object within a clean expression, not even objects *not* in the transitive state of the enclosing object.

Similar constructs, named *read-only references* and *functional methods* respectively, have been presented for Universes in [16], also for encapsulation purposes. It is only allowed to invoke functional methods on read-only references since they do not cause side-effects. This means, however, that no modification of *any* object is allowed and creation of new objects is also not permitted in a functional method.

Interfaces in Java can be used to hide those methods that are state changing. Unfortunately, the protection offered by using interface techniques in *e.g.* Java can be easily circumvented since Java's interfaces can be downcasted to the real type of the object and then any state changing methods may be invoked.

A similar technique using the type system of Java is presented in [15] to provide a read-only construct for the Java language. In this system every programmer supplied type has an implicit *read-only* supertype. This supertype consists only of those methods labelled by the programmer as not state changing. Every variable declared as read-only is for the analysis replaced with its corresponding read-only supertype. Since the proposal relies on Java's type system the use of protected member variables and methods is restricted as pointed out by [16]. Also it creates a dependency on inheritance and on a strong type system. Another drawback with this approach is that the programmer is required to manually annotate the methods correctly with respect to their read-only properties or otherwise methods that are read-only cannot be used as such. Annotating the methods manually could be a demanding task since aliases must be traced and since the dynamic binding of methods must be considered.

9 Conclusion and Future Work

Exporting references can give rise to problems since references always give full access to their referenced objects' protocol. We presented a mode system that defines a read-only construct that can be used to control access to methods depending on the mode of reference used. It is statically checkable and does not impose any runtime or memory overhead. We proved some read-only properties stating that the transitive state of an object does not change when a read method is invoked on a read reference, and that only read references to its transitive state are exported, during the evaluation of a method invoked on a read reference. By the aid of the mode system it is possible to export subobjects by reference without risking the transitive state of the exported object being changed.

One direction for future work is to reduce the risk of the programmer accidentally passing in aliases to the target object's state as arguments to read methods invoked on read references. We will also examine the effect of allowing member variables to be excluded from its object's state and thus allowing them to be changed in read methods invoked on read references.

References

- [1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP '97 Proceedings*, 1997. 457
- [2] H. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *The 29th Annual Symposium on Principles of Programming Languages, POPL.*, 2002. 457
- [3] H. Banerjee and D. A. Naumann. A static analysis for instance-based confinement in java, 2002. In Manuscript, Available from <http://guinness.cs.stevens-tech.edu/~naumann/publications.html>. 461
- [4] B. Bokowski and J. Vitek. Confined types. In *OOPSLA '99 Proceedings*, 1999. 457
- [5] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, pages 2–27, Berlin, Heidelberg, New York, 2001. Springer. 458
- [6] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997. 463
- [7] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Proceedings*, 1998. 462, 465
- [8] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, 1997, revised 6/99. 461
- [9] J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley Publishing Company, 2000. 462
- [10] H. Hakonen, V. Leppänen, T. Raita, T. Salakoski, and J. Teuhola. Improving object integrity and preventing side effects via deeply immutable references. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, FUSST*, 1999. 457

- [11] V. L. Harri Hakonen and T. Salakoski. Object integrity while allowing aliasing. In *The 16th IFIP World Computer Congress International Conference on Software: Theory and Practice, ICS'2000*, pages 91 – 96., 2002. 458
- [12] J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285. ACM Press, 1991. 470
- [13] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. In *OOPS Messenger 3(2)*, April 1992, 1992. 457
- [14] G. Kniesel. Encapsulation=visibility + accessibility (revised). Technical Report IAI-TR-96-12, Institut for Informatik III, 1998. 457
- [15] G. Kniesel and D. Theisen. Java with transitive access control. In *IWAOOS'99 – Intercontinental Workshop on Aliasing in Object-Oriented Systems. In association with the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999. 457, 458, 460, 470
- [16] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. 457, 459, 469, 470
- [17] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998. 457, 469, 470
- [18] M. Skoglund and T. Wrigstad. A mode system for readonly references. In *3rd workshop on Formal Techniques for Java Programs*, 2001. Available from: <http://www.dsv.su.se/~matte/publ/ecoop2001.html>. 462
- [19] M. Skoglund and T. Wrigstad. Alias control with read-only references. In *Proceedings of 6th International Conference on Computer Science and Informatics*, 2002. 457
- [20] B. Stoustrup. *The C++ Programming Language Third Edition*. Addison-Wesley Publishing Company, 1997. 470

Class-Based versus Object-Based: A Denotational Comparison

Bernhard Reus *

School of Cognitive and Computing Sciences, University of Sussex
{bernhard}@cogs.susx.ac.uk

Abstract. In object-oriented programming one distinguishes two kinds of languages. Class-based languages are centered around the concept of classes as descriptions of objects. In object-based languages the concept of a class is substituted by constructs for the creation of individual objects. Usually, the object-based languages attract interest because of their “simplicity”. This paper contains a thorough denotational analysis which reveals that simplicity is quickly lost if one tackles verification issues. This is due to what is sometimes called “recursion through the store”. By providing a denotational semantics for a simple class-based and a simple object-based language it is shown that the denotational semantics of the object-based language needs much more advanced domain theoretic machinery than the class based one. The gap becomes even wider when we define concepts of specification and appropriate verification rules.

1 Introduction and Motivation

In their seminal book *A Theory of Objects* [1, Chapter 4] Abadi and Cardelli write “*Object-based languages are intended to be both simpler and more flexible than traditional class-based languages.*” and “*Object-based languages are refreshing in their simplicity, compared with even simple class based languages.*” While object-based languages are definitely *more flexible*, they are *simpler* only with respect to syntax but much more involved with respect to semantics and programming logics. To demonstrate this is the main objective of this paper.

Semantics in object-oriented languages usually refers to some operational semantics. Also [1] mainly deal with syntax, operational semantics and typing. Denotational semantics does not get much attention although e.g. *loc.cit.* [Chap. 14] contains a sophisticated typed denotational semantics for a functional object calculus. Consequently, programming logics (axiomatic semantics) for object-oriented languages are usually treated in a syntactic style, proving soundness of a logic w.r.t. the operational semantics of a language [2, 4, 8, 15, 20]. Some notable exceptions are the extension of “higher-order Algol” to objects [16, 11] and the co-algebraic approach propagated by [10]. The former uses a different

* partially supported by the EPSRC under grant GR/R65190/01 and by the Nuffield Foundation under grant NAL/00244/A

style which makes comparison with object-based languages difficult and the latter does not give rise to any particular proof principles apart from co-induction which does not seem appropriate for the object-based case (see [19]).

Denotational techniques were quite successfully applied to the verification of functional programs in the LCF project (Logic of Computable Functions [13]). It is more abstract and allows for concise soundness proofs. Therefore, we use denotational semantics for a rigorous analysis of class-based and object-based languages.

Class-based languages use the *class* concept to describe a collection of objects that are the *instances* of the class. Object-based languages get around the notion of *class* by providing features for the creation of individual objects. Classes separate methods from objects but the class types still establish a link. They indicate to which collection an object belongs and which method suite applies. In the object-based case any object provides its own method suite. Method invocation is therefore always with respect to an object's methods the code of which is stored in heap. Recursive methods can be simply obtained by calling the same method of the same object. Object specifications must account for this kind of recursion and necessarily become recursive themselves. Therefore, existence of specifications cannot be taken for granted. We will use the techniques presented in [14] to obtain an existence result.

As representatives for object-based and class-based languages we chose the imperative untyped object-calculus of [1] and a very rudimentary sublanguage of Java, respectively. The syntax of both languages is about equally rich. Throughout the paper we just deal with an untyped semantics¹. As types are properties of syntax they can be expressed on the specification level.

The results of the object-based languages appear in greater detail in [19], the class-based analysis and the comparison are the original contributions. Some preliminary attempts have been published in [18, 17].

In [5] a different but related comparison has been carried out for various interpretations of the functional object calculus in type theory. It stops short of classes and does not discuss the imperative paradigm nor verification issues.

The paper is organised as follows: In Sect. 2 we introduce the syntax of both languages, the class-based and the object-based one. Their denotational semantics is given in the following section. Programming logics on the level of denotations are then discussed, again for both languages, in Section 4 by providing a notion of specification and discussing verification rules. A comparison in a nutshell (Sect. 5) and some remarks about further extensions (Sect. 6) conclude the paper.

2 Syntax

In this section the syntax of the class-based and object-based sample languages is introduced. We restrict our attention to small kernel languages.

¹ This does not mean that we ignore types completely. In the class-based case the type of an object is essential for method dispatch.

2.1 Preliminaries

The name space is divided into class names \mathcal{C} , field names \mathcal{F} , and method names \mathcal{M} . Moreover, Var denotes a infinitely countable set of untyped variables.

Note that in the rest of the paper $x \in \text{Var}$ will always denote a variable and all (possibly qualified) appearances of m will refer to labels in \mathcal{M} , all (possibly qualified) appearances of f will refer to labels in \mathcal{F} and all (possibly qualified) appearances of C will refer to labels in \mathcal{C} .

2.2 Class-Based

The syntax of our simple class-based language is defined below. For the sake of simplicity, we omit method parameters, access modes, static attributes and static methods.

$a, b ::= x$		<code>new C()</code>	variables
		<code>a.f</code>	object creation
		<code>a.f = b</code>	field selection
		<code>a.m()</code>	field update
		<code>let x=a in b</code>	method call
			local def.

The “self” reference `this` is just considered a special (predefined) variable name, therefore `this` $\in \text{Var}$. Note that we do not distinguish between expressions and statements (like in the object calculus). There is no extra operator for sequential composition as it can be expressed using “let”, i.e. $a; b \equiv \text{let } x = a \text{ in } b$ where x does not occur freely in a and b . Methods are considered non-void but this is not a real restriction.

Additionally, there is a syntax for class definitions:

$$\begin{aligned}
 c ::= & \text{class } C \{ \begin{array}{l} f_i \quad i=1, \dots, n \\ m_j = b_j \quad j=1, \dots, m \\ C() = b \\ \end{array} \\
 & \} \\
 cl ::= & \epsilon \mid c \ cl
 \end{aligned}$$

A class definition is similar to one in Java with all type declarations omitted. A class definition contains a list of field declarations, a list of method declarations and a constructor. All functions and the constructor are nullary.

This syntax is similar to the one of Featherweight Java [9] or Java-light [12] but the former is purely functional and the latter is typed and covers a larger subset of the Java syntax.

2.3 Object-Based

Recall that the syntax of the imperative untyped object calculus of [1] is as follows:

$a, b ::= x$		variable
		$[\mathbf{f}_i = b_i \ i=1..n, \mathbf{m}_j = \zeta(x_j) b_j \ j=1..m]$
		object creation
		$a.f$
		field select
		$a.f \leftarrow b$
		field update
		$a.m()$
		method call
		$a.m \leftarrow \zeta(x)b$
		method update
		$\text{clone}(a)$
		shallow copy
		$\text{let } x = a \text{ in } b$
		local def.

Note that, by contrast to [1], we do distinguish between fields and methods explicitly.

3 Denotational Semantics

In this section we give a denotational semantics for both languages, the class-based and the object-based one.

3.1 Preliminaries

When specifying the recursive types needed for the interpretation of object calculi we often have to employ record type formation in the following sense. Let \mathbb{L} be a (countable) set of labels and A a predomain then the type of records with entries from A and labels from \mathbb{L} is defined as follows:

$$\text{Rec}_{\mathbb{L}}(A) = \Sigma_{L \in \mathcal{P}_{\text{fin}}(\mathbb{L})} A^L$$

where A^L is the set of all total functions from L to A . It can be easily seen that $\text{Rec}_{\mathbb{L}}$ is a locally continuous functor on predomains with continuous maps. A record with labels l_i and corresponding entries v_i ($1 \leq i \leq k$) is written $\{l_1 = v_1, \dots, l_k = v_k\}$. Notice that $\text{Rec}_{\mathbb{L}}(A)$ is always non-empty as it contains the element $\langle \emptyset, \emptyset \rangle$.

Definition 1. *The update (and extension) operation for records is defined as in Table 1. If $r \in \text{Rec}_{\mathbb{L}}(A)$ and $l \in \mathbb{L}$ we will write $r.l \downarrow$ to express that l is in the domain of r , ie. label l is defined in r .*

Some other basic domains are needed. Let Loc be some countable set of locations (addresses) for objects and Val the set of basic untyped values like booleans or integers. Observe that $\text{Loc} \subseteq \text{Val}$, ie. locations can be seen as values and thus be stored in environments. Let $\text{Env} = \text{Loc}^{\text{Var}}$ denote the domain of local environments (or stacks) that are used to deal with bound variables.

Table 1. Record update and extension

$$\{\{m_i=f_i\}^{i=1\dots n}[m:=f]\} = \begin{cases} \{\{m_1=f_1, \dots, m_n=f_n, m=f\}\} & \text{if } m \text{ is different} \\ & \text{from all } m_i \\ \{\{m_1=f_1, \dots, m=f, \dots, m_n=f_n\}\} & \text{if } m = m_i \end{cases}$$

It is sensible to assume that \mathbf{Val} is a flat predomain, ie. any ascending chains in \mathbf{Val} consist of one element only. In the same way, \mathbf{Loc} is a flat predomain too.

In general, we use predomains (domains without a least element) as they are simpler to deal with. Note that the category of predomains and continuous partial maps (\rightarrow) is equivalent to the category of domains with strict continuous maps. As usual, \times binds stronger than \rightarrow .

As the concept of a “method” is important to class-based as well as object-based languages let us define a domain \mathbf{Cl} (for *closure*) in which methods (always without arguments in this paper) will find their interpretation:

$$\mathbf{Cl} = \mathbf{Loc} \times \mathbf{St} \rightarrow \mathbf{Val} \times \mathbf{St}$$

A closure is a partial function mapping a location, representing the callee object, and (the old) store to a value, the result of the method, and a new store which accounts for the side effects during execution of the method in question.

3.2 Class-Based

The class-based language introduced above finds its interpretation within the following system of predomains:

Definition 2. *Let the semantic counterparts of objects (1), stores (2), and method suites (3) be defined as below.*

- (1) $\mathbf{Ob} = \mathbf{Rec}_{\mathcal{F}}(\mathbf{Val}) \times \mathcal{C}$
- (2) $\mathbf{St} = \mathbf{Rec}_{\mathbf{Loc}}(\mathbf{Ob})$
- (3) $\mathbf{Ms} = \mathbf{Rec}_{\mathcal{C}}(\mathbf{Rec}_{\mathcal{M}}(\mathbf{Cl}) \times (\mathbf{St} \rightarrow \mathbf{Loc} \times \mathbf{St}))$

Note that an object consists of a record of field values plus the name of its class type. This information is stored in order to be able to do the dynamic dispatch for method calls (due to subtype polymorphism). A *method suite* is a record $\mathbf{Rec}_{\mathcal{M}}(\mathbf{Cl}) \times (\mathbf{St} \rightarrow \mathbf{Loc} \times \mathbf{St})$ which contains all methods of a class including the constructor `new`. Therefore \mathbf{Ms} is the record of all method suites for all declared classes. It should be stressed that the definition of \mathbf{St} as $\mathbf{Rec}_{\mathbf{Loc}}(\mathbf{Ob})$ faithfully reflects the idea of a state as an assignment of objects to a finite set of locations. Moreover, this formulation, by contrast to the functional modelling of states, has the advantage that \mathbf{St} is a *flat* predomain.

The function $\text{type} : \mathbf{Ob} \rightarrow \mathcal{C}$ that returns an object’s type, ie. $\text{type}(o) = o.2$ will be useful later.

The interpretation of the syntax is given by the following semantic equations:

Definition 3. Given an environment $\rho \in \text{Env}$, an environment of method suites $\mu \in \text{Ms}$ and an expression a its interpretation $\llbracket a \rrbracket \rho : \text{Ms} \rightarrow \text{St} \rightarrow \text{Val} \times \text{St}$ is defined in Table 2 (page 478). Note that the semantic **let** on the right hand side of the definitions is strict in its first argument.

Table 2. Denotational semantics of expressions (class-based)

$$\begin{array}{ll}
\llbracket x \rrbracket \rho \mu \sigma & = \langle \rho(x), \sigma \rangle \\
\llbracket \text{this} \rrbracket \rho \mu \sigma & = \langle \rho(\text{this}), \sigma \rangle \\
\llbracket \text{new } C() \rrbracket \rho \mu \sigma & = \mu.C.2(\sigma) \\
\llbracket a.f \rrbracket \rho \mu \sigma & = \text{let } \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \mu \sigma \text{ in } \sigma'.\ell.1.f \\
\llbracket a.f=b \rrbracket \rho \mu \sigma & = \text{let } \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \mu \sigma \text{ in} \\
& \quad \text{let } \langle v, \sigma'' \rangle = \llbracket b \rrbracket \rho \mu \sigma' \text{ in } \langle v, \sigma''[\ell := \langle \sigma''.\ell.1[f := v], \sigma''.\ell.2] \rangle \\
\llbracket a.m() \rrbracket \rho \mu \sigma & = \text{let } \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \mu \sigma \text{ in } \mu.\text{type}(\sigma'.\ell).1.m(\langle \ell, \sigma' \rangle) \\
\llbracket \text{let } x=a \text{ in } b \rrbracket \rho \mu \sigma & = \text{let } \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \mu \sigma \text{ in } \llbracket b \rrbracket \rho[\ell/x] \mu \sigma' .
\end{array}$$

Given an environment of method suites $\mu \in \text{Ms}$ a class definition c finds its interpretation $\llbracket c \rrbracket : \text{Ms} \rightarrow \text{Ms}$ as defined in Table 3 (page 478). Accordingly, the semantics of a list of class definitions is as follows:

$$\begin{array}{ll}
\llbracket c \text{ cl} \rrbracket \mu & = \llbracket cl \rrbracket \mu[C := \llbracket c \rrbracket \mu] \\
\llbracket \epsilon \rrbracket \mu & = \mu
\end{array}$$

As this does not allow one to write a recursive class definition the right semantics $\llbracket cl \rrbracket : \text{Ms}$ of a class definition list is as follows

$$\llbracket cl \rrbracket = \text{fix } \lambda \mu. \llbracket cl \rrbracket \mu$$

where the usage of the fixpoint operator is justified since Ms is a domain and $\llbracket cl \rrbracket : \text{Ms} \rightarrow \text{Ms}$ is a continuous function between domains. Continuity is a consequence

Table 3. Denotational semantics of class definitions

$$\left[\left[\begin{array}{l} \text{class } C \{ \\ \quad f_i \quad i=1..n \\ \quad m_j = b_j \quad j=1..m \\ \quad C() = b \\ \quad \} \end{array} \right] \right] \mu = \left\langle \begin{array}{l} \{\{m_j = \lambda \langle \ell, \sigma \rangle. \llbracket b_j \rrbracket \rho[\ell/\text{this}] \mu \sigma\}^{j=1..m}, \\ \lambda \sigma. \langle \ell, \llbracket b \rrbracket \rho[\ell/\text{this}] \mu \sigma[\ell := \langle \{f_i = \text{init}_{C,i}\}^{i=1..n}, C] \rangle] \end{array} \right\rangle$$

where ℓ is a fresh location not in the domain of σ and $\text{init}_{C,i}$ denotes a default value for field f_i in C .

Table 4. Denotational semantics of the imperative object calculus
$$\begin{array}{l}
\llbracket x \rrbracket \rho \sigma = \langle \rho(x), \sigma \rangle \\
\llbracket [f_i =_{\zeta}(x_i) b_i \quad i=1..n, \\ m_j =_{\zeta}(x_j) b_j \quad j=1..m] \rrbracket = \left\langle \ell, \sigma[\ell := \langle \{f_i = \llbracket b_i \rrbracket \rho \sigma\}^{i=1..n}, \\ \{m_j = \lambda \langle \ell', \sigma' \rangle. \llbracket b_j \rrbracket \rho[\ell'/x_j] \sigma'\}^{j=1..m} \rangle] \right\rangle \\
\text{where } \ell \text{ is a fresh location not in the domain of } \sigma \\
\llbracket a.f \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \sigma'.\ell.1.f \\
\llbracket a.f \leftarrow b \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \\
\langle \ell, \sigma'[\ell := \langle \sigma'.\ell.1[f := \llbracket b \rrbracket \rho \sigma'], \sigma'.\ell.2] \rangle \\
\llbracket a.m() \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \sigma'.\ell.2.m(\langle \ell, \sigma' \rangle) \\
\llbracket a.m \leftarrow_{\zeta}(x) b \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \\
\langle \ell, \sigma'[\ell := \langle \sigma'.\ell.1, \sigma'.\ell.2[m := \lambda \langle \ell', \sigma'' \rangle. \llbracket b \rrbracket \rho[\ell'/x] \sigma''] \rangle] \rangle \\
\llbracket \mathbf{clone}(a) \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \langle \ell', \sigma'[\ell' := \sigma'.\ell] \rangle \\
\text{where } \ell' \text{ is a fresh location not in the domain of } \sigma' \\
\llbracket \mathbf{let} x = a \mathbf{in} b \rrbracket \rho \sigma = \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket a \rrbracket \rho \sigma \mathbf{in} \llbracket b \rrbracket \rho[\ell/x] \sigma' .
\end{array}$$

of the definitions in Tables 2 and 3. All interpretations depending on the method suites (method application and object creation) are continuous in μ .

3.3 Object-Based

The imperative object calculus finds its interpretation within the following system of recursive types:

Definition 4. Let the semantic counterparts of objects (1) and stores (2) be defined as below:

- (1) $\mathbf{Ob} = \mathbf{Rec}_{\mathcal{F}}(\mathbf{Val}) \times \mathbf{Rec}_{\mathcal{M}}(\mathbf{Cl})$
- (2) $\mathbf{St} = \mathbf{Rec}_{\mathbf{Loc}}(\mathbf{Ob})$

In the following we will write $\mathbf{St}_{\mathbf{Val}}$ for $\mathbf{Rec}_{\mathbf{Loc}}(\mathbf{Rec}_{\mathcal{F}}(\mathbf{Val}))$. There is an obvious projection $\pi_{\mathbf{Val}} : \mathbf{St} \rightarrow \mathbf{St}_{\mathbf{Val}}$ with $\pi_{\mathbf{Val}}(\sigma).\ell \simeq \pi_1(\sigma.\ell)$ where π_1 projects on the first component.

The interpretation of the syntax is given by the following semantic equations:

Definition 5. Given an environment $\rho \in \mathbf{Env} = \mathbf{Loc}^{\mathbf{Var}}$ and an object expression a its interpretation $\llbracket a \rrbracket \rho : \mathbf{St} \rightarrow \mathbf{Loc} \times \mathbf{St}$ is defined in Table 4 (page 479).

In the object-based case the semantics of field and method select are identical. So are the semantics of field and method update. The only difference is that the fields are of the simpler type \mathbf{Val} .

Note that we can still write recursive methods although semantically there is no fixpoint of recursive definitions as for the class-based case. The recursion

works simply “through the store”, it is established by self application. For example, if $\llbracket [m = \zeta(x)x.m()] \rrbracket \rho \sigma = \langle \ell, \sigma^n \rangle$ where

$$\sigma^n := \sigma[\ell := \langle \emptyset, \{[m = \lambda(\ell', \sigma'). \sigma'.\ell'.2.m(\langle \ell', \sigma' \rangle)]\} \rangle]$$

we obtain

$$\begin{aligned} \llbracket [m = \zeta(x)x.m()] . m() \rrbracket \rho \sigma &= \mathbf{let} \langle \ell, \sigma' \rangle = \llbracket [m = \zeta(x)x.m()] \rrbracket \rho \sigma \mathbf{in} \\ &\quad \sigma'.\ell'.2.m(\langle \ell, \sigma' \rangle) \\ &= \sigma^n.\ell'.2.m(\langle \ell, \sigma^n \rangle) \\ &= \sigma^n.\ell'.2.m(\langle \ell, \sigma^n \rangle) \\ &= \dots \end{aligned}$$

which does not yield a defined result.

4 Logics of Programs

Once the denotational semantics of classes and objects is settled one can start reasoning about these denotations. This is in analogy with the LCF (Logic of Computable Functions) project (see e.g. [13]) for the functional paradigm. In LCF one reasons about functional programs by referring to their semantic (fix-point) interpretation using results of domain theory. An analogous procedure will be carried out for the object-oriented languages defined above. Reasoning about classes and objects will be done by referring to their semantics defined in the previous section.

4.1 Specifications

First we have to define an appropriate notion of *specification* for classes and objects. For the class-based case we follow [8, 15, 20], for the object-based case [2]. Whereas in the cited approaches specifications are defined syntactically and their semantics implicitly by proof rules (verified w.r.t. some operational semantics) we work consistently on the denotational level. The more abstract description leads to a conceptually clearer understanding of the principles involved and to better structured soundness proofs.

As every class and every object in the object-based language contain a number of methods, some common structures in specifications can be singled out: for every method there is a result specification and a transition specification. This distinction is usually blurred in class-based verification calculi, but we will see below why it is important for object-based specifications.

$$\begin{array}{ll} B_m \in \wp(\mathbf{Val} \times \mathbf{St}) & \text{result specification} \quad \text{for } m \in \mathcal{M} \\ T_m \in \wp(\mathbf{Loc} \times \mathbf{St} \times \mathbf{Val} \times \mathbf{St}) & \text{transition specification for } m \in \mathcal{M} \end{array}$$

Taking into account that we are only interested in partial correctness (at least for the considerations of this paper) the meaning of the specifications can be described informally as follows:

B_m : “If method m terminates its result fulfils the result specification.”
 T_m : “If method m terminates its effect fulfils the transition specification.”

An object is given by a location and the store it resides in and a class by its name and the method suites it defines. Therefore, a class specification is a predicate in $\wp(\mathcal{C} \times \text{Ms})$ whereas an object specification is a predicate in $\wp(\text{Loc} \times \text{St})$.

4.2 Class-Based

Definition 6. *Specifications of methods, Spec_C^m , classes Spec_C , and packages, \mathbf{Spec}_C , are defined below:*

$$\begin{aligned} \text{Spec}_C^m &: \wp(\text{Val} \times \text{St}) \times \wp(\text{Loc} \times \text{St} \times \text{Val} \times \text{St}) \rightarrow \wp(\mathcal{C} \times \text{Cl}) \\ \text{Spec}_C &: \text{Rec}_{\mathcal{M}}(\wp(\text{Val} \times \text{St}) \times \wp(\text{Loc} \times \text{St} \times \text{Val} \times \text{St})) \rightarrow \wp(\mathcal{C} \times \text{Ms}) \\ \mathbf{Spec}_C &: \text{Rec}_{\mathcal{C}}(\text{Rec}_{\mathcal{M}}(\wp(\text{Val} \times \text{St}) \times \wp(\text{Loc} \times \text{St} \times \text{Val} \times \text{St}))) \rightarrow \wp(\text{Ms}) \\ \langle \mathcal{C}, f \rangle \in \text{Spec}_C^m(B_m, T_m) &\text{ iff } \forall \ell \in \text{Loc}. \forall v \in \text{Val}. \forall \sigma, \sigma' \in \text{St}. \\ &(\text{type}(\sigma.\ell) = \mathcal{C} \wedge f(\ell, \sigma) = \langle v, \sigma' \rangle) \Rightarrow B_m(v, \sigma') \wedge T_m(\ell, \sigma, v, \sigma') \\ \langle \mathcal{C}, \mu \rangle \in \text{Spec}_C(r) &\text{ iff } \forall m \in \mathcal{M}. \\ &r.\mathcal{C}.m \downarrow \Rightarrow \mu.\mathcal{C}.1.m \downarrow \wedge \langle \mathcal{C}, \mu.\mathcal{C}.1.m \rangle \in \text{Spec}_C^m(r.\mathcal{C}.m) \\ \mu \in \mathbf{Spec}_C(R) &\text{ iff } \forall \mathcal{C} \in \mathcal{C}. R.\mathcal{C} \downarrow \Rightarrow \mu.\mathcal{C} \downarrow \wedge \langle \mathcal{C}, \mu \rangle \in \text{Spec}(R.\mathcal{C}) \end{aligned}$$

Typical examples of transition specifications T_m are specifications à la Hoare:

$$T_m(\ell, \sigma, v, \sigma') \equiv P(\ell, \sigma) \Rightarrow Q(\ell, \sigma, v, \sigma')$$

Note that P may refer to the old state σ as well as to the value of the self parameter **this**, ℓ . On the other hand, Q may refer to the self parameter ℓ , to the new state σ' after method execution and to the result v of the method. It may also refer to the pre-state σ in order to be able to talk about the “old values” of variables.²

Theorem 1. *The specifications Spec_C^m , Spec_C , and \mathbf{Spec}_C always exist and are admissible predicates.*

Proof. Existence is trivial as the specifications are defined explicitly. This point is only emphasized to address the difference with respect to object specifications. Let us show admissibility for Spec_C^m , the other cases work analogously. First we rewrite Spec_C^m :

$$\begin{aligned} \langle \mathcal{C}, f \rangle \in \text{Spec}_C^m(B_m, T_m) &\text{ iff } \forall \ell \in \text{Loc}. \forall v \in \text{Val}. \forall \sigma, \sigma' \in \text{St}. \\ &(\text{type}(\sigma.\ell) \neq \mathcal{C} \vee f(\ell, \sigma) \uparrow \vee (B_m(f(\ell, \sigma)) \wedge T_m(\ell, \sigma, f(\ell, \sigma)))) \end{aligned}$$

As Cl is a flat predomain admissibility must only be shown with respect to the second component f . Since admissible predicates are closed under universal quantification, disjunction, conjunction, and composition with continuous maps (and function application is continuous) it only remains to show that \uparrow , B_m , and T_m are admissible. The former is by definition, the latter are because they are predicates on a flat predomain.

² Alternatively, one can employ logical variables to talk about old values of variables.

In order to prove that a specification is true for a program (package) one has to apply fixpoint induction. The corresponding proof rule reads as follows:

$$\frac{\mu \in \mathbf{Spec}_C(R) \Rightarrow \llbracket cl \rrbracket \mu \in \mathbf{Spec}_C(R)}{\text{fix } \lambda \mu. \llbracket cl \rrbracket \mu \in \mathbf{Spec}_C(R)}$$

which presupposes that the specification is admissible.

This is a denotational equivalent of the inductive proof rule presented e.g. in [21]³.

Which method is executed for a method application $o.m()$ actually depends on the type of o : $\text{type}(\llbracket o \rrbracket)$. For verification purposes this is problematic as the dynamic type of an object is not known at verification (ie. compile) time. In [20] a special method triple $\{P\}C :: m()\{Q\}$ was suggested with the intuitive meaning that all methods m in class C and all subclasses of C fulfil the specification in terms of pre- and post-conditions P and Q . Moreover, the following proof rule was proposed for any method m provided it does not depend on other methods.

$$(\dagger) \quad \frac{\forall C' \leq C. \{P\}C' :: m()\{Q\} \vdash \{P\} \text{body}(C' :: m)\{Q\}}{\{P\}C :: m()\{Q\}}$$

where $\text{body}(C' :: m)$ denotes the body of the corresponding method. This rule can be verified w.r.t. the operational semantics by induction on the number of unfoldings of method m . A less tedious soundness proof uses fixpoint induction and a slightly stronger notion of specification:

Definition 7. *Specifications of packages obeying behavioural subtyping:*

$$\begin{aligned} \mathbf{Spec}_C^{\text{beh}} &: \text{Rec}_C(\text{Rec}_{\mathcal{M}}(\wp(\text{Val} \times \text{St}) \times \wp(\text{Loc} \times \text{St} \times \text{Val} \times \text{St}))) \rightarrow \wp(\text{Ms}) \\ \mu \in \mathbf{Spec}_C^{\text{beh}}(R) &\text{ iff } \forall C \in \mathcal{C}. \forall C' \geq C. R.C' \downarrow \Rightarrow \langle C, \mu \rangle \in \text{Spec}_C(R.C') \end{aligned}$$

Thus, a package μ fulfils specification $\mathbf{Spec}_C^{\text{beh}}(R)$ if all classes in μ are correct w.r.t. their own class definition but also the specifications of all super classes. By Theorem 1 and the fact that admissible predicates are closed under conjunction one obtains that $\text{Spec}_C^{\text{beh}}$ is admissible, too.

Theorem 2. *If the method m in class C is recursive but does not depend on other methods, then the rule (\dagger) is sound w.r.t. our semantics.*

Proof. Let R be the specification $\mathbf{Spec}_C(\{C = \{m = \langle \text{True}, T_m \rangle\}\})$ where $T_m(\ell, \sigma, v, \sigma') = P(\ell, \sigma) \Rightarrow Q(\ell, v, \sigma')$. Then the semantic counterpart of the rule above is:

$$\frac{\forall C' \leq C. \mu \in \text{Spec}_C^{\text{beh}}(R) \Rightarrow (\llbracket cl \rrbracket \mu).C'.1.m \in \text{Spec}_C^m(\text{True}, T_m)}{\text{fix } \lambda \mu. \llbracket cl \rrbracket \mu \in \text{Spec}_C^{\text{beh}}(R)}$$

³ More precisely, the semantical equivalent of [21] is rather computational induction which implies fixpoint induction.

which is equivalent to

$$\frac{\mu \in \text{Spec}_C^{\text{beh}}(R) \Rightarrow \forall C' \leq C. (\llbracket cl \rrbracket \mu).C'.1.m \in \text{Spec}_C^m(\text{True}, T_m)}{\text{fix } \lambda \mu. \llbracket cl \rrbracket \mu \in \text{Spec}_C^{\text{beh}}(R)}$$

Under the given assumptions it is easy to show that

$$\forall C' \leq C. (\llbracket cl \rrbracket \mu).C'.1.m \in \text{Spec}_C^m(\text{True}, T_m) \text{ iff } \llbracket cl \rrbracket \mu \in \mathbf{Spec}_C^{\text{beh}}(R)$$

which then is equivalent to the valid fixpoint induction rule

$$\frac{\mu \in \text{Spec}_C^{\text{beh}}(R) \Rightarrow \llbracket cl \rrbracket \mu \in \text{Spec}_C^{\text{beh}}(R)}{\text{fix } \lambda \mu. \llbracket cl \rrbracket \mu \in \text{Spec}_C^{\text{beh}}(R)}$$

Apart from the method specifications T_m , class specifications could be extended to contain requirements on the constructor **new** (as well as on any static method or field).

Given method suites μ , a class invariant for C is a predicate on *objects* $I \in \wp(\text{Loc} \times \text{St})$ such that all methods $\mu.C.m$ in class C fulfil the method specification

$$T_m(\ell, \sigma, v, \sigma') \equiv I(\ell, \sigma) \Rightarrow I(\ell, \sigma')$$

and object creation establishes I , in other words $I(\mu.C.2(\sigma))$ for all $\sigma \in \text{St}$.

4.3 Object-Based

Due to self application the argument of an object's method is the object itself. Proving properties of an object means proving properties of its methods but this requires that the object itself already satisfies the specification.

A first tentative definition of an object specification is given below. Note that we add an additional component $A \in \wp(\text{Loc} \times \text{St})$ to describe properties of the fields not covered by the method result and transition specifications.

Definition 8. *A tentative object specifications is as follows:*

$$\begin{aligned} \text{Spec}_o &: (\wp(\text{Loc} \times \text{St}) \times \text{Rec}_{\mathcal{M}}(\wp(\text{Val} \times \text{St}) \times \wp(\text{Loc} \times \text{St} \times \text{Val} \times \text{St}))) \\ &\quad \rightarrow \wp(\text{Loc} \times \text{St}) \\ \langle \ell, \sigma \rangle &\in \text{Spec}_o(A, \{\mathbf{m}_j = \langle B_{\mathbf{m}_j}, T_{\mathbf{m}_j} \rangle\}^{j=1..m}) \text{ iff } \forall \mathbf{m} \in \mathcal{M}. \forall v \in \text{Val}. \forall \sigma' \in \text{St}. \\ &\quad \langle \ell, \sigma \rangle \in A \wedge ((\langle \ell, \sigma \rangle \in \text{Spec}_o(A, \{\mathbf{m}_j = \langle B_{\mathbf{m}_j}, T_{\mathbf{m}_j} \rangle\}^{j=1..m}) \\ &\quad \wedge \sigma.l.m(\ell, \sigma) = \langle v, \sigma' \rangle \Rightarrow B_m(v, \sigma') \wedge T_m(\ell', \sigma, v, \sigma')) \end{aligned}$$

The above definition is implicit and due to the contravariant occurrence of Spec_o it is not clear that the specification exists. If all constituent predicates are monotonic one can define the specification to be the fixpoint of the defining functional. But the least fixpoint would not give the intended meaning (as methods would simply be required to be undefined) and the greatest one just leaves us with

co-induction as appropriate proof principle which. Co-induction requires an invariant that is often not easy to find or that turns out to be just the specification itself (for an example see [19]).

A more intuitive way to prove objects correct is given in [2]. As the argument of a method do not assume the callee itself but an arbitrary object obeying the object specification.

Definition 9. *The generalised notion of object specification is defined below:*

$$\text{Spec}_O : ((\wp(\text{Loc} \times \text{St}) \rightarrow \wp(\text{Loc} \times \text{St})) \times \\ \text{Rec}_{\mathcal{M}}((\wp(\text{Val} \times \text{St}_{\text{Val}}) \rightarrow \wp(\text{Val} \times \text{St})) \times \wp(\text{Loc} \times \text{St}_{\text{Val}} \times \text{Val} \times \text{St}_{\text{Val}}))) \\ \rightarrow \wp(\text{Loc} \times \text{St}))$$

Let $\text{Spec}_o(A, \{\mathfrak{m}_j = \langle B_{\mathfrak{m}_j}, T_{\mathfrak{m}_j} \rangle\}^{j=1..m})$ be the predicate $S \in \wp(\text{Loc} \times \text{St})$ with

$$(\ell, \sigma) \in S \equiv A(S)(\ell, \sigma) \wedge \\ \forall \mathfrak{m} \in \mathcal{M}. \forall \ell' \in \text{Loc}. \forall \sigma' \in \text{St}. \langle \ell', \sigma' \rangle \in S \Rightarrow \\ \forall v \in \text{Val}. \forall \sigma'' \in \text{St}. \\ \sigma.l.2.m(\ell', \sigma') = (v, \sigma'') \Rightarrow B_m(S)(v, \sigma'') \wedge T_m(\ell', \pi_{\text{Val}}(\sigma'), v, \pi_{\text{Val}}(\sigma''))$$

provided S is unique with the above property.

Note that we added some recursive dependencies: A and B_m may use Spec_o recursively. Properties of fields or results that are (locations of) objects themselves can be specified in terms of that object's class type. In the object-based case one cannot refer to a class specification so one must ensure that it is possible to refer to the specification one is about to define recursively.

The following theorem states an existence theorem for object specifications.

Theorem 3. *Given admissible A , $B_{\mathfrak{m}_j}$ and $T_{\mathfrak{m}_j}$ as in the definition above where A and B satisfy the following conditions:*

- (i) $e : X \subseteq X'$ implies $F(e, e) : A(X) \subseteq A(X')$ for all $e \sqsubseteq \text{id}_{\text{St}}$.
- (ii) for all $\mathfrak{m} \in \mathcal{M}$, $e : X \subseteq X'$ implies $F(e, e) : B_{\mathfrak{m}}(X) \subseteq B_{\mathfrak{m}}(X')$ for all $e \sqsubseteq \text{id}_{\text{St}}$.

where $F(Y, X) = \text{Rec}_{\text{Loc}}(\text{Rec}_{\mathcal{F}}(\text{Val}) \times \text{Rec}_{\mathcal{M}}(\text{Loc} \times Y \rightarrow \text{Val} \times X))$ and $e : X \subseteq Y$ abbreviates $\forall o : \text{Loc} \times \text{St}. o \in X \wedge e(o) \downarrow \Rightarrow e(o) \in Y$. Then for Φ defined as

$$\Phi(Y, X)(\ell, \sigma) \equiv A(X)(\ell, \sigma) \wedge \\ \forall \mathfrak{m} \in \mathcal{M}. \forall \ell' \in \text{Loc}. \forall \sigma' \in \text{St}. \langle \ell', \sigma' \rangle \in Y \Rightarrow \\ \forall v \in \text{Val}. \forall \sigma'' \in \text{St}. \\ \sigma.l.m(\ell', \sigma') = (v, \sigma'') \Rightarrow B_m(X)(v, \sigma'') \wedge T_m(\ell', \pi_{\text{Val}}(\sigma'), v, \pi_{\text{Val}}(\sigma''))$$

there exists a unique admissible $S \in \wp(\text{Loc} \times \text{St})$ with $S = \Phi(S, S)$.

Proof. The proof uses Andrew Pitts' relational properties of domains [14] and can be found in [19].

Table 5. In a nutshell: class-based vs. object-based semantics

	class based	object based
object	$Ob = Rec_{\mathcal{F}}(Val) \times \mathcal{C}$	$Ob = Rec_{\mathcal{F}}(Val) \times Rec_{\mathcal{M}}(Cl)$
store	$St = Rec_{Loc}(Ob)$	$St = Rec_{Loc}(Ob)$
meth's	$Ms = Rec_{\mathcal{C}}(Rec_{\mathcal{M}}(Cl) \times (St \rightarrow Loc \times St))$	
unfolded	$St = Rec_{Loc}(Rec_{\mathcal{F}}(Val) \times \mathcal{C})$	$St = Rec_{Loc}(Rec_{\mathcal{F}}(Val) \times Rec_{\mathcal{M}}(Loc \times St \rightarrow Val \times St))$
recursion	fixpoint of $\llbracket cl \rrbracket : Ms \rightarrow Ms$	by self application “through store”
spec type	$\wp(\mathcal{C} \times Ms)$	$\wp(Loc \times St)$
existence	trivial	only for admissible specs if T_m refers to flat part of σ only
proof rule	fixpoint induction	the specification itself
admissibility	required but always guaranteed	required

This theorem guarantees existence only if T_m does not refer to the method part of the store. It is yet unknown whether existence can be ensured otherwise (in a less restrictive way). As a consequence, method update is problematic from a specification point of view. The transitions specification cannot express any changes caused by updates if they are not allowed to refer to the method part of the object (store). Method update is considered difficult also in [5] but for a different reason. It is hard to include in the strongly typed encodings.

We are now in a position to explain why we have distinguished between T_m and B_m in the first place. B_m is allowed to refer to the higher-order (method) part of the store whereas T_m is not. This has useful consequences as B_m can be used to express method invariants. If a specification for $\langle \ell, \sigma \rangle$ uses a B_m defined as

$$\langle v, \sigma' \rangle \in B_m(X) \equiv \langle \ell, \sigma' \rangle \in X$$

it is stipulated that the object ℓ still fulfils the specification (represented by X but later bound by recursion) after execution of m in the new state σ' .

Another advantage is that just by definition of $Spec_O$ the object introduction rule of [2] is correct in our semantics. It explains much better why the calculus is sound than a rather involved induction on derivations.

5 Comparison

Unfolding the definitions of St for the class-based and the object based language one immediately observes a difference as emphasized in Table 5. It is obvious that the domain of interpretation for the object-based case is itself defined recursively, and that the right hand side of its definition even contains a negative occurrence. This is a particular instance of a concept known as “recursion through the store”. Recursive methods can be implemented without the explicit recursion of the class-based languages by calling some code residing

in the store. Specifications are not recursively defined and thus their existence is trivial in the class-based case whereas it needs some heavy domain theoretic machinery to guarantee existence of the object specifications. In the latter case existence requires already the constituent predicates to be admissible and monotonic. Transition specifications must not refer to methods, they may just talk about the fields of objects, but cannot express effects of method updates.

Finally, the main proof rule for the class-based approach is fixpoint induction triggered by the recursive definition of method suites. For the object-calculus the definition of the specifications provides the proof rule that uses as assumption – as “induction hypothesis” – that the specification holds for the arbitrary object that is used as callee of the method in question.

6 Conclusion

The comparison between a typical class-based and a typical object-based language in terms of denotational semantics reveals that the semantic domains and the logic on top of these domains are much more involved in the object-based case. In the latter case, specifications may not exist if method update is allowed. For both languages recursion is involved, although on different levels: for class-based languages only the method suites are recursively defined, for object-based languages the underlying semantic domains and the specifications are recursively defined but not the methods themselves. Due to the recursion, all predicates used for program verification must be admissible in both cases. But only for the class-based language this can be guaranteed in general.

The semantics demonstrate in an intuitive way that crucial proof rules for the logics in [20] and [2] are correct.

Further research includes full proofs of the soundness of the program logics [20, 2] as well as a typed variant of the presented denotational semantics. Other natural extensions include method parameters and access restrictions (data encapsulations). Static methods can easily be incorporated into the class-based approach following the scheme of the nullary constructor. Also abnormal termination (exception handling) and control flow operators could be incorporated (cf. [10]).

One significant advantage of the presented approach is that it does not commit itself to any particular form of logic so other verification calculi can be treated in the very same fashion. In particular it is independent of the underlying (usually first-order) logic. Therefore, our analysis is orthogonal to and thus compatible with the *spatial logic* suggested by [6, 7].

Acknowledgement

Thanks for discussions about class-based and object-based semantics go to Thomas Streicher, but also to Cristiano Calcagno, Peter O’Hearn and Uday Reddy. The anonymous referees gave valuable help to improve the presentation.

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996. 473, 474, 476
- [2] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development: Proceedings / TAPSOFT '97, 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, 1997. 473, 480, 484, 485, 486
- [3] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.* Springer, Berlin, 1999. 487
- [4] P. America and F. S. de Boer. A proof theory for a sequential version of POOL. Technical Report <http://www.cs.uu.nl/people/frankb/Available-papers/spool.dvi>, University of Utrecht, 1999. 473
- [5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS'97), Sendai, Japan*, volume 1281, pages 415–438. Springer LNCS, September 1997. 474, 485
- [6] C. Calcagno, S. Ishtiaq, and P. W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Principles and Practice of Declarative Programming*. ACM Press, 2000. 486
- [7] C. Calcagno and P. W. O'Hearn. A logic for objects. Talk given in March 2001, 2001. 486
- [8] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computations Structures*, volume 1578 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. 473, 480
- [9] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java. A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 34, pages 132–146. ACM SIGPLAN Notices, 1999. 475
- [10] B. Jacobs and E. Poll. A monad for basic java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 150–164. Springer Verlag, 2000. 473, 486
- [11] S. N. Kamin and U. S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994. 473
- [12] T. Nipkow and D. von Oheimb. Machine-checking the Java Specification: Proving Type-Safety. In Alves-Foss [3]. 475
- [13] L. C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987. 474, 480
- [14] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 321, December 1993.). 474, 484
- [15] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods*, 1998. 473, 480
- [16] U. Reddy. Objects and classes in Algol-like languages. In *FOOL 5*, 1998. Available at URL <http://pauillac.inria.fr/~remy/fool/proceedings.html>. 473
- [17] B. Reus. A logic of recursive objects. In *Formal Techniques for Java Programs*, volume 251 - 5/1999 of *Informatik Berichte*, pages 58–64. FernUniversität Hagen, 1999. 474

- [18] B. Reus. A logic of recursive objects (abstract). In S. Demeyer A. Moreira, editor, *Object-oriented Technology, ECOOP '99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, page 107, Berlin, 1999. Springer. 474
- [19] B. Reus and Th. Streicher. Semantics and logics of objects. In *Proceedings of the 17th Symp. Logic in Computer Science*, 2002. To appear. Draft available from www.cogs.susx.ac.uk/users/bernhard/papers.html. 474, 484
- [20] B. Reus, M. Wirsing, and R. Hennicker. A Hoare-Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In *FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 300–317, Berlin, 2001. Springer. 473, 480, 482, 486
- [21] D. von Oheimb. Hoare logic for mutual recursion and local variables. In V. Raman C. Pandu Rangan and R. Ramanujam, editors, *Found. of Software Techn. and Theoret. Comp. Sci.*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999. 482

BRAIN: Backward Reachability Analysis with Integers

Tatiana Rybina and Andrei Voronkov

University of Manchester
{rybina,voronkov}@cs.man.ac.uk

1 General Description

BRAIN is a tool for analysis of infinite-state systems in which a state can be represented by a vector of integers. Currently it is used to verify safety and deadlock properties expressed as reachability statements in a quantifier-free language with variables ranging over natural numbers.

2 Input Language

The input language of BRAIN consists of problem descriptions specifying the following.

1. The sets of initial and final states by a quantifier-free formula of Presburger arithmetic. Instead of final states, one can specify the set of deadlock states, which will be computed by BRAIN automatically from the description of transitions.
2. Transitions expressed as *guarded assignments* `guard->assignment` (see [7]), where `guard` is again a quantifier-free formula of Presburger arithmetic and `assignment` is a sequence of assignments to variables. Examples will be given below.
3. Queries of two types: reachability of a set of states from another set of states, or reachability of a deadlock state from a set of states.

An example specification of the IEEE Futurebus protocol (see [6]) is given in Figure 1.

3 Transition Systems

The semantics of a problem expressed in the language can be defined as follows. Let \mathcal{V} be the set of all variables declared in the problem (see the keyword `var` in Figure 1). We call a *state* a mapping from the set of variables into the corresponding domain (currently, only the domain of natural numbers is supported). Such a mapping can be homomorphically extended to arbitrary terms of Presburger arithmetic over \mathcal{V} by defining $s(n) = n$, for every non-negative integer n

```

%                               transition read_shared_3
%   problem name                i >= 1 & e = 0 ->
%                               s := s + 1,
problem futurebus              e := 0,
  var                            i := i - 1
    s : nat,                      transition read_modified
    e : nat,                      s >= 1 ->
    i : nat                       s := 0,
%                               e := e + 1,
%   specifications of the sets   i := i + s - 1
%   of initial and final states  transition write_back
%                               e >= 1 ->
state initial                   e := e - 1,
  s = 0 & e = 0 & i >= 1         i := i + 1
state error                      %
  e >= 2                         % a query: is any error state
%                               % reachable from an initial
%   transitions expressed by    % state?
%   guarded assignments        %
%                               query unsafe
transition read_shared_1        error <-- initial
  i >= 1 & e >= 1 ->            %
  s := s + 2,                  % alternatively, we can ask if
  e := e - 1,                  % a deadlock state is reachable
  i := i - 1                    %
transition read_shared_2        % query dead
  i >= 1 & e >= 1 ->          % deadlock <-- initial
  s := s + 1,
  e := e - 1

```

Fig. 1. A specification of the IEEE Futurebus protocol

and $s(u_1 + u_2) = s(u_1) + s(u_2)$. A *guard* G is a quantifier-free formula of Presburger arithmetic with variables in \mathcal{V} . In addition to the function symbol $+$ and equality BRAIN allows one to use $-$ and all the standard comparison operators, such as \leq . If a formula C of variables \mathcal{V} is true in a state s , we write $s \models C$. Every such formula C *symbolically represents* a set of states, namely the set of states in which it is true, i.e., $\{s \mid s \models C\}$.

A *transition* t is a set of pairs of states. Transitions are declared in BRAIN using *guarded assignments* of the form

$$G \rightarrow v_1 := u_1, \dots, v_n := u_n,$$

where all the v_i 's are variables and the u_i 's are terms. This guarded assignment defines a (deterministic) transition t with the following properties: $(s, s') \in t$ if $s \models G$, $s'(v_1) = s(u_1), \dots, s'(v_n) = s(u_n)$, and for every variable $v \in \mathcal{V} - \{v_1, \dots, v_n\}$ we have $s'(v) = s(v)$. Though every guarded assignment specifies

a deterministic transition, the transition relation specified by a set of guarded assignments may be non-deterministic.

One can generalize the notion of transition system in at least two ways:

1. Conditions can be arbitrary formulas of Presburger arithmetic, not necessarily quantifier-free.
2. One can consider a collection of variables \mathcal{V}' of the same size as \mathcal{V} and use in actions arbitrary formulas over $\mathcal{V} \cup \mathcal{V}'$. Our sequence of assignments $v_1 := u_1, \dots, v_n := u_n$ can then be rewritten as a formula $v'_1 = u_1 \wedge \dots \wedge v'_n = u_n$.
3. Formulas of temporal logic can be used instead of reachability statements.
4. Domains other than natural numbers can be used.

Some of these generalizations are incorporated in the system described in [3]. We cannot implement them in the current version of **BRAIN** immediately since this would require different algorithms for checking reachability. We are planning to extend the system to a more general one.

4 Why Backward Reachability

Our backward reachability algorithm uses systems of linear equalities and inequalities over nonnegative integers. We call such systems *simple constraints*. Each simple constraint represents the set of states defined by the conjunction of equalities and inequalities in it. For example, the final state of the Futurebus example is represented by the simple constraint consisting of one inequality $e \geq 2$. Instead of disjunctions of simple constraints, **BRAIN** uses sets of simple constraints. We say that a finite set D of constraints *symbolically represents* a set of states S if for every state s we have $s \in S$ if and only if for some $C \in D$ we have $s \models C$.

Backward reachability was chosen for a simple reason: symbolic application of a guarded assignment to a set of states represented by a formula $C(\mathcal{V})$ gives a quantified formula, while quantifiers arising from the backward application of a guarded assignment can be immediately eliminated. Let $C(\mathcal{V})$ be a formula specifying a set S of states and we have a transition t specified by a guarded assignment $G(\mathcal{V}) \rightarrow \mathcal{V} := \bar{u}(\mathcal{V})$. It is not hard to argue that the set $t(S)$ is defined by the formula

$$\exists \mathcal{V}' (C(\mathcal{V}') \wedge G(\mathcal{V}') \wedge \mathcal{V} = \bar{u}(\mathcal{V}')),$$

which is quantified even when C and G are quantifier-free. But the set $t^{-1}(S)$ is defined by the formula

$$\exists \mathcal{V}' (C(\mathcal{V}') \wedge G(\mathcal{V}) \wedge \mathcal{V}' = \bar{u}(\mathcal{V})),$$

which is equivalent to the following quantifier-free formula

$$C(\bar{u}(\mathcal{V})) \wedge G(\mathcal{V}).$$

Thus, it is much easier to symbolically compute $t^{-1}(S)$ than it is to compute $t(S)$.

5 The Reachability Algorithm

The backward reachability algorithm `LocalBackward` of BRAIN is implemented via a saturation algorithm given in Figure 2. This algorithm is taken from [8], where implementation of BRAIN is discussed in more detail.

The algorithm works with several sets of simple constraints, IS representing the set of initial states, FS representing the set of final states, $used$ and $unused$ representing intermediate states which arise from the backward reachability analysis. The algorithm is parametrized by a function `select` which selects a simple constraint in a set of simple constraints. The function `pdfn(A)` returns a set Q of simple constraints such that A is equivalent to $\bigvee_{C \in Q} C$.

At each step we select a simple constraint S , backward-apply all transitions to it, obtaining new simple constraints which are immediately checked for satisfiability. The satisfiable ones are added to the set $unused$. At each step we also perform an *entailment-check*: if a simple constraint C entails a simple constraint N , then N is removed from the search space.

The algorithm also checks whether any initial state satisfies a constraint C generated by the algorithm. If yes, then the set of states represented by C contains an initial state, hence some final state is reachable from this initial state. It is not hard to argue that the algorithm always terminates when there exists a final state reachable from an initial state, if the function `select` is fair. If no final state is reachable from an initial state, then the algorithm may either terminate due to entailment checks or not terminate at all. The algorithm is relatively simple because at each step we are dealing with a set of constraints of the same variables \mathcal{V} .

6 Algorithms over Simple Constraints

The backward reachability algorithm of BRAIN requires efficient algorithms for checking satisfiability and entailment of constraints. These algorithms are described in [8]. They are based on an algorithm [9] for building the basis of simple constraints similar to the algorithm of [1]. The algorithm computes the basis of a constraint consisting of non-decomposable solutions. The algorithm is *incremental*: if a new equation is added to the system the algorithm incrementally recomputes the basis. Incremental computation of the basis is very useful for checking intersection with the set of the initial states and also for deleting redundant inequalities or equalities.

7 Performance

Paper [8] contains experimental results comparing BRAIN with Action Language Verifier [2], HyTech [5], and DMC [4]. For integer-valued problems HyTech and DMC use relaxation, i.e., they solve real reachability problems instead of integer reachability problems. Therefore, they are correct only when they report non-reachability. The benchmarks are coming from specification of cache coherence

```

procedure LocalBackward
input: quantifier-free formulas  $In, Fin$ ,
        finite set of simple guarded assignments  $U$ 
output: “reachable” or “unreachable”
begin
   $IS := \text{pdnf}(In); FS := \text{pdnf}(Fin)$ 
  if there exist  $I \in IS, F \in FS$  such that  $\models \exists \mathcal{V}(I \wedge F)$  then return “reachable”
   $unused := FS; used := \emptyset$ 
  while  $unused \neq \emptyset$ 
     $S := \text{select}(unused)$ 
     $used := used \cup \{S\}; unused := unused - \{S\}$ 
    forall  $u \in U$ 
      (* backward application of  $u$  *)
       $N := S^{-u}$ 
      (* satisfiability-check for the new constraint  $N$  *)
      if  $\models \exists \mathcal{V}(N)$  then
        (* intersection-checks *)
        if there exists  $I \in IS$  such that  $\models \exists \mathcal{V}(N \wedge I)$  then return “reachable”
        (* entailment-checks *)
        if for all  $C \in used \cup unused$  we have  $\not\models \forall \mathcal{V}(N \supset C)$  then
           $unused = unused \cup \{N\}$ 
          forall  $C' \in used \cup unused$ 
            (* more entailment-checks *)
            if  $\models \forall \mathcal{V}(C' \supset N)$  then remove  $C'$  from  $used$  or  $unused$ 
    return “unreachable”
end

```

Fig. 2. Local backward reachability algorithm used in BRAIN

protocols, properties of Java programs, and some other applications and have been collected by Giorgio Delzanno.¹ Our results reported in [8] show that BRAIN is usually faster than HyTech and considerably faster than the other two systems, often by several orders of magnitude. BRAIN also consumes much less memory than HyTech and Action Language Verifier.

There are several problems which could only be solved by BRAIN, but not by any of the other systems. On some of these benchmarks, BRAIN made about 10^9 entailment checks within the overall running time of about 15 minutes.

However, we would like to note that all of these systems are on some benchmarks more powerful than BRAIN since they can use techniques such as widening or transitive closure which the current version of BRAIN does not have.

8 Availability

The system is available at <http://www.cs.man.ac.uk/~voronkov/BRAIN/>.

¹ See www.disi.unige.it/person/DelzannoG/.

References

- [1] F. Ajili and E. Contejean. Avoiding slack variables in the solving of linear diophantine equations and inequations. *Theoretical Computer Science*, 173(1):183–208, 1997. 492
- [2] T. Bultan. Action Language: a specification language for model checking reactive systems. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 335–344, Limerick, Ireland, 2000. ACM. 492
- [3] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999. 491
- [4] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001. 492
- [5] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hy-tech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997. 492
- [6] J. Rushby. Specification, proof checking and model checking for protocols and distributed systems with PVS, November 1997. Tutorial notes for FORTE/PSTV’97, <http://pvs.csl.sri.com/pvs/examples/forte97-tutorial/forte97.html>. 489
- [7] T. Rybina and A. Voronkov. A logical reconstruction of reachability. In *Submitted*, 2002. 489
- [8] T. Rybina and A. Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. In *Proceedings of CAV 2002*, 2002. To appear. 492, 493
- [9] A. Voronkov. An incremental algorithm for finding the basis of solutions to systems of linear Diophantine equations and inequations. unpublished, January 2002. 492

The Development Graph Manager MAYA

Serge Autexier¹, Dieter Hutter¹, Till Mossakowski², and Axel Schairer¹

¹ DKFI GmbH, Stuhlsatzenhausweg 3, D 66123 Saarbrücken
{autexier,hutter,schairer}@dfki.de

² FB 3, University of Bremen, P.O. Box 330 440, D 28334 Bremen
{till}@tzi.de

1 Overview

Formal methods are used in the software development process to increase the security and safety of software. The software systems as well as their requirement specifications are formalised in a textual manner in some specification language like CASL [3] or VSE-SL [10]. The specification languages provide constructs to structure the textual specifications to ease the reuse of components. Exploiting this structure, e.g. by identifying shared components in the system specification and the requirement specification, can result in a drastic reduction of the proof obligations, and hence of the development time which again reduces the overall project costs.

However, the logical formalisation of software systems is error-prone. Since even the verification of small-sized industrial developments requires several person months, specification errors revealed in late verification phases pose an incalculable risk for the overall project costs. An *evolutionary formal development* approach is absolutely indispensable. In all applications so far development steps turned out to be flawed and errors had to be corrected. The search for formally correct software and the corresponding proofs is more like a *formal reflection* of partial developments rather than just a way to assure and prove more or less evident facts.

The MAYA-system supports an evolutionary formal development since it allows users to specify and verify developments in a structured manner, incorporates a uniform mechanism for verification *in-the-large* to exploit the structure of the specification, and maintains the verification work already done when changing the specification. MAYA relies on development graphs as a uniform representation of structured specifications, which enables the use of various (structured) specification languages like CASL [3] and VSE-SL [10] to formalise the software development. To this end MAYA provides a generic interface to plug in additional parsers for the support of other specification languages. Moreover, MAYA allows the integration of different theorem provers to deal with the actual proof obligations arising from the specification, i.e. to perform verification *in-the-small*.

Textual specifications are translated into a structured logical representation called a *development graph* [1, 4], which is based on the notions of consequence relations and morphisms and makes arising proof obligations explicit. The user

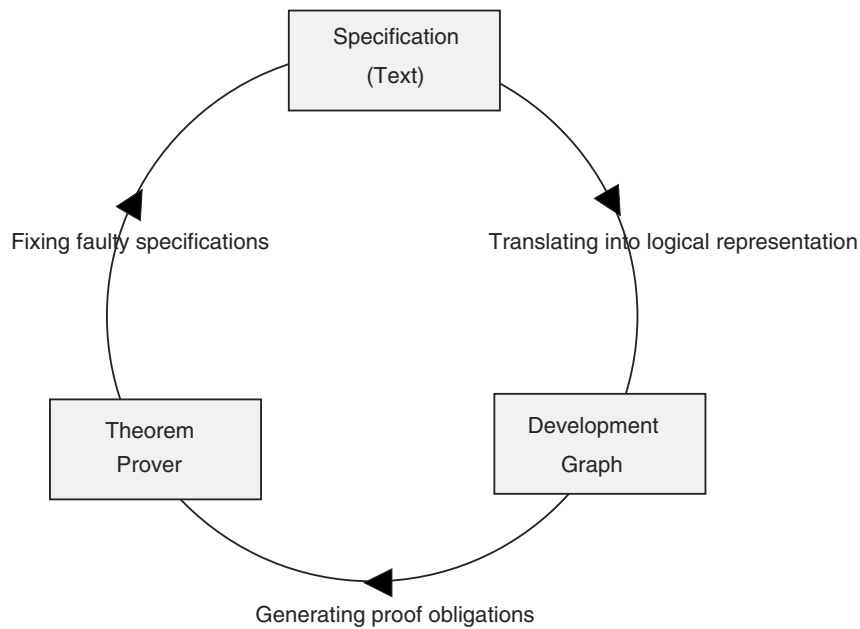


Fig. 1. Formal life cycle

can tackle these proof obligations with the help of theorem provers connected to MAYA like Isabelle [8] or INKA [5].

A failure to prove one of these obligations usually gives rise to modifications of the underlying specification (see Fig. 1). MAYA supports this evolutionary process as it calculates minimal changes to the logical representation readjusting it to a modified specification while preserving as much verification work as possible. If necessary it also adjusts the database of the interconnected theorem prover. Furthermore, MAYA communicates explicit information how the axiomatisation has changed and also makes available proofs of the same problem (invalidated by the changes) to allow for a reuse of proofs inside the theorem provers. In turn, information about a proof provided by the theorem provers is used to optimise the maintenance of the proof during the evolutionary development process.

2 From Textual to Logical Representation

The specification of a formal development in MAYA is always done in a textual way using specification languages like CASL or VSE-SL. MAYA incorporates parsers to translate such specifications into the MAYA-internal specification language DGRL (“Development Graph Representation Language”). While unstructured specifications are solely represented as a signature together with a set of logical formulas, the structuring operations of the specification languages (such as **then**, **and**, or **with** in CASL) are translated into the structure of a development graph. Each node of this graph corresponds to a theory. The axiomatisation

```

emacs: Ausr2011Amzommacs [21.1 (patch 14) "Giyahoga Valley" XEmacs Lucid] AMAST.casl
File Edit Hule Apps Options Buffers Tools Top <<< . >>> Bot Help
Open Find Save Print Cut Copy Paste Undo Redo Kill Undo Redo Help
-----
spec natlist =
{
  generated type nat ::= null | e(p:nat);
  var x,y,z:nat;
  op * : nat * nat -> nat, comm, assoc, unit e(null);
  op +(x:nat, y:nat):nat =
    y when x = null
    else e+(p(x), y);
  axiom +(x,y) = +(y,x);
  axiom +(x,(y,z)) = +(x,y).z;
}

then
{
  generated type natlist ::= nil
    | cons(set:nat, rest:natlist);
  var l1,l2:natlist;
  var n1,n2:nat;
  op app : natlist * natlist -> natlist, assoc, unit nil;
  axiom app(cons(n1,l1).l2) = cons(n1, app(l1,l2));
  op sdiast(n:nat, l:natlist):natlist =
    cons(n,nil) when l = nil
    else cons(set(l), sdiast(n,rest(l)));
  op delete_until(n:nat, l:natlist):natlist =
    nil when l = nil
    else rest(l) when n = set(l)
    else delete_until(n, rest(l));
}

-----
spec stack =
{
  sort elem;
}

then
{
  generated type stack ::= empty_stack
    | push(top:elem, pop:stack);
  op poprec(e:elem, s:stack):stack =
    empty_stack when s = empty_stack
    else pop(s) when e = top(s)
    else poprec(e, pop(s));
}

view viewit : stack to natlist =
  sort e elem |-> nat.
  stack |-> natlist.
  op poprec:elem * stack -> stack |-> delete_until.
  empty_stack:stack |-> nil.
  top: stack -> elem |-> fat.
  pop: stack -> stack |-> rest.
  push:elem * stack -> stack |-> cons.
end
-----
Hocorrv -----XEmacs: AMAST.casl (Fundamental PerDel)-----To

```

Fig. 2. Textual CASL-specification

of this theory is split into a local part which is attached to the node as a set of higher-order formulas and into global parts, denoted by ingoing definition links, which import the axiomatisation of other nodes via some consequence morphisms. While a so-called *local* link imports only the local part of the axiomatisation of the source node of a link, *global* links are used to import the entire axiomatisation of a source node (including all the imported axiomatisations of other nodes). In the same way local and global *theorem links* are used to postulate relations between nodes (see [1] for details). A global theorem link represents that all axioms defining the theory of the source node are conjectures in the theory of the target node, while local theorem links represent that only the local axioms of the source node are conjectures in the theory of the target node. Consider, as an example, the CASL-specification of lists of natural numbers and stacks over arbitrary elements (cf. Fig. 2). The *view* postulates that lists over natural numbers are indeed stacks. The specifications are translated into the development graph viewed in the upper left part of Fig. 3. The resulting development graph is structurally more verbose than the original textual specification structure because of the different definitions of scope and visibility for signature symbols: The visibility rules for the different CASL structuring operations are encoded in the uniform structuring operation (definition links) of the development graph. The left subgraph results from the specification of stacks over arbitrary elements, while the right subgraph corresponds to the lists over natural numbers. The *view* is encoded by a global theorem link from the top node of the left subgraph to the top node of the right subgraph.

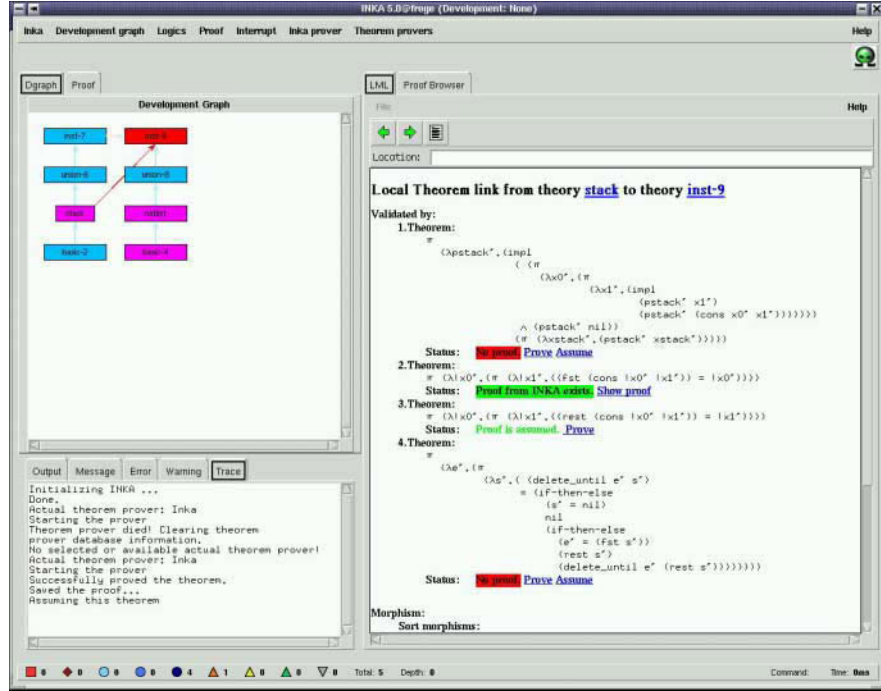


Fig. 3. Bookkeeping the status of proof obligations in MAYA

3 Verification In-the-Large

The development graph is the central data-structure to store and maintain the formal (structured) specification, the arising proof obligations and the status of the corresponding verification effort (proofs) during a formal development.

MAYA distinguishes between proof obligations postulating properties between different theories, like the notion of *view* in CASL or *satisfies* in VSE-SL, and lemmata postulated within a single theory, e.g. with the `%implies` annotation in CASL. As theories correspond to subgraphs within the development graph, a relation between different theories, represented by a global theorem link, corresponds to a relation between two subgraphs. Each change of these subgraphs can affect this relation and would invalidate previous proofs of this relation. Therefore, MAYA decomposes relations between different theories into individual relations between the local axiomatisation of a node and a theory (denoted by a local theorem link). Each of these relations decomposes again into a set of proof obligations postulating that each local axiom of the node is a theorem in the target theory with respect to the morphism attached to the link. In the running example the global theorem link between the top nodes of both subgraphs is decomposed into a set of local theorem links from each node of the left subgraph

to the top node of the right subgraph (cf. Fig. 3). The proof obligations of such a local theorem link are viewed in the right-hand side of Fig. 3.

While definition links establish relations between theories, theorem links denote lemmata postulated about such relations. Thus, the reachability between two nodes establishes a formal relation between the connected nodes (i.e. the theory of the source node is part of the theory of the target node wrt. the morphisms attached to the connecting links). MAYA uses this property to prove relations between theories by searching for paths between the corresponding nodes (instead of decomposing the corresponding proof obligation in the first place).

4 Verification In-the-Small

When verifying a local theorem link or proving speculated lemmata, the conjectures have to be tackled by some interconnected theorem prover. In both cases the proofs are done *within* the theory of a specific node. Thus, conceptually each node may include its own theorem prover which is provided with the local axioms of the node and all axioms imported via incoming definition links. In principle, there is a large scale of integration types. The tightest integration consists of having a theorem prover for each node wrt. which theory conjectures must be proven, and the theorem prover returns a proof object generated during the proof of a conjecture. Those are stored together with the conjecture and can be used by MAYA to establish the validity of the conjecture if the specification is changed. The loosest integration consists in having a single generic theorem prover, which is requested to prove a conjecture within some theory and is provided with the axiomatisation of this theory. The theorem prover only returns whether it could prove a conjecture or not, without any information about axioms used during the proof. For a detailed discussion of the advantages and drawbacks of the different integration scenarios see [2].

Currently, MAYA supports two integration types: One where information about used axioms is provided by the theorem prover, and one where no such information is provided. In the first case, MAYA stores the proof information and the axioms used during the proof (cf. status of theorem 2 in Fig. 3). In the second case, MAYA assumes there is a proof for the proof obligation (cf. status of theorem 3 in Fig. 3), as there is no information about the proof. In both scenarios, MAYA makes use of generic theorem provers which are provided with the axiomatisation of the current theory. Currently MAYA provides all axioms and lemmata located at theories that are imported from the actual theory by definition links to the prover. Switching between different proof obligations may cause a change of the current underlying theory and thus a change of the underlying axiomatisation. MAYA provides a generic interface to plug in theorem provers (based on an XML-RPC protocol) that allows for an incremental update of the database of the prover.

5 Evolution of Developments

Changes of specifications are done inside the textual representation. Parsing a modified specification results in a modified DGRL-specification. In order to support a management of change, MAYA computes the differences of both DGRL-specifications and compiles them into a sequence of basic operations in order to transform the development graph corresponding to the original DGRL-specification to a new one corresponding to the modified DGRL-specification. Examples of such basic operations are the insertion or deletion of a node or a link, the change of the annotated morphism of a link, or the change of the local axiomatisation of a node. As there is no optimal solution to the problem of computing differences between two specifications, MAYA uses heuristics based on names and types of individual objects to guide the process of mapping corresponding parts of old and new specification. Since the differences of two specifications are computed on the basis of the internal DGRL-representation, new specification languages can easily be incorporated into MAYA by providing a parser for this language and a translator into DGRL.

The development graph is always synthesised or manipulated with the help of the previously mentioned basic operations (insertion/deletion/change of nodes/links/axiomatisation) and MAYA incorporates sophisticated techniques to analyse how these operations will affect proof obligations or proofs stored within the development graph. They incorporate a notion of monotonicity of theories and morphisms, and take into account the sequence in which objects are inserted into the development graph. Furthermore, the information about the decomposition and subsumption of global theorem links obtained during the verification *in-the-large* is explicitly maintained and exploited to adjust them once the development graph is altered. Finally, the knowledge about proofs, e.g. the used axioms, provided by the interconnected theorem provers during the verification *in-the-small* is used to preserve or invalidate the proofs.

6 Conclusion

The MAYA-system is mostly implemented in Common Lisp while parts of the GUI, shared with the OMEGA-system [9], are written in Mozart. The CASL-parser is provided by the CoFI-group in Bremen. The MAYA-system is available from the MAYA-web-page at www.dfki.de/~inka/maya.html.

Future extensions of the system will include a treatment of hiding [7], a uniform treatment of different logics based on the notion of heterogenous development graphs [6], and the maintenance of theory-specific control information for theorem provers. The latter comprises a management for building up the database of theorem provers by demand rather than providing all available axioms and lemmata at once as well as the management of meta-level information, like tactics or proof plans, inside MAYA.

References

- [1] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an evolutionary formal software-development using CASL. In *Recent Developments in Algebraic Development Techniques, WADT'99, Bonas, France*, Springer LNCS 1827, 2000. 495, 497
- [2] S. Autexier, T. Mossakowski. Integrating HOLCASL into the Development Graph Manager MAYA. In A. Armando (Ed.) *Frontiers of Combining Systems (FroCoS'02)*, Santa Margherita Ligure, Italy, Springer LNAI, April, 2002. 499
- [3] CoFI Language Design Task Group. *The Common Algebraic Specification Language (CASL) – Summary*, Version 1.0 and additional Note S-9 on Semantics, available from <http://www.cofi.info>, 1998. 495
- [4] D. Hutter. Management of change in verification systems. In *Proceedings 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, pages 23–34. IEEE Computer Society, 2000. 495
- [5] S. Autexier, D. Hutter, H. Mantel, A. Schairer: System Description: INKA 5.0 - A Logic Voyager. In H. Ganzinger, *CADE-16*, Springer, LNAI 1632, 1999. 496
- [6] T. Mossakowski. Heterogeneous development graphs and heterogeneous borrowing. In M. Nielsen (Ed.) *Foundations of Software Science and Computation Structures (FOSSACS02)*, Grenoble, France, Springer LNCS, April, 2002. 500
- [7] T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In A. Konermann, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE2001)*. Springer, LNCS 2029, 2001. 500
- [8] L. C. Paulson. *Isabelle - A Generic Theorem Prover*, Springer LNCS 828, 1994. 496
- [9] J. Siekmann et al. LOUI : Lovely OMEGA user interface. *Formal Aspects of Computing*, 3(11):326-342, 1999. 500
- [10] D. Hutter et. al. Verification Support Environment (VSE), *Journal of High Integrity Systems*, Vol. 1, pages 523–530, 1996. 495

Author Index

- Aceto, Luca 239
Adélaïde, Michaël 132
Agha, Gul 223
Alpuente, María 117
Aspinall, David 1
Autexier, Serge 495
Baldan, Paolo 254
Barthe, Gilles 41
Berg, Joachim van den 304
Borgström, Johannes 287
Bossi, Annalisa 271
Bracciali, Andrea 254
Breunese, Cees-Bart 304
Bruni, Roberto 254
Courtieu, Pierre 41
Devillers, Raymond 192
Dong, Yifei 147
Dufay, Guillaume 41
Escobar, Santiago 117
Ésik, Zoltán 239
Fiadeiro, José Luiz 75, 426
Focardi, Riccardo 271
Giegerich, Robert 349
Guo, Guangyuan 178
Haack, Christian 83
Haneberg, Dominik 319
Hill, Patricia M. 380
Hornus, Samuel 163
Howard, Brian 83
Huisman, Marieke 334
Hutter, Dieter 441, 495
Ingólfssdóttir, Anna 239
Jacobs, Bart 304
Janicki, Ryszard 178
King, Andy 365
Klaudel, Hanna 192
Koutny, Maciej 192
Lopes, Antónia 426
Lucas, Salvador 117
Meyer, Carsten 349
Miller, Dale 60
Mossakowski, Till 99, 495
Mosses, Peter D. 21
Nestmann, Uwe 287
Pavlovic, Dusko 411
Piazza, Carla 271
Pommereau, Franck 192
Ramakrishnan, C.R. 147
Reif, Wolfgang 319
Reus, Bernhard 473
Rossi, Sabina 271
Roux, Olivier 132
Rybina, Tatiana 489
Sannella, Donald 1
Sarna-Starosta, Beata 147
Schairer, Axel 441, 495
Schnoebelen, Philippe 163
Schröder, Lutz 99
Simon, Axel 365
Skoglund, Mats 457
Smith, Douglas R. 411
Smolka, Scott A. 147
Sousa, Simão Melo de 41
Spoto, Fausto 380
Stell, John G. 396
Stenzel, Kurt 319
Stoughton, Allen 83
Thati, Prasanna 223
Trentelman, Kerry 334
Ulidowski, Irek 208
Voronkov, Andrei 489
Walukiewicz, Igor 15
Wells, Joe B. 83
Ziaei, Reza 223