Lecture Notes in Computer Science 1742
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

P.S. Thiagarajan R. Yap (Eds.)

Advances in Computing Science – ASIAN'99

5th Asian Computing Science Conference
Phuket, Thailand, December 10-12,1999
Proceedings

Springer

Volume Editors

P.S. Thiagarajan
Chennai Mathematical Institute
92 G.N. Chetty Road, T. Nagar, Chennai 600 017, India
E-mail: pst@smi.ernet.in

Roland Yap
National University of Singapore, School of Computing
Lower Kent Ridge Road, Singapore 119260
E-mail: ryap@comp.nus.edu.sg

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Advances in computing science - ASIAN '99 : proceedings / 5th Asian Computing Science Conference, Phuket, Thailand, December 10 - 12, 1999. P. S. Thiagarajan ; R. Yap (ed.) - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1742)
ISBN 3-540-66856-X

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

ISSN 0302-9743
ISBN 3-540-66856-X Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10749973 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

The Asian Computing Science Conference (ASIAN) series was initiated in 1995 to provide a forum for researchers in computer science from the Asian region to meet and to promote interaction with researchers from other regions. The previous four conferences were held, respectively, in Bangkok, Singapore, Kathmandu, and Manila. The proceedings were published in the Lecture Notes in Computer Science Series of Springer-Verlag.

This year's conference (ASIAN'99) attracted 114 submissions from which 28 papers were selected through an electronic PC meeting. In addition, 11 papers were selected for shorter presentations at the poster sessions.

The themes for this year's conference were announced to be:

- Embedded and Real-Time Systems
- Formal Reasoning and Verification
- Distributed and Mobile Computing

The key note speaker for ASIAN'99 is Amir Pnueli (Weizmann Institute, Israel) and the invited speakers are Nicolas Halbwachs (VERIMAG, CNRS, France) and Krishna Palem (The Georgia Institute of Technology and Courant Institute, New York University, USA). We thank them for accepting our invitation.

This year's conference is being sponsored by the Asian Institute of Technology (Thailand), INRIA (France), the National University of Singapore (Singapore), and UNU/IIST (Macau). We thank all these institutions for their continued support of the ASIAN series.

This year's conference will be held in Phuket, Thailand. We are much obliged to the Prince of Songkhla University for providing the conference venue and to Rattana Wetprasit for making the local arrangements.

We also wish to thank the PC members and the large number of referees for the substantial work put in by them in assessing the submitted papers.

Finally, it is a pleasure to acknowledge the friendly and efficient support provided by Alfred Hofmann and his team at Springer-Verlag in bringing out this volume.

October 1999

P. S. Thiagarajan
Roland Yap

Program Committee

Gerard Berry (ENSMP & INRIA, France)
Phan Min Dung (AIT, Thailand)
Cedric Fournet (Microsoft Research, UK)
Kokichi Futatsugi (JAIST, Japan)
Shigeki Goto (Waseda U., Japan)
Dang Van Hung (UNU/IIST, Macau)
Vinod Kathail (HP, USA)
Michael Lyu (CUHK, Hong Kong)
Yen-Jen Oyang (NTU, Taiwan)
Frank Pfenning (CMU, USA)
Sanjiva Prasad (IIT, Delhi, India)
Abdul Sattar (Griffith U., Australia)
R.K. Shyamasundar (TIFR Bombay, India)
Aravind Srinivasan (Bell Labs, USA)
Peter Stuckey (U. of Melbourne, Australia)
P.S. Thiagarajan (**co-chair**) (CMI, India)
Farn Wang (IIS, Academia Sinica, Taiwan)
Limsoon Wong (KRDL, Singapore)
Roland Yap (**co-chair**) (NUS, Singapore)

General Chair

Kanchana Kanchanasut (Asian Institute of Technology, Thailand)

Local Chair

Rattana Wetprasit (Prince of Sonkhla University, Thailand)

Steering Committee

Shigeki Goto (Waseda U., Japan)
Joxan Jaffar (NUS, Singapore)
Gilles Kahn (INRIA, France)
Kanchana Kanchanasut (AIT, Thailand)
Jean-Jacques Levy (INRIA, France)
R.K. Shyamasundar (TIFR Bombay, India)
Kazunori Ueda (Waseda U., Japan)
Zhou Chaochen (UNU/IIST, Macau)

Referees

S. Aditya
G. Antoniou
M. Asaka
R. Ayani
L. Birkedal
A. Bouali
J. Bowen
J. Bradfield
S. Bressan
A. Brogi
S. Brookes
C. E. Brown
K. Bruce
R. Bryant
L. Cardelli
K. S. Cheng
W. N. Chin
T-R. Chuang
C. Colby
I. K. Cuhk
D. D'Souza
B. Davey
R. Davies
A. Dawar
G. DeMichelis
A. Degtyarev
R. Deng
Y. Deswarte
A. A. Diwan
G. Dobbie
G. Dong
M. Dorigo
D. Dubhashi
W. W Fai
S. Finne
S. Ganguly
H. Ganzinger
N. Garg
C. George
G. Gonthier
A. Gordon
D. D. Hanh
R. Hariharan
J. Harland
D. Harmanec
N. Heintze
P. A. Heng
J. G. Henriksen
M. Henz
E. Horita
P-A. Hsiung
Z. Huang
C. C. Hung
S. Ichikawa
S. Iida
L. Jacob
R. Jagadeesan
P. Jayanti
S. Jha
H. Jifeng
M. Kankanhalli
R. Kawahara
E. Kazmierczak
W. Ke
U. Kohlenbach
A. Konagaya
G. Konjevod
M. Koutny
K. Kumaran
B. A. Kuperman
M. Kwiatkowska
Y. C. Kwong
J. Lee
P. Lee
P. Leong
R. Letz
A. Leung
H-f. Leung
L. Libkin
A. Lim
T. Ling
K. Lodaya
P. MacKenzie
P. Madhusudan
M. Maher
O. Maler
K. Marriott
M. Matsumoto
T. Y. Meng
J. Moore
R. Moore
A. Mori
M. Mukund
K. Nakano
K. NarayanKumar
U. Nestmann
P. Niebert
R. O'Callahan
K. Ogata
H. Ohta
C-H. L. Ong
P. K. Pandya
H. H. Pang
B. Pientka
B. Pierce
J. Polakow
B. Prabhakaran
P. Quaglia
J. Radhakrishnan
N. Raja
V. Raman
R. Ramanujam
S. Ramesh
R. Ramirez
A. Ranade
J. Rathke
B. R. Rau
J. Rehof
M. Saeki
I. Saniee
A. Santosa
H. Saran
M. Schlansker
C. Schurmann
A. Seth
N. Shinagawa
G. Shroff
J. Sifakis
G. Sivakumar
M. Sohoni
H. Sondergaard
L. Sonenberg

D. J. Song
L. Sterling
H. Suda
H. Sun
S. Suryanata
D. Syme
N. Takahashi
V. Tam
H. Tamaki
H. Tamura
K. Tan
K-L. Tan
K. Thirunarayan
J. Thornton
R. Topor
S. Tripakis
K. Ueda
N. Umeda
M. Veanes
S. Vishwanathan
R. Voicu
A. Voronkov
P. Wadler
C-Y. Wang
D-W. Wang
H. Wang
D. S. Warren
K. Watkins
J-J. Wu
Y. L. Wu
C-Z. Yang
Y. Zhangq

Table of Contents

Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing[⋆]

Nicolas Halbwachs and Pascal Raymond

Vérimag[⋆⋆], Grenoble – France
{Nicolas.Halbwachs,Pascal.Raymond}@imag.fr

Abstract. This paper surveys the techniques and tools developped for the validation of reactive systems described in the synchronous data-flow language LUSTRE [HCRP91]. These techniques are based on the specification of safety properties, by means of *synchronous observers*. The model-checker LESAR [RHR91] takes a LUSTRE program, and two observers — respectively describing the expected properties of the program, and the assumptions about the system environment under which these properties are intended to hold —, and performs the verification on a finite state (Boolean) abstraction of the system. Recent work concerns extensions towards simple numerical aspects, which are ignored in the basic tool. Provided with the same kind of observers, the tool LURETTE [RWNH98] is able to automatically generate test sequences satisfying the environment assumptions, and to run the test while checking the satisfaction of the specified properties.

1 Introduction

Synchronous languages [Hal93, BG92, LGLL91, HCRP91] have been proposed to design so-called "reactive systems", which are systems that maintain a permanent interaction with a physical environment. In this area, system reliability, and therefore design validation, are particularly important goals, since most reactive systems are safety critical. As a consequence, many validation tools have been proposed, which are dedicated to deal with systems described by means of synchronous languages. These tools either concern automatic verification [LDBL93, DR94, JPV95, Bou98, RHR91], formal proof [BCDP99], or program testing [BORZ98, RWNH98, MHMM95, Mar98].

As a matter of fact the validation of synchronous programs, on one hand raises specific problems — like taking into account known properties of the environment — and on the other hand allows the application of specific techniques — since the programs to be validated are *deterministic* systems with inputs, in contrast with classical concurrent processes, which are generally modelled as non-deterministic and closed systems. Both for formal verification and for testing, the user has to specify:

[⋆] This work was partially supported by the ESPRIT-LTR project "SYRF".

[⋆⋆] Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG associated with IMAG.

1. the intended behavior of the program under validation, which may be more or less precisely defined. In particular, it may consist of a set of properties, and, for the kind of considered systems, critical properties are most of the time *safety properties.*
2. the assumptions about the environment under which the properties specified in (1) are intended to hold. These assumptions are generally safety properties, too.

In synchronous programming, a convenient way of specifying such safety properties is to use "*synchronous observers*" [HLR93], which are programs observing the inputs and the outputs of the program under validation, and detect the violation of the property. Once these observers have been written, automatic validation tools can use them for

formal verification: One can verify, by model-checking, that for each input flow satisfying the assumption, the corresponding output flow satisfy the property. In general, this verification is performed on a finite-state abstraction of the program under verification.

automatic testing: The assumption observer is used to generate realistic test sequences, which are provided to the program; the property observer is used as an "oracle" determining whether each test sequence "passes" or "fails".

In this paper, we present these approaches in the context of the declarative language LUSTRE [HCRP91]. A model-checker for LUSTRE, called LESAR [RHR91], has been developped for long, and extended towards dealing with simple numerical properties. Two testing tools, LUTESS [BORZ98] and LURETTE [RWNH98] are also available; here, we focus on LURETTE, which has some numerical capabilities.

2 Synchronous Observers in LUSTRE

2.1 Overview of Lustre

Let us first recall, in a simplified way, the principles of the language LUSTRE: A LUSTRE program operates on *flows* of values. Any variable (or expression) x represents a flow, i.e., an infinite sequence $(x_0, x_1, \ldots, x_n, \ldots)$ of values. A program is intended to have a cyclic behavior, and x_n is the value of x at the nth cycle of the execution. A program computes output flows from input flows. Output (and possibly local) flows are defined by means of equations (in the mathematical sense), an equation "x=e" meaning "$\forall n, x_n = e_n$". So, an equation can be understood as a temporal invariant. LUSTRE operators operate globally on flows: for instance, "x+y" is the flow $(x_0+y_0, x_1+y_1, \ldots, x_n+y_n, \ldots)$. In addition to usual arithmetic, Boolean, conditional operators — extended pointwise to flows as just shown — we will consider only two temporal operators:

– the operator "pre" ("*previous*") gives access to the previous value of its argument: "pre(x)" is the flow $(nil, x_0, \ldots, x_{n-1}, \ldots)$, where the very first value "*nil*" is an undefined ("non initialized") value.

– the operator "->" ("followed by") is used to define initial values: "x -> y" is the flow $(x_0, y_1, \ldots, y_n, \ldots)$, initially equal to x, and then equal to y forever.

As a very simple example, the program shown below is a counter of "events": It takes as inputs two Boolean flows "evt" (true whenever the counted "event" occurs), and "reset" (true whenever the counter should be reinitialized), and returns the number of occurrences of "events" since the last "reset". Once declared, such a "node" can be used anywhere in a program, as a user-defined operator. For instance, our counter can be used to generate an event "minute" every 60 "second", by counting "second" modulo 60.

```
node Count(evt, reset: bool)
    returns(count: int);
let
    count = if (true -> reset) then 0
            else if evt then pre(count)+1
            else pre(count)
tel
```

```
mod60 = Count(second, pre(mod60=59));
minute = (mod60 = 0);
```

2.2 Synchronous Observers

Now, an observer in LUSTRE will be a node taking as inputs all the flows relevant to the safety property to be specified, and computing a single Boolean flow, say "ok", which is true as long as the observed flows satisfy the property.

For instance, let us write an observer checking that each occurrence of an event "danger" is followed by an "alarm" before the next occurrence of the event "deadline". It uses a local variable "wait", triggered by "danger" and reset by "alarm", and the property will be violated whenever "deadline" occurs when "wait" is on.

```
node Property(danger, alarm, deadline: bool)
    returns (ok: bool);
var wait: bool;
let
    wait = if alarm then false
           else if danger then true
           else (false -> pre(wait));
    ok = not(deadline and wait);
tel
```

Assume that the above property is intended to hold about a system S, computing "danger" and "alarm", while "deadline" comes from the environment. Obviously, except if S emits "alarm" simultaneously with each "danger", it cannot fulfill the property without any knowledge about "deadline". Now, assume we know that "deadline" never occurs earlier than two cycles after "danger".

```
node Assumption(danger, deadline: bool)
    returns (ok: bool);
let  ok = not deadline or
             (true -> pre(not danger and
                 (true -> pre(not danger))));
tel
```

This assumption can also be expressed by an observer.

```
(danger, alarm, ...) = S(deadline, ...);
realistic = Assumption(danger, deadline);
correct = Property(danger, alarm, deadline);
```

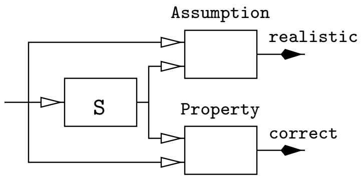


Fig. 1. Validation Program

2.3 Validation Program

Now we are left with 3 programs: the program S under validation, and its two observers, Property and Assumption. We can compose them in parallel, in a surrounding program called "Validation Program" (see Fig.1). Our verification problem comes down to showing that, whatever be the inputs to the validation program, either the output "correct" is always true, or the output "realistic" is sometimes false. The advantages of using synchronous observers for specification have been pointed out:

– there is no need to learn and use a different language for specifying than for programming.
– observers are *executable*; one can test them to get convinced that the specified properties are the desired ones.

Notice that synchronous observers are just a special case of the general technique [VW86] consisting in describing the negation of the property by an automaton (generally, a Büchi automaton), and showing, by performing a synchronous product of this automaton and the program, that no trace of the program is accepted by the automaton. The point is that, in synchronous languages, the synchronous product is the normal parallel composition, so this technique can be applied within the programming language.

3 Model-Checking

3.1 Lustre Programs as State Machines

Of course, a Lustre program can be viewed as a transition system. All operators, except pre and ->, are purely combinational, i.e., don't use the notion of *state*. The result of a -> operator depends on whether the execution is in its first cycle or not: let *init* be an auxiliary Boolean state variable, which is initially true, and then always false. The result of a pre operator is the value previously taken by its argument, so each pre operator has an associated state variable. All these state variables define the state of the program. Of course, programs that have only Boolean variables have finitely many states and can be fully verified by model-checking [QS82, CES86, BCM$^+$90, CBM89]: when the program under verification and both of its observers are purely Boolean, one can traverse the finite set of states of the validation program. Only states reached from the initial state without falsifying the output “realistic” are considered, and in each reached state, one check that, for each input, either “realistic” is false, or “correct” is true. This can be done either enumeratively (i.e., considering each state in turn) or symbolically, by considering sets of states as Boolean formulas.

3.2 Lustre Programs as Interpreted Automata

Programs with numerical variables can be partially verified, using a similar approach. We consider such a program as an intepreted automaton: the states of the automaton are defined by the values of the Boolean state variables, as above. The associated interpretation deals with the numerical part: conditions and actions on numerical variables are associated with the transitions of the automaton. An example of such an interpreted automaton will be shown in Section 4. If it happens that a property can be proved on the (finite) control part of the automaton, then it is satisfied by the complete program. Otherwise, the result is unconclusive.

3.3 LESAR

Lesar is a verification tool dedicated to Lustre programs. It performs the kind of verification described above, by traversing the set of control states of a validation program, either enumeratively of symbolically. More precisely, it restricts its search to the part of the program that can influence the satisfaction of the property. This part, sometimes called the *cone of influence*, can be easily determined, because of the declarative nature of the language: all dependences between variables are explicit. This is an important feature, since experience shows that, in many practical cases, the addressed property only concerns a very small part of a program: in such a case, Lesar may be able to verify the property, even if the whole state space of the program could not be built.

4 Towards Numerical Properties

Only properties that depend only on the control part of the program can be verified by model checking. The reason is that Lesar can consider as reachable some control states that are in fact unreachable because of the numerical interpretation, which is ignored during the state space traversal: some transitions are considered feasible, while being forbidden by their numerical guards. Let us illustrate this phenomenon on a very simple example, extracted from a subway speed regulation system:

A train detects beacons placed along the track, and receives a signal broadcast each second by a central clock. Ideally, it should encounter one beacon each second, but, to avoid shaking, the regulation system applies a hysteresis as follows: let #b and #s be, respectively, the current numbers of encountered beacons

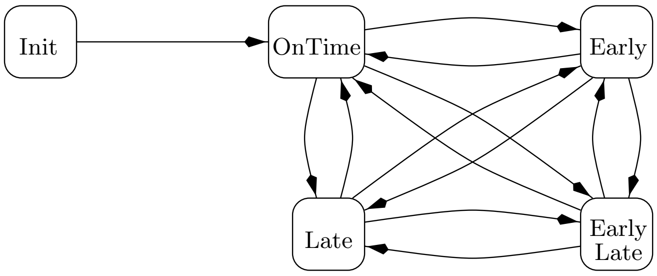


Fig. 2. Interpreted automaton of the subway example

and of elapsed seconds. Whenever $\#\mathsf{b} - \#\mathsf{s}$ becomes greater 10, the train is considered early, until $\#\mathsf{b} - \#\mathsf{s}$ becomes negative. Symmetrically, whenever $\#\mathsf{b} - \#\mathsf{s}$ becomes smaller than -10, the train is considered late, until $\#\mathsf{b} - \#\mathsf{s}$ becomes positive. We only consider the part of the system which determines whether the train is early of late. In LUSTRE, the corresponding program fragment could be:

```
diff =  0 -> if second and not beacon then pre(diff)–1
             else if beacon and not second then pre(diff)+1
             else pre(diff);
early = false -> if diff > 10 then true
                 else if diff < 0 then false
                 else pre(early);
late =  false -> if diff < –10 then true
                 else if diff > 0 then false
                 else pre(late);
```

This program has 3 Boolean state variables: the auxiliary variable *init* (initially true, and then false forever) and the variables storing the previous values of early and late. The corresponding interpreted automaton has the control structure shown by Fig 2, and, for instance, the transitions sourced in the state "OnTime" are guarded as follows:

g_1: $\mathsf{diff} > 10 \wedge \mathsf{diff} \geq -10 \;\rightarrow\;$ Early g_3: $\mathsf{diff} > 10 \wedge \mathsf{diff} < -10 \;\rightarrow\;$ EarlyLate
g_2: $\mathsf{diff} \leq 10 \wedge \mathsf{diff} < -10 \;\rightarrow\;$ Late g_4: $\mathsf{diff} \leq 10 \wedge \mathsf{diff} \geq -10 \;\rightarrow\;$ OnTime

Without any knowledge about numerical guards, the model-checker does not know that some of these guards (g_1 and g_2) can be simplified, nor that one of them (g_3) is unsatisfiable. This is why the state "EarlyLate" is considered reachable.

A transition the guard of which is numerically unsatisfiable will be called *statically unfeasible*. In our example, if we remove statically unfeasible transitions, we get the automaton of Fig. 3, where the state "EarlyLate" is no longer reachable. A simple way of improving the power of a model-checker is to provide

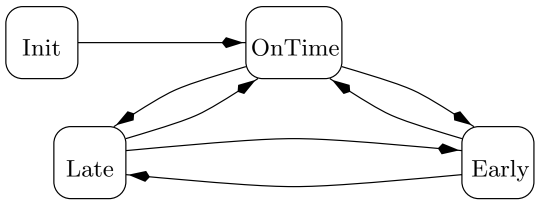


Fig. 3. The subway example without statically unfeasible transitions

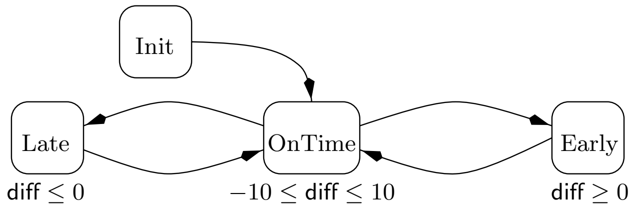


Fig. 4. The subway example without dynamically unfeasible transitions

it with the ability of detecting statically unfeasible transitions, in some simple cases. For instance, unfeasibility of guards made of *linear relations* is easy to decide[1].

This is why LESAR has been extended with such a decision procedure in linear algebra: when a state violating the property is reached by the standard model-checking algorithm, the tool can look, along the paths leading to this state, for transitions guarded by unfeasible linear guards. If all such "bad" paths can be cut, the "bad" state is no longer considered reachable. This very partial improvement significantly increases the number of practical cases where the verification succeeds.

Of course, we are not always able to detect statically unfeasible transitions. Moreover, some transitions are unfeasible because of the dynamic behavior of numerical variables. For instance, in the automaton of Fig. 3, there are direct transitions from state "Early" to state "Late" and conversely. Now, these transitions are clearly impossible, since diff varies of at most 1 at each cycle, and cannot jump from being ≥ 0 in state "Early" to becoming < -10 in state "Late". Such transitions are called *dynamically unfeasible.* Detecting dynamically unfeasible transitions is much more difficult. We experiment "linear relation analysis" [HPR97] — an application of abstract interpretation — to synthesize invariant linear relations in each state of the automaton. If the guard of a transition is not satisfiable within the invariant of its source state, then the transition

[1] at least for rational solutions; but since unfeasibility in rational numbers implies unfeasibility in integers, such an approximate decision is still conservative.

is unfeasible. In our example, we get the invariants shown in Fig. 4, which allow us to remove all unfeasible transitions.

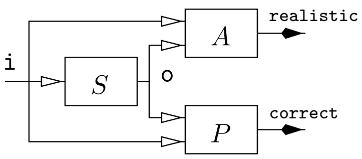
5 Automatic Testing

In spite of the progress of formal verification, testing is and will remain an important validation technique. On one hand, the verification of too complex systems — with too complex state space, or important numerical aspects — will remain unfeasible. On the other hand, some validation problems are out of the scope of formal verification: it is the case when parts of the program cannot be formally described, because they are unknown or written in low level languages; it is also the case when one wants to validate the final system within its actual environment. So, verification and testing should be considered as complementary techniques. Moreover, testing techniques and tools should be mainly devoted to cases where verification either fails or does not apply. This is why we are especially interested in techniques that cope with numerical systems, that don't need a formal description of the system under test (black box testing), and the cost of which doesn't depend on the internal complexity of the tested system.

Intensive testing requires automation, since producing huge test sets by hand is extremely expensive and error-prone. Now, it appears that the prerequisite for automatic generation of test sets is the same as for verification: an automatic tester will need a formal description of both the environment — to generate only realistic test cases — and the system under test — to provide an "oracle" deciding whether each test passes or fails. In section 2, we proposed the use of synchronous observers for these formal descriptions. In the Lurette [RWNH98] and Lutess [BORZ98] tools, such observers are used to automatically generate and run test sequences. In this section, we explain the principles of this generation.

The specific feature of reactive systems is, of course, that they run in closed loop with their environment. In particular, they are often intended to *control* their environment. This means that the current input (from the environment) may depend on the past outputs (from the system). In other words, the realism of an input sequence does not make sense independently of the corresponding output sequence, computed by the system under test. This is why, in our approach, test sequences are generated on the fly, as they are submitted to the system under test.
More precisely, we assume that the following components are available:

- an executable version of the system under test, say S. We only need to be able to run it, step by step.
- The observers A and P, respectively describing the assumptions about the environment and the properties to be checked during the test.



Moreover, the output "realistic" of the observer A is required *not* to depend instantaneously of the outputs "o" of S. Since "o" is supposed to be computed from the current input "i", it would be a kind of causality loop that the realism of "i" depend on "o".

Basically, the tester only needs to know the source code of the observer A, and to be able to run the system S and the observer P, step by step. It considers, first, the initial state of A: in this state, the LUSTRE code of A can be simplified, by replacing each expression "e_1-> e_2" by "e_1", and each expression "pre(e)" by "*nil*". After this simplification, the result "realistic" is a combinational expression of the input "i", say "b(i)". The satisfaction of the Boolean formula b(i) can be viewed as a constraint on the initial inputs to the system. A constraint solver — which will be detailed below — is used to randomly select an input vector i_0 satisfying this constraint. Now, S is run for a step on i_0, producing the output vector o_0 (and changing its internal state). Knowing both i_0 and o_0, one can run A and P for a step, to make them change their internal state, and to get the oracle "correct" output by P. The LUSTRE code of A can be simplified according to its new state, providing a new constraint on "i". The same process can be repeated as long as the test passes (i.e., P returns "correct = *true*"), or for a given number of steps.

The considered tools mainly differ in the selection of input vectors satisfying a given constraint. In LUTESS [BORZ98], one consider only purely Boolean observers. A constraint is then a purely Boolean formula, which is represented by a Binary Decision Diagram. A correct selection corresponds to a path leading to a "true" leaf in this BDD. The tool is able to perform such a selection, either using an equiprobable strategy, or taking into account user-given directives. LURETTE [RWNH98] is able to solve constraints that are Boolean expressions involving Boolean inputs and *linear relations* on numerical inputs.

Example: Let us illustrate the generation process on a very simple example. Assume S is intended to regulate a physical value u, by constraining its second derivative. Initially, both u and its derivative are known to be 0. Then, the second derivative of u will be in an interval $[-\delta, +\delta]$ around the (previous) output x of S. An observer of this behavior can be written as follows:

```
node A (u, x: real) returns (realistic: bool);
var dudt, d2udt2: real;
let
    dudt = 0 ->(u - pre(u));
    d2udt2 = dudt - pre(dudt);
    realistic = (u=0) -> ((pre(x) - delta <= d2udt2)
                        and (d2udt2 <= pre(x) + delta));
tel
```

At the first cycle, the code of A is simplified to

```
dudt = 0; d2udt2 = nil; realistic = (u=0);
```

There is only one way of satisfying the constraint, by choosing $u_0 = 0$. The system S is run for one cycle, with this input value, let x_0 be the returned value. At the second cycle, we know that

$$\mathsf{pre(u)} = 0\ ,\ \mathsf{pre(dudt)} = 0\ ,\ \mathsf{pre(x)} = x_0$$

So the code of A is simplified to

```
dudt = u; d2udt2 = dudt;
realistic = ($x_0$–delta <= d2udt2) and (d2udt2 <= $x_o$+delta);
```

which gives the (linear) constraint $x_0 - \delta \leq u \leq x_0 + \delta$. Assume the value $u_1 = x_0 + \delta$ is selected, and provided to S, which returns some new value x_1. At the next cycle, we know that

$$\mathsf{pre(u)} = \mathsf{pre(dudt)} = x_0 + \delta\ ,\ \mathsf{pre(x)} = x_1$$

So, the code of A simplifies to

```
dudt = u – ($x_0$+delta); d2udt2 = dudt – ($x_0$+delta);
realistic = ($x_1$–delta <= d2udt2) and (d2udt2 <= $x_1$+delta)
```

which gives the constraint $x_1 + 2x_0 \leq u \leq x_1 + 2x_0 + 2\delta$, and so on...

6 Conclusion

We have presented some validation techniques, which mainly derive from the specification of properties by synchronous observers. While not being restricted to synchronous models, this way of specifying properties is especially natural and convenient in that context, since the same kind of language can be used to describe the system and its properties.

Our presentation was centered on the language LUSTRE, but the techniques could be adapted to any synchronous language. Notice, however, that some ideas were directly suggested by the declarative nature of LUSTRE. For instance, synchronous observers were a natural generalization of the *relations* in ESTEREL, which are a way of expressing known implications or exclusion between input events. When transposed into LUSTRE, these relations are just special cases of invariant Boolean expressions. Generalized to any Boolean LUSTRE expression, this mechanism provides a way of specifying any safety property. Also, in test sequence generation, the idea of considering an observer as a (dynamic) constraint is especially natural when the observer is written in LUSTRE, but can be adapted to any synchronous language.

References

[BCDP99] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

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

[BORZ98] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: testing environment for synchronous software. In *Tool support for System Specification Development and Verification*. Advances in Computing Science, Springer, 1998.

[Bou98] A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CES86] 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.

[DR94] R. De Simone and A. Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In D. Dill, editor, *6th International Conference on Computer Aided Verification, CAV'94*, Stanford, June 1994. LNCS 818, Springer Verlag.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

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

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[JPV95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunication software. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), July 1995. LNCS 939, Springer Verlag.

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[Mar98] B. Marre. Test data selection for reactive synchronous software. In *Dagstuhl-Seminar-Report 223: Test Automation for Reactive Systems - Theory and Practice*, September 1998.

[MHMM95] M. Müllerburg, L. Holenderski, O. Maffeis, and M. Morley. Systematic testing and formal verification to validate reactive programs. *Software Quality Journal*, 4(4):287–307, 1995.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, June 1986.

Emerging Application Domains and the Computing Fabric

Krishna V. Palem

The Georgia Institute of Technology and
Courant Institute, NYU
`http://react-ilp.cs.nyu.edu`

Computing devices are clearly proliferating in a variety of domains ranging from controlling household appliances, to the cockpits of aircraft. Thus, there is an ever increasing demand for cheap and compact computers intended to perform a few functions, extremely well. Historically, *customization* has been the answer to this need. However, customization unfortunately implies very high costs—the exacerbated costs limit the scope of proliferation of course. To redress this and thus enable the extraordinary growth-potential of these emerging domains—loosely referred to as embedded systems here—current research in computing is revisiting established and stable technologies ranging from (high-level) programming languages at the software end of the spectrum, to gate-level design and synthesis at the hardware end. A theme that seems to be emerging, is to provide the future application developer with the advantages of customization, at costs approaching currently mass-produced commercial-off-the-shelf (COTS) software and hardware. Thus, hardware is increasingly being viewed as a flexible fabric, amenable to low-cost customization based on the application developer's needs and preferences. A goal of this talk is to outline the "point-technologies" that are being innovated to help realize this vision.

Concretely, let us consider an application development cycle today for a COTS microprocessor. Typically, an application developer starts out with a program developed in high-level language and then compiles it using an optimizing complier, into code that runs on hardware with a fixed *instruction-set-architecture* (or ISA). This notion of application development is now being revisited, where the fixed aspect of the vendor-specified ISA is no longer considered to be essential. Specifically, *two* alternate approaches are being envisioned. *In the first*, the programmer's high-level language application is viewed as input to a process aimed at designing a customized implementation of key kernels from the application that need to be accelerated: examples of such kernels include the cosine transform central to MPEG2, the ghostscript part of a postscript application and others. In this setting, a portion of the silicon stays dedicated to the executing kernel during its life-time. A *second* approach envisions processors that have a "core" with a fixed vendor-supplied ISA, but with some amount of *customizable* logic added on. In this context, the goal is to identify and execute key kernels from the application as customized implementations that use the custom logic; in this approach, the same piece of logic and hence silicon is reused as new kernels are encountered.

To realize either of the above goals, it is crucial to provide tools and technologies that can accept a high-level C (or Java) application, and transform it into an optimized implementation in silicon. A significant part of this lecture will be devoted to surveying extant and emerging technologies that aim to provide such solutions. These "next-generation" technologies start out as an amalgam of existing optimizing compilers, as well as CAD tools for VLSI design. To actually achieve the goals stated earlier, a number of research challenges emerge, ranging over areas in programming languages, computer architecture, VLSI design, and hardware-software co-design, to name a few. A significant part of this talk will be devoted to surveying this landscape and identifying key research questions of interest. The Trimaran (www.trimaran.org) research infrastructures is especially well-suited for conducting this research; the talk will conclude with a brief overview of this infrastructure.

The Game of the Name in Cryptographic Tables

Roberto M. Amadio[1] and Sanjiva Prasad[2]

[1] Université de Provence, Marseille
[2] Indian Institute of Technology, Delhi

Abstract. We present a name-passing calculus that can be regarded as a simplified π-calculus equipped with a *cryptographic table*. The latter is a data structure representing the relationships among names. We illustrate how the calculus may be used for modelling cryptographic protocols relying on symmetric shared keys and verifying secrecy and authenticity properties. Following classical approaches [3], we formulate the verification task as a reachability problem and prove its decidability assuming finite principals and bounds on the sorts of the messages synthesized by the attacker.
Keywords: cryptographic protocols, π-calculus, verification.

1 Introduction

Cryptographic protocols are commonly used to establish secure communication channels between distributed *principals*. Cryptographic protocols seem good candidates for formal verification and several frameworks have been proposed for making possible *formal* and *automatable* analyses. Formal analyses require formalization of (a) the protocol, (b) the attacker model, and (c) security properties, as well as (d) an effective technique to check satisfaction of the properties. Addressing the vulnerabilities of the protocol rather than those of the cryptosystem, a number of approaches assume 'perfect encryption' and model a protocol as a collection of interacting processes competing against a hostile environment. These approaches usually rely either on model-checking techniques (see, *e.g.*, [5]), or on general-purpose proof assistant tools to establish invariant properties (see, *e.g*, [6]).

The model-checking approach has been remarkably successful in uncovering subtle protocol bugs [5], but its applicability is limited to finite instances of protocols with bounds on various parameters such as the number of runs and the complexity of the messages. Moreover, they require an explicit modelling of the attacker. In contrast, theorem-proving approaches model the attacker's capabilities abstractly and can deal with more general situations, but are not fully automatic.

A more recent trend has been the use of *name-passing* process calculi for studying cryptographic authentication protocols. Abadi and Gordon have presented the *spi*-calculus [1], an extension of the π-calculus with cryptographic primitives. Principals of a protocol are expressed in a π-calculus-like notation, whereas the attacker is represented implicitly by the process calculus notion of 'environment'. Security properties are modelled in terms of contextual equivalences, in contrast to previous approaches which are based on the security model of, *e.g.*, [3]. The *spi*-calculus provides a precise notation with a formal operational semantics, particularly for expressing the generation of fresh names, for the scope of names, and for identifying the different threads in a protocol and the order of events in each thread. These features are important: in the various notations found in the literature, the issues of name generation, scoping, data sorts, and synthesis capabilities of the adversary were often treated in an *ad hoc* and/or approximate manner.

Unfortunately, the addition of the cryptographic primitives to the π-calculus considerably complicates reasoning about the behaviour of processes. Although there have been some attempts to simplify this reasoning (see, e.g., [2]), the developed theory has not yet led to automatic or semi-automatic verification methods.

In this paper, we present a simple name-passing process calculus equipped with a cryptographic table and show how (symmetric key) cryptographic protocols may be modelled. The aim is to develop a framework for automatable analyses of security properties with minimal modelling of the attacker. The main novelties of our work lie in (a) the use of the table to characterise the sharing of information between principals and the environment, and the cryptographic capabilities of the environment; and (b) the use of *sorts* of messages synthesized by the environment to limit the state space that needs exploration.

We depart from the approach of Abadi and Gordon in three major ways. First, we insist on considering every transmissible value as a name, thus eliminating complications to the theory arising from structured values. We keep track of the relationships amongst names (what name is a ciphertext of what plaintext) by means of a *cryptographic table*. Secondly, we model secrecy and authenticity properties as reachability properties that are largely insensitive to the ordering of the actions and to their branching structure. Intuitively, we designate configurations reached after a successful attack as *erroneous*. Protocol verification then involves showing invariance of the property that such error configurations are not reachable. Thirdly, we eliminate named communication channels and let all communications between principals and the environment transit on a public medium.

In this framework we show, by a "diagram chasing" method, that the verification problem is decidable provided processes are finite and finite bounds are assumed on the sorts of the messages synthesized by the environment. Recent work by Huima [4] seems to suggest that this last hypothesis can be removed.

2 The Calculus

We define a process calculus enriched with a 'cryptographic table' to model and analyse symmetric key cryptographic protocols. We use $a, b, \ldots$ for names and $\mathbf{a}, \mathbf{b}, \ldots$ for vectors of names. N denotes the set of names. In the sequel, principals' behaviour as well as secrecy and authenticity annotations will be represented as processes.

Definition 1 (processes). *A process (typically p, q) is defined by the following grammar:*

$$\begin{aligned} p ::= &\ \mathbf{0} \mid err \mid !a.p \mid ?a.p \mid (\nu a)\,p \mid [a=b]p,q \mid p \mid q \mid A(\mathbf{a}) \\ &\mid \mathsf{let}\ a = \{\mathbf{b}\}_c\ \mathsf{in}\ p \mid \mathsf{case}\ \{\mathbf{b}\}_c = a\ \mathsf{in}\ p\ . \end{aligned}$$

As usual, $\mathbf{0}$ is the terminated process; *err* is a distinguished 'error' process; $!a.p$ sends a to the environment and becomes p; $?a.p$ receives a name from the environment, binds it to a and becomes p; $(\nu a)\,p$ creates a new restricted name a and becomes p; $[a=b]p,q$ tests the equality of a and b and accordingly executes p or q; $p \mid q$ is the parallel composition of p and q; $A(\mathbf{a})$ is a recursively defined process; $\mathsf{let}\ a = \{\mathbf{b}\}_c\ \mathsf{in}\ p$ defines a to be the encryption of $\mathbf{b}$ with key c in p; finally, $\mathsf{case}\ \{\mathbf{b}\}_c = a\ \mathsf{in}\ p$ defines $\mathbf{b}$ to be the decryption of a with key c in p. The input and restriction operators act as name binders. Moreover, a is bound in $\mathsf{let}\ a = \{\mathbf{b}\}_c\ \mathsf{in}\ p$ and the names $\mathbf{b}$ are bound in $\mathsf{case}\ \{\mathbf{b}\}_c = a\ \mathsf{in}\ p$. We denote with $fn(p)$ the set of names free in p. We assume that for every process identifier $A(\mathbf{a})$ there is a unique recursive equation $A(\mathbf{a}) = p$ such that $fn(p) \subseteq \{\mathbf{a}\}$.

Let T be a relation in $(\bigcup_{k\geq 1} N^k) \times N \times N$. We write $a \in n(T)$ if the name a occurs in a tuple of the relation T. We write $(\mathbf{b}, c, a) \in T$ as $\{\mathbf{b}\}_c = a \in T$. This notation is supposed to suggest that c is a key, $\mathbf{b}$ is a tuple of plaintext, and a is the corresponding ciphertext. The relation T induces a strict order $<_T$ on $n(T)$ which we define as the least transitive relation such that: $\{b_1, \ldots, b_n\}_c = a \in T \;\Rightarrow\; b_1, \ldots, b_n, c <_T a$.

Definition 2 (cryptographic table). *A cryptographic table T is a relation in $(\bigcup_{k\geq 1} N^k) \times N \times N$ which satisfies the following properties: T is finite, $<_T$ is acyclic,*

$$\begin{array}{ll} \{\mathbf{b}\}_c = a \in T \text{ and } \{\mathbf{b}\}_c = a' \in T \;\Rightarrow a = a' & (T \text{ is single valued}), \\ \{\mathbf{b}\}_c = a \in T \text{ and } \{\mathbf{b}'\}_{c'} = a \in T \Rightarrow \mathbf{b} = \mathbf{b}' \text{ and } c = c' & (T \text{ is injective})\ . \end{array}$$

We introduce a notion of *sort* for the names in a cryptographic table.

Definition 3 (sorts). *The collection of sorts Srt is the least set that contains the ground sort* 0 *and such that $(s_1, \ldots, s_n) \in Srt$ if $s_i \in Srt$ for $i = 1, \ldots, n$ with $n \geq 2$.*

Every name occurring in a cryptographic table can be assigned a unique sort.

Definition 4 (sorting). *Let T be a cryptographic table. We define a function $srt_T : n(T) \to Srt$ as follows:*

$$srt_T(a) = \begin{cases} 0 & \textit{if } a \textit{ is minimal in } <_T \\ (s_1, \ldots, s_n) & \textit{if } \{b_1, \ldots, b_{n-1}\}_{b_n} = a \in T,\ srt_T(b_i) = s_i, i = 1, \ldots, n\ . \end{cases}$$

Intuitively, a sort describes the *shape* of a message represented by a name. For example, if a has sort $(0, (0, 0))$, it represents a message of the form $\{b\}_k$, where k itself is of the form $\{c\}_{k'}$. The notion of sort plays an important role in the development of the results of section 4, particularly in bounding the state space that needs exploration.

Definition 5 (configuration). *A configuration r is a triple $(\nu\{\mathbf{a}\})\,(p \mid T)$ where $\{\mathbf{a}\}$ is a* set *of restricted names, p is a process, and T is a cryptographic table.*

We write $r \equiv r'$ if r and r' are identical configurations up to α-renaming of bound names and associativity-commutativity of parallel composition, with $\mathbf{0}$ as its identity.

We define a *reduction relation* on configurations. The first five rules describe the non-cryptographic computation performed by a process. Rules (out) and (in) concern communication: the sending of a name to the environment and the reception of a name from the environment. Rules (ν), (m), and (rec) describe internal computation: generation of new names, conditional, and unfolding of recursive definitions.

$$(out)\ (\nu\{\mathbf{a}\})\,(!a.p \mid q \mid T) \to (\nu\{\mathbf{a}\}\backslash a)\,(p \mid q \mid T)$$

$$(in)\ \ (\nu\{\mathbf{a}\})\,(?a.p \mid q \mid T) \to (\nu\{\mathbf{a}\})\,([b/a]p \mid q \mid T) \quad \text{if } b \notin \{\mathbf{a}\}$$

$$(\nu)\ \ (\nu\{\mathbf{a}\})\,((\nu a)\,p \mid q \mid T) \to (\nu\{\mathbf{a}\} \cup \{a\})\,(p \mid q \mid T) \quad a \notin fn(q) \cup n(T)$$

$$(m)\ \ (\nu\{\mathbf{a}\})\,([a = b]p_1, p_2 \mid q \mid T) \to \begin{cases} (\nu\{\mathbf{a}\})\,(p_1 \mid q \mid T) \text{ if } a = b \\ (\nu\{\mathbf{a}\})\,(p_2 \mid q \mid T) \text{ if } a \neq b \end{cases}$$

$$(rec)\ (\nu\{\mathbf{a}\})\,(A(\mathbf{b}) \mid q \mid T) \to (\nu\{\mathbf{a}\})\,([\mathbf{b}/\mathbf{c}]p \mid q \mid T) \ \text{ if } A(\mathbf{c}) = p$$

The cryptographic table plays a role in the next three rules. Note that the table allows sharing of information between principals and environments. The rules (let^1) and (let^2) compute the ciphertext a' associated with $\{\mathbf{b}\}_c$ in T while adding $\{\mathbf{b}\}_c = a'$ to T if it is not already there. The rule $(case)$ tries to decode the ciphertext a with key c. In the rule $(case)$, a deadlock occurs (specified by the absence of a transition) if either the vectors $\mathbf{b}$ and $\mathbf{b}'$ do not have the same length or an incorrect key is used for decoding.

$$\begin{array}{ll}(let^1) & (\nu\{\mathbf{a}\})\,(\mathsf{let}\ a = \{\mathbf{b}\}_c\ \mathsf{in}\ p \mid q \mid T) \to (\nu\{\mathbf{a}\})\,([a'/a]p \mid q \mid T) \\ & \text{if } \{\mathbf{b}\}_c = a' \in T\end{array}$$

$$\begin{array}{ll}(let^2) & (\nu\{\mathbf{a}\})\,(\mathsf{let}\ a = \{\mathbf{b}\}_c\ \mathsf{in}\ p \mid q \mid T) \to (\nu\{\mathbf{a}\} \cup \{a'\})\,([a'/a]p \mid q \mid T \cup \{\{\mathbf{b}\}_c = a'\}) \\ & \text{if } a' \text{ is fresh and } \ \nexists a''\ \ (\{\mathbf{b}\}_c = a'' \in T).\end{array}$$

$$\begin{array}{ll}(case) & (\nu\{\mathbf{a}\})\,(\mathsf{case}\ \{\mathbf{b}\}_c = a\ \mathsf{in}\ p \mid q \mid T) \to (\nu\{\mathbf{a}\})\,([\mathbf{b}'/\mathbf{b}]p \mid q \mid T) \\ & \text{if } \{\mathbf{b}'\}_c = a \in T\end{array}$$

Finally, the last three rules (let^1_e), (let^2_e), and $(case_e)$ describe the encoding/synthesis and decoding/analysis performed by the environment: in the rule (let^1_e) the environment learns a private name by encoding, in the rule (let^2_e) the environment creates a new ciphertext, and in the rule $(case_e)$ the environment learns new names by decoding.

(let_e^1) $(\nu\{\mathbf{a}, a\})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a\}) \to (\nu\{\mathbf{a}\})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a\})$
if $\{\mathbf{b}, c\} \cap \{\mathbf{a}, a\} = \emptyset$ and $a \notin \{\mathbf{a}\}$

(let_e^2) $(\nu\{\mathbf{a}\})\,(p \mid T) \to (\nu\{\mathbf{a}\})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a\})$
if $\{\mathbf{b}, c\} \cap \{\mathbf{a}\} = \emptyset$, a is fresh, and $\nexists\, a' \ (\{\mathbf{b}\}_c = a' \in T)$

$(case_e)$ $(\nu\{\mathbf{a}\})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a\}) \to (\nu\{\mathbf{a}\}\backslash\{\mathbf{b}\})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a\})$
if $\{a, c\} \cap \{\mathbf{a}\} = \emptyset$ and $\{\mathbf{b}\} \cap \{\mathbf{a}\} \neq \emptyset$

We remark that (i) a very general treatment of the capabilities of the environment is obtained from the rules (*out*), (*in*) and the last three rules and (ii) the attacker knows all the names not explicitly restricted by a ν-operator. We write $r \to_R r'$ to make explicit the reduction rule R being applied. We note that encoding a new tuple (plaintext,key) generates a new ciphertext. The side conditions on the only rules that modify the table — (let^2) and (let_e^2) — ensure that the acyclicity and injective function properties of the cryptographic table are preserved by reduction.

Lemma 1. *The set of configurations is closed under reduction.*

Definition 6 (error). *A* configuration with error *is a configuration having the shape:* $(\nu\{\mathbf{a}\})\,(err \mid p \mid T)$.

We write $r \downarrow err$ if r is a configuration with error (read r commits on err) and $r \downarrow_* err$ if $r \to^* r'$ and $r' \downarrow err$. We note that configurations with errors are closed under reduction. In the sequel we will use the process err to flag various undesirable outcomes such as leakage of secrets or improper authentication, and so we will be interested in deciding whether $r \downarrow_* err$.

We briefly comment on the relationship with the π-calculus. The main differences are: (i) We let all communications go through a unique (unnamed) channel that connects the principals to the environment. (ii) We add cryptographic primitives, which affect the contents of the cryptographic table. In principle, we can code this process calculus in a variety of π-calculus. This amounts to: (i) Decorating all input-output actions with fresh global channels — thus replacing $!b.p$ with $\overline{a}b.p$ and likewise $?b.p$ with $a(b).p$, where a is a fresh name. (ii) Representing the cryptographic table as a process that receives messages on a global channel, say c. The coding and decoding operations are represented as remote procedure calls from the principals and the environment to the cryptographic process. To make sure that messages are not intercepted, we assume that the cryptographic process is the *unique receiver* on the channel c. We refrain from going into this development because it seems much more effective, both in the mathematical development and in the practical applications, to expose directly the structure of the cryptographic table.

Finally, we remark that the reduction rules can be easily turned into a labelled transition system whose actions (internal reduction, input, free and bound output) are inherited from the π-calculus. We can then rely on the π-calculus notion of bisimulation to reason about the equivalence of configurations. Thus our approach does not preclude the expression of security properties based on process equivalence [1].

Some syntactic sugar. We now describe how several concepts may be encoded in the core calculus given above. We first describe several abbreviations that improve readability. We sometimes use *multiple* abbreviations if the order of expanding them out is apparent from the context.

Sending or receiving a tuple of names can be encoded in the calculus with monadic communication by considering tuples as ciphertexts encoded with a distinguished globally-known key g:

$$!(\boldsymbol{b}).p \equiv \mathsf{let}\ a = \{\boldsymbol{b}\}_g\ \mathsf{in}\ !a.p \qquad ?(\boldsymbol{b}).p \equiv ?a.\mathsf{case}\ \{\boldsymbol{b}\}_g = a\ \mathsf{in}\ p\ .$$

A ciphertext $\{\boldsymbol{b}\}_c$ may appear as a component of the output tuple, or as a value in a match, in an encoding or as a parameter. In all these cases it is intended that $\{\boldsymbol{b}\}_c$ stands for the name a resulting from the encoding $\mathsf{let}\ a = \{\boldsymbol{b}\}_c\ \mathsf{in} \ldots$ For instance:

$$\begin{array}{ll} !(\boldsymbol{b'}, \{\boldsymbol{b}\}_c, \ldots).p & \equiv \mathsf{let}\ b = \{\boldsymbol{b}\}_c\ \mathsf{in}\ !(\boldsymbol{b'}, b, \ldots).p \\ [a = \{\boldsymbol{b}\}_c]p, q & \equiv \mathsf{let}\ b = \{\boldsymbol{b}\}_c\ \mathsf{in}\ [a = b]p, q \\ \mathsf{let}\ b = \{\boldsymbol{b'}, \{\boldsymbol{b}\}_c, \ldots\}_{c'}\ \mathsf{in}\ p & \equiv \mathsf{let}\ a = \{\boldsymbol{b}\}_c\ \mathsf{in}\ \mathsf{let}\ b = \{\boldsymbol{b'}, a, \ldots\}_{c'}\ \mathsf{in}\ p \\ A(\boldsymbol{a'}, \{\boldsymbol{b}\}_c, \ldots) & \equiv \mathsf{let}\ a = \{\boldsymbol{b}\}_c\ \mathsf{in}\ A(\boldsymbol{a'}, a, \ldots)\ . \end{array}$$

In a filtered input, we check that the input has a component that is equal to a certain value (marked as $\underline{\underline{b}}$) or that has a certain shape, *e.g.* $\{\boldsymbol{b}\}_c$, and we stop otherwise.

$$\begin{array}{ll} ?(\boldsymbol{b'}, \underline{\underline{b}}, \ldots).p & \equiv ?(\boldsymbol{b'}, c, \ldots).[c = b]p, \mathbf{0} \\ ?(\boldsymbol{b'}, \overline{\overline{\{\boldsymbol{b}\}_c}}, \ldots).p & \equiv ?(\boldsymbol{b'}, b, \ldots).\mathsf{case}\ \{\boldsymbol{b}\}_c = b\ \mathsf{in}\ p\ . \end{array}$$

As in the filtered input, we check that the decryption of the ciphertext yields a certain component.

$$\begin{array}{ll} \mathsf{case}\ \{\boldsymbol{b'}, \underline{\underline{b}}, \ldots\}_c = a\ \mathsf{in}\ p & \equiv \mathsf{case}\ \{\boldsymbol{b'}, x, \ldots\}_c = a\ \mathsf{in}\ [x = b]p, \mathbf{0} \\ \mathsf{case}\ \{\boldsymbol{b'}, \overline{\overline{\{\boldsymbol{c}\}_d}}, \ldots\}_{c'} = a'\ \mathsf{in}\ p & \equiv \mathsf{case}\ \{\boldsymbol{b'}, a, \ldots\}_{c'} = a'\ \mathsf{in}\ \mathsf{case}\ \{\boldsymbol{c}\}_d = a\ \mathsf{in}\ p\ . \end{array}$$

Security property annotations. Next we describe annotations with which we decorate protocols when analysing their secrecy and integrity properties. We must clarify that these annotations (and the auxiliary processes arising from them) are *not* part of the protocol, and find use only in verification, where we analyse the translation (image) of an annotated protocol.

We mark the generation of a name a intended to remain *secret* in a protocol configuration with the annotation $(\nu a)^{sec}\, p$. We can easily program an observer $W(a)$ such that if the environment ever discovers the name a, then an error configuration is reachable:

$$W(a) \equiv ?a'[a = a']err, \mathbf{0} \quad \text{(secrecy observer).} \tag{1}$$

Secrecy annotations are then translated as follows:

$$(\nu a)^{sec}\, p \equiv (\nu a)\, (p \mid W(a)) \quad \text{(secrecy annotation).} \tag{2}$$

In order to program an observer for authenticity properties we need a limited form of private channel. Fortunately, a private key a already allows the encoding of a private use-at-most-once channel a as follows:

$$\overline{a}\boldsymbol{b}.p \equiv !\{\boldsymbol{b}\}_a.p \qquad a(\boldsymbol{b}).p \equiv ?\{\boldsymbol{b}\}_a.p\ . \tag{3}$$

We note that this encoding does not work for arbitrary (multiple-use) channels, since the environment may replay messages, and without mechanisms like time-stamping, it would not be possible to differentiate replayed messages from genuine fresh messages. However, the encoding can be generalised to a bounded use channel, if each message instance contains distinguished components that uniquely index each distinct use of the channel.

To specify authenticity properties we mark certain send and receive actions in the principals with authenticity annotations $auth\_o(\_)$, $auth\_i(\_)$ which represent, respectively, a sender x registering a message m with a 'judge' process along a private channel j (in the sense of (3)) prior to sending that message to y, and the receiver y claiming authenticity of a received message (purportedly sent by x, which it has presumably authenticated). Note that the 'judge' $J_{x,y}$, rules on the authenticity of a single message allegedly sent by x to y.

$$\begin{array}{ll}
auth\_o((x,y,m)).p & \equiv (\nu n)\,\overline{j}(\mathsf{o},(x,y,m),n).j'(\underline{\underline{n}}).p \\
auth\_i((x,y,m)).p & \equiv \overline{j}(\mathsf{i},(x,y,m),\_).p \\
J_{x,y} & \equiv j(d,(\underline{\underline{x}},t,m),n)[d=\mathsf{o}](\overline{j'}n.J'_{x,y}(m)),([d=\mathsf{i}](err,\mathbf{0})) \\
J'_{x,y}(m) & \equiv j(d,(\underline{\underline{x}},t',m'),\_).[d=\mathsf{i}]([m'=m]\mathbf{0},err),J'_{x,y}(m)\ .
\end{array}$$

Here we assume two distinguished names o and i, which indicate whether the authenticity of a message is being registered or claimed. Communication with the judge is over restricted channels j, j' to disallow the environment from making bogus assertions of authenticity. Our encodings will ensure the channel j appears in the principals to which x, y are instanced, and j' only in the principal corresponding to x. Note that although j is used twice, the different uses are distinguishable by the names o and i, and so replays can be recognised.

Relying on these encodings we translate authenticity annotations as follows:

$$!(\boldsymbol{b})^{auth}.p \equiv auth\_o(\boldsymbol{b}).!(\boldsymbol{b}).p \qquad ?(\boldsymbol{b})^{auth}.p \equiv ?(\boldsymbol{b}).auth\_i(\boldsymbol{b}).p\ .$$

We observe that in an authenticated output, the principal receives an acknowledgement from the judge process on channel j' *before* actually outputting the message. Since we assume processes do not fail, our encoding can rely on the assumption that if a principal registers a message with a judge, it will send that message.

3 Modelling Cryptographic Protocols

We now illustrate how a protocol specified informally can be transformed into an annotated process in the enriched notation of §2. A verification of the protocol is beyond the scope of this paper. We consider a symmetric key protocol due to Yahalom. This protocol concerns principals a, b, and a trusted server c running in a possibly hostile environment that can intercept all communications. Initially, principal a and principal b each share a symmetric secret key with c. At the end of the protocol, principals a, b (and c) share a third symmetric secret key, a 'session key', which can be used by principals a and b to exchange information. A run of the protocol is informally described by the following list of events:

$$\begin{array}{ll}
(1)\ a \to b:\ \ a, b,\ n_a & (4)\ \ a \to b:\ \ b, \{a, k_{ab}\}_{k_{bc}},\ \{n_b\}_{k_{ab}} \\
(2)\ b \to c:\ \ b,\ \{a, n_a, n_b\}_{k_{bc}} & (5)\ \ a \to b:\ \ a, b, \{d\}_{k_{ab}} \\
(3)\ c \to a:\ \ a, \{b, k_{ab}, n_a, n_b\}_{k_{ac}},\ \{a, k_{ab}\}_{k_{bc}} & (5')\ b \to a:\ \ b, a, \{d\}_{k_{ab}}\ .
\end{array}$$

Principal a sends a clear-text message containing a nonce challenge to b. Instead of responding directly to a, b generates a new nonce n_b, its response to a's challenge, which, added to components of the original message, is sent encrypted to c. Server c creates a secret key k_{ab}, which is placed in two separately encrypted message pieces that are sent to a: the first part, readable by a, contains a's original challenge, b's retort n_b and the shared secret k_{ab}. The other part, not readable by a but by b, contains the same shared secret and a's identity; it is forwarded by a in event 4 to b, together with b's challenge *encrypted with this new shared secret*. We have explicitly mentioned the intended recipients in messages 1, 3, and 4. We have also shown (two possibilities of) the first post-protocol message 5 (5′) in which datum d is sent encrypted using the new session key k_{ab}.

The *secrecy property* we would like to verify is that the keys k_{ac}, k_{bc}, and k_{ab} remain secret. The *authenticity property* that we would like to verify is that the message successfully received by b at the second part of event 5 (alternatively by a in 5′) of the protocol is the same as the one sent by a (respectively, by b in 5′). An informal justification of the protocol is that following *event* 3, principal a will believe it is interacting with principal b if the third element of the message component encrypted with k_{ac} is the same as the nonce n_a that it had generated in *event* 1. Following *event* 4, principal b will believe it has authenticated and established a secure channel with principal a, provided the nonce n_b encrypted with the received key k_{ab} is equal to the nonce generated at *event* 2.

We will model a system *q(a,b,c)* consisting of processes p_a, p_b, and p_c representing principals a, b and c, which are assumed to follow the protocol honestly. Verifying that a session between a and b cannot be attacked means showing that the system *q(a,b,c)* can never evolve into a configuration with error.

From the message sequence chart above we can extract the sequence of events where a principal acts as a sender or a receiver, as well as the name generation, cryptographic and matching operations that it performs. The process p_a, for the first alternative of the first post-protocol event, is then specified as follows:

$$\begin{array}{l}
(1)\ (\nu n_a)\,(!(a, b,\ n_a). \\
(3)\ ?(\underline{\underline{a}}, \{\underline{\underline{b}}, k, \underline{\underline{n_a}}, n\}_{k_{ac}}, y). \\
(4)\ !(b, y, \{n\}_k). \\
(5)\ !(a, b, \{d\}_k)^{auth}.\mathbf{0})\ .
\end{array}$$

The behaviours p_b and p_c are defined in a similar way. The process we consider for analysis is:

$$q(a, b, c) \equiv (\nu k_{ac})^{sec}\,(\nu k_{bc})^{sec}\,(\nu j)\,(\nu j')\,(p_a \mid p_b \mid p_c \mid J_{a,b} \mid \emptyset)$$

where $\emptyset$ is the empty cryptographic table. Keys k_{ac} and k_{bc} being long-term secrets, we restrict these names at the top-level. To indicate that the authenticity judge is observing a session involving a and b, we place a judge process $J_{a,b}$ in parallel with the principals and restrict the names j, j'. What we have presented is only illustrative: we also have to consider a similar process with event 5′ instead of event 5, and with the judge $J_{b,a}$. In general, to consider the protocol operating in more complicated scenarios, we can emend q by enriching the contexts in which the principals are placed. For instance, allowing the environment to attempt replay attacks using messages from other sessions may be achieved by starting with a non-empty table.

4 Reachability

In the previous sections, we expressed secrecy and authenticity properties as reachability properties. We now present some results on the problem of determining whether a configuration can reach one with error. As usual, a name substitution σ is a function on names that is the identity almost everywhere.

Definition 7 (injective renaming). *Let r and r' be configurations. We write $r \cong r'$ if there is an injective substitution σ such that $\sigma r \equiv r'$.*

We study reachability modulo injective renaming.

Lemma 2. (1) *The relation $\cong$ is reflexive, symmetric, and transitive.*

(2) *If $r \cong r'$ then $r \downarrow err$ iff $r' \downarrow err$.*

(3) *If $r \cong r'$ and $r \to_R r_1$ then $\exists\, r_1'\ \ r' \to_R r_1'$ and $r_1 \cong r_1'$.*

We consider rewriting modulo injective renaming.

Lemma 3. *Let $r \equiv (\nu\{\mathbf{a}\})\,(p \mid T)$ be a configuration. Then:*

(1) *The set $\{r' \mid r \to_R r'\}$ where R is not (let^2_e) is finite modulo injective renaming.*

(2) *Given a sort s, the set of configurations to which r may reduce by (let^2_e), while introducing a name of sort s in the cryptographic table, is finite modulo injective renaming.*

Therefore, if we can bound the sorts in the cryptographic table then the reduction relation defined in section 2 is finitely branching modulo injective renaming.

A second important result is that all reduction rules except input are strongly confluent modulo injective renaming.

Lemma 4. *Let r be a configuration and suppose $r \to_{R_1} r_1$ and $r \to_{R_2} r_2$ where R_1 is not the input rule. Then*

$$\exists\, r_1', r_2'\ \ (r_1 \to^{0,1}_{R_1'} r_1', r_2 \to^{0,1}_{R_2'} r_2',\ \textit{and}\ r_1' \cong r_2')$$

where R_2' in not the input rule and $\to^{0,1}_{R}$ indicates reduction in 0 or 1 steps.

The rule (let^2_e) can be postponed except in certain particular cases.

Lemma 5. *Suppose r is a configuration and*

$$r \equiv (\nu \mathbf{a})\,(p \mid T) \rightarrow_{let^2_e} r' \equiv (\nu \mathbf{a})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a'\}) \rightarrow_R r''$$

where $\{\mathbf{b}\}_c = a'$ is the tuple introduced by the first reduction. Then in the following cases the first reduction (let^2_e) can be postponed or eliminated:
(1) *If $R = (in)$ and the name taken in input is not a' then*

$$\exists\, r_1, r_2 \ \ (r \rightarrow_{in} r_1 \rightarrow_{let^2_e} r_2) \ \text{and}\ r'' \cong r_2 \ .$$

(2) *If $R = (let^2_e)$ and $r'' \equiv (\nu \mathbf{a})\,(p \mid T \cup \{\{\mathbf{b}\}_c = a', \{\mathbf{b_1}\}_{c_1} = a'_1\})$ where $\{\mathbf{b_1}\}_{c_1} = a'_1$ is the tuple introduced by the second reduction and this tuple does not depend on a',* i.e., *$a' \notin \{\mathbf{b_1}, c_1\}$. Then the two ($let^2_e$) reductions can be permuted:*

$$r \rightarrow (\nu \mathbf{a})\,(p \mid T \cup \{\{\mathbf{b_1}\}_{c_1} = a'_1\}) \rightarrow r'' \ .$$

(3) *If $R = (let_1)$ and we have*

$$\begin{array}{l} r' \ \equiv (\nu \mathbf{a})\,(p'' \mid \mathsf{let}\ a = \{\mathbf{b}\}_c \ \mathsf{in}\ p' \mid T \cup \{\{\mathbf{b}\}_c = a'\}) \rightarrow_{let_1} \\ r'' \equiv (\nu \mathbf{a})\,(p'' \mid [a'/a]p' \mid T \cup \{\{\mathbf{b}\}_c = a'\}) \ . \end{array}$$

Then the (let^2_e) reduction can be eliminated as follows:

$$r \rightarrow_{let^2} (\nu \mathbf{a}, a')\,([a'/a]p' \mid p'' \mid T \cup \{\{\mathbf{b}\}_c = a'\}) \rightarrow_{let^1_e} r''$$

(4) *For all cases of R other than (in), (let^2_e) or (let_1): $\exists\, r_1, r_2$ $(r \rightarrow_R r_1 \rightarrow_{let^2_e} r_2)$ and $r'' \cong r_2$.*

In the remaining cases the name a' affects the following rule, so that (let^2_e) reduction cannot be postponed. Next, we introduce a measure $d(s)$ of a sort s which will provide an upper bound on the number of (let^2_e) reductions that might be needed for the synthesis of a name.

Definition 8 (sort measure). *We define a measure d on sorts as follows:*

$$d(0) = 0 \quad d(s_1, \ldots, s_n) = 1 + d(s_1) + \ldots + d(s_n) \ .$$

Lemma 6. *Suppose $r_1 \rightarrow_{let^2_e} \cdots \rightarrow_{let^2_e} r_{n+1} \rightarrow_{in} r_{n+2}$ where the name received in input in the last reduction has sort s and $n \geq d(s)$. Then at least $n - d(s)$ (let^2_e) reductions can be postponed modulo injective renaming,* i.e.

$$r_1 \equiv r'_1 \rightarrow_{let^2_e} \cdots \rightarrow_{let^2_e} r'_{d(s)+1} \rightarrow_{in} r'_{d(s)+2} \rightarrow_{let^2_e} \cdots \rightarrow_{let^2_e} r'_{n+2}$$

and $r'_{n+2} \cong r_{n+2}$.

Proofhint. The construction of a name a of sort s taken in input needs at most $d(s)$ (let^2_e) reductions. All the other (let^2_e) reductions can be moved to the right of the input by iterated application of the lemma 5(1-2). ◇

To summarise, (let^2_e) reductions can be postponed except when they are needed in the construction of a name to be input. In this case, the number of needed (let^2_e) reductions is bounded by $d(s)$ if s is the sort of the input name.

Next, let us concentrate on the reachability problem in the case where all principals are finite processes (in practical applications this is often the case). We note that the secrecy and authenticity annotations compile to finite processes.

Definition 9 (configuration measure). *We define the measure of a configuration* $r \equiv (\nu\{\mathbf{a}\})\,(p \mid T)$ *as the pair* $(|p|, |\mathbf{a}|)$ *where* $|p|$ *is the size of the process and* $|\mathbf{a}|$ *is the cardinality of* $\{\mathbf{a}\}$*.*

Lemma 7. (1) *Rules* (out), (in), (ν), (m), (let^1), (let^2), *and* $(case)$ *decrease the size of the process* $|p|$*.*

(2) *Rules* (let^1_e) *and* $(case_e)$ *decrease the size of the restricted names* $|\mathbf{a}|$ *while leaving unchanged the size of the processes.*

(3) *Rule* (let^2_e) *leaves the measure* $(|p|, |\mathbf{a}|)$ *unchanged.*

(4) *If* $r \rightarrow_{let^2_e} r'$ *and* $r' \downarrow err$ *then* $r \downarrow err$*.*

In the following, we concentrate on the issue of deciding reachability of a configuration with error assuming that for every input we can compute a *finite and complete* set of sorts which is defined as follows.

Definition 10 (complete set of sorts). *Given a configuration* $r \equiv (\nu\mathbf{a})\,(?a.p \mid q \mid T)$ *we say that a set of sorts* S *is* complete *for the input* $?a.p$ *if whenever there is a reduction sequence starting from* r *leading to* err *and whose first reduction is performed by* $?a.p$*, there is a reduction sequence leading to* err *where the name taken in input has a sort in* S*.*

The problem of determining tight bounds on a complete set of sorts is not trivial. Consider $p \equiv (\nu k)\,(?a.!\{a\}_k \mid ?\{\{c\}_{k'}\}_k err)$. It is easily checked that the set $\{0\}$ is not complete for the input $?a.!\{a\}_k$, but the set $\{(0,0)\}$ is. Nevertheless, if a finite complete set of sorts can be computed then the following strategy decides if r can reach err.

1. Perform in an arbitrary order the reductions different from (let^2_e) and (in).
2. Analyse the current configuration:
   (a) err has been reached: stop and report that err is reachable.
   (b) err has not been reached and no (in) reductions are possible: backtrack if possible, otherwise report that err is not reachable and stop.
   (c) Otherwise:
      i. Non-deterministically select an input action and compute a finite complete set of sorts for it, say $\{s_1, \ldots, s_n\}$.
      ii. Non-deterministically perform a sequence of (let^2_e) reductions of length at most $max\{d(s_1), \ldots, d(s_n)\}$.
      iii. Non-deterministically select an input for the input action. Goto step 1.

Theorem 1. *Starting from a configuration* r *the strategy above terminates and it will report an error iff* $r \downarrow_* err$*.*

Proofhint. To show termination, note that every loop from step 1 to step 2(c)(iii) decreases the well-founded measure of definition 9.

($\Rightarrow$) The strategy examines a subset of the reachable configurations and therefore it is obviously sound.

($\Leftarrow$) Let r be the initial configuration. The rewriting in step 1 terminates in a configuration r' by lemma 7. By iterated application of lemma 4 and lemma 2, if $r \downarrow_* err$ then $r' \downarrow_* err$. In step 2(b), if we have not reached err then we can safely claim by lemma 7(4) that err is not reachable. In step 2(c)(ii), we know, by lemma 6, that if there is a sequence that leads to error then there is a sequence that leads to error whose initial sequence of (let_e^2) is bounded by the sorts' measure. By lemma 3(2), there is a finite number of sequences of (let_e^2) reductions of a given length (modulo injective renaming). Thus there are a finite number of cases to consider. In step 2(c)(iii), we apply again lemma 3(1) to conclude that there are a finite number of cases to consider modulo injective renaming. $\diamond$

Future work. We need to explore whether there are techniques to datermine bounds on the sorts of input names. Even assuming bounds on sorts of inputs, we need to study the practical feasibility of our technique and its complexity.

Acknowledgements. We would like to thank Kevin Compton, Mads Dam, Andrew Gordon, and Joachim Parrow for discussions and the anonymous referees for suggestions. The first author was partly supported by *IFCPAR 1502-1* and *Action Coopérative PRESYSA* and the second author by *IFCPAR 1502-1*, *AICTE 1-52/CD/CA(08)/96-97*, and BRICS.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proc. ACM Computer and Comm. Security*, 1997.
2. M. Abadi and A. Gordon. A bisimulation method for cryptographic protocols. In *Proc. ESOP 98, Springer Lect. Notes in Comp. Sci. 1382*, 1998.
3. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.
4. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Formal methods and security protocols, FLOC Workshop, Trento*, 1999.
5. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. TACAS, Springer Lect. Notes in Comp. Sci. 1996*, 1996.
6. L. Paulson. Proving properties of security protocols by induction. In *Proc. IEEE Computer Security Foundations Workshop*, 1997.

A Parallel Approximation Algorithm for the Max Cut Problem on Cubic Graphs

T. Calamoneri[1]*, I. Finocchi[1], Y. Manoussakis[2]**, and R. Petreschi[1]

[1] Dept. of Computer Science - Univ. of Rome “La Sapienza” - Italy
{calamo,finocchi,petreschi}@dsi.uniroma1.it
[2] Dept. of Computer Science - Univ. Paris Sud - Orsay - France
yannis@lri.lri.fr

Abstract. We deal with the *maximum cut problem* on cubic graphs and we present a simple $O(\log n)$ time parallel algorithm, running on a CRCW PRAM with $O(n)$ processors. The approximation ratio of our algorithm is $1.\bar{3}$ and improves the best known parallel approximation ratio, i.e. 2, in the special case of cubic graphs.

Keywords: Cubic graphs, Max Cut, NP-completeness, PRAM Model.

1 Introduction and Notation

In this paper we deal with the *maximum cut problem*, that can be formally stated as follows:

INSTANCE: An undirected n-vertex graph $G(V, E)$.

SOLUTION: A partition of V into two disjoint sets V_r (right side class) and V_l (left side class).

MEASURE: The number of edges with one endpoint in V_l and one endpoint in V_r, i.e. the cardinality of $E_s = \{(u, v)$ such that either $u \in V_l$ and $v \in V_r$ or $u \in V_r$ and $v \in V_l\}$.

The problem of finding a maximum cut of a given graph is, in general, NP-complete [6, 7] and has been deeply studied (see, for example, [3, 4, 5, 11, 12, 14, 16, 17]).

In this paper we point out our attention on the special class of cubic graphs [1, 9]. Even restricted to this class, the maximum cut problem does not become easier, since it has been proved to be NP-complete if the graph is triangle-free and is at most cubic [18] and to be APX-complete, even if the degree of G is bounded by a constant [15]. Here we present new theoretical results characterizing the cardinality of the cut in cubic graphs with respect to the degree of vertices in

* This author was partially supported by the Italian National Research Council (CNR).

** Part of this research has been done while visiting Dept. of Computer Science, University of Rome “La Sapienza”.

the graph (V, E_s). These results make it possible to design a simple $O(\log n)$ time parallel algorithm, running on a CRCW PRAM with $O(n)$ processors.

The best known approximation ratio for the maximum cut problem in parallel is 2 and follows from [13]. Our results improve it in the special case of cubic graphs, since our algorithm achieves an approximation ratio $1.\overline{3}$. So, this parallel algorithm approaches the best known sequential approximation ratio, that is 1.138 proved in [8]. It is worthwhile to note that the sequential algorithm for Max Cut presented in [8] is based on the use of the primal-dual technique and its parallelization seems not to be easy.

The remainder of this paper is organized as follows. Section 2 considers the problem from a theoretical point of view, while Section 3 addresses the design of the parallel approximation algorithm. Conclusions and open problems are presented in Section 4.

In the sequel, we denote with B the bipartite graph $B(V_l \cup V_r, E_s)$ and with E_d the set $E - E_s$, i.e. $E_d = \{(u, v)$ s.t. either $u, v \in V_l$ or $u, v \in V_r\}$. From now on we call edges in E_s and E_d *solid* and *dotted* edges, respectively.

We partition the vertices of V_l (V_r) according to their *solid degree*, that is their degree in B. In particular, $V_l = L_0 \cup L_1 \cup L_2 \cup L_3$ and $V_r = R_0 \cup R_1 \cup R_2 \cup R_3$, where L_i and R_i are the sets of vertices of solid degree $i = 0, 1, 2, 3$ in V_l and V_r, respectively. We also denote with l_i (r_i) the cardinality of L_i (R_i).

2 Some Theoretical Results

The aim of this section is to give sufficient conditions on V_l and V_r in order to guarantee the approximation ratio obtained in this work.

In the following we state some results referring to V_l, but – for symmetry properties – it will be always possible to exchange the roles of V_l and V_r.

First of all, observe that it is always possible to modify the partition so that no vertex has solid degree 0; indeed, if such a vertex exists, it is enough to move it from its class to the other one. Therefore, it is not restrictive to suppose $l_0 = r_0 = 0$.

Moreover, under the condition $l_1 = l_2 = 0$, we can trivially deduce the following equations which will be useful for the rest of the section:

1. $3l_3 = 3r_3 + 2r_2 + r_1$, derived by counting the number of solid edges as sum of solid degrees of vertices in V_l and V_r.
2. $n = l_3 + r_1 + r_2 + r_3$, derived by counting the total number of vertices.

Lemma 1. *If V_l contains only vertices of solid degree* 3*, i.e. $V_l = L_3$, then $\frac{n}{4} \leq l_3 \leq \frac{n}{2}$.*

Proof. The condition is easily deduced from Equations 1 and 2. In particular, in order to obtain the lower bound we isolate r_1 from Equation 2 and we substitute it in Equation 1. For what concerns the upper bound, we isolate r_3 from Equation 2 and we substitute it in Equation 1.

It is to notice that lower and upper bounds on l_3 hold when $r_2 = r_3 = 0$ and $r_1 = r_2 = 0$, respectively.

Lemma 2. *If V_l contains only vertices of solid degree 3, i.e. $V_l = L_3$, then $r_1 \geq 2n - 5l_3$.*

Proof. From Equation 1 and Equation 2 it follows that $r_2 + r_3 = \frac{3l_3 + r_2 - r_1}{3}$ and that $r_1 = n - l_3 - (r_2 + r_3)$, respectively.

By substituting the first equality in the second one, simple calculations lead to $r_1 = \frac{3}{2}n - 3l_3 - \frac{r_2}{2}$. In this latter equality we substitute $r_2 \leq n - l_3 - r_1$, obtained by Equation 2 and by the fact that $r_3 \geq 0$.

The inequality in the statement follows immediately.

Let c be the maximum number of pairwise vertex-disjoint odd length cycles in G and let c_r be the number of odd length cycles in the subgraph of G induced by V_r. Note that, due to the structure of the partition of vertices, all cycles in V_r are vertex-disjoint. Trivially, $c \geq c_r$. Moreover, let us remind that the *odd girth* g of a graph G is defined as the length a shortest odd cycle (if any) in G.

Lemma 3. *For any partition (V_l, V_r) of the vertices of G it holds: $0 \leq c_r \leq \frac{r_1}{g}$.*

Proof. It is easy to see that only vertices of R_1 can be involved in cycles in the subgraph induced by V_r. Moreover, this subgraph is a collection of isolated vertices, simple paths and cycles. The upper bound for c_r holds when all the connected components of the subgraph induced by V_r are odd length cycles, each having length g.

3 A Parallel Algorithm for Finding a "Large" Cut

In this section we present a parallel algorithm for finding a "large" cut of a cubic graph.

The idea behind our algorithm is to find any bipartition of the vertices and to increase the number of edges in the cut by appropriately moving vertices from one side of the bipartition to the other. In particular, we move to the opposite side both vertices of solid degree 0 and vertices of solid degree 1; indeed, as shown in Figure 1, each time we transfer a 0-degree vertex it becomes a 3-degree vertex and the number of solid edges increases by 3 and each time we transfer a 1-degree vertex it becomes a 2-degree vertex and the number of solid edges increases by 1.

Before detailing the algorithm for approximating the maximum cut, we present some preliminary procedures for coloring vertices that will be used in the following.

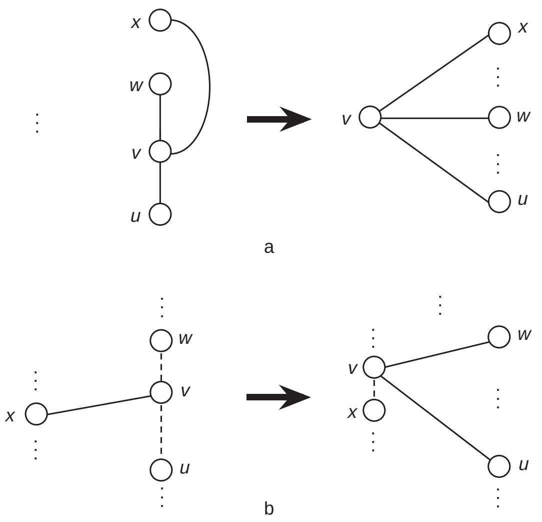


Fig. 1. (a) Vertex v is moved from R_0 to V_l; (b) Vertex v is moved from R_1 to V_l.

Lemma 4. *Let $G(V, E)$ be a not necessarily connected graph of maximum degree 2. Then $O(\log n)$ parallel steps, using $O(n)$ processors on a EREW PRAM model, are sufficient to find a 3-coloring of G, where the number of vertices having color 3 is as small as possible (possibly 0).*

Proof. Observe that $G(V, E)$ is a collection of isolated vertices, paths and cycles.

It is not difficult to recognize all the connected components and to decide whether they are vertices, paths or cycles by using the pointer jumping technique [10].

Then we work on each connected component as follows:

– If the connected component is an isolated vertex, we give it color 1.
– If the connected component is a path, we find a 2-coloring using the pointer jumping technique. Namely, we start from any vertex v, we assign to v color 1 and to its neighbors color 2; in the general step i of the pointer jumping loop, we assign the color of vertex j to not yet colored vertices at distance 2^i from j. First observe that after $\lceil \log n \rceil - 1$ iterations each vertex has a color, as guaranteed by the pointer jumping technique. The found coloring is trivially a 2-coloring and it is valid. Indeed, vertex v assigns its color (that is, 1) to all vertices with even distance from it, while v's adjacent vertices assign their color (that is, 2) to all vertices with even distance from them, and so with odd distance from v. It follows that adjacent vertices cannot have the same color.

– If the connected component is a cycle, we find a 3-coloring such that color 3 is assigned to at most one vertex. We choose at random an edge (u, w) of the cycle and we remove it; what remains is obviously a path and the previous procedure can be run. If in the output u and w have the same color, then we set $c(w) = 3$. The proof of correctness is very similar to the previous one.

For what concerns the parallel complexity, the pointer jumping technique guarantees a $O(\log n)$ time using $O(n)$ processors on a EREW PRAM model.

We want to underline that although there exist more efficient algorithms to find a 3-coloring of a cycle [2], the previous algorithm guarantees that at most one vertex receives color 3.

Before stating the next lemma, we need to introduce the concept of *independent dominating set* of a graph. An independent dominating set of a graph $G(V, E)$ is a subset $V' \subseteq V$ such that for any vertex $u \in V - V'$ there is a vertex $v \in V'$ for which $(u, v) \in E$, and such that no two vertices in V' are joined by an edge in E.

Lemma 5. *Let $G(V, E)$ be a graph of maximum degree 3. Then $O(\log n)$ parallel steps, using $O(n)$ processors on a CRCW PRAM model, are sufficient to find an independent dominating set S for G.*

Proof. Assume w.l.o.g. that G is connected (otherwise we apply the same argument for each connected component of G).

We build an independent dominating set S as follows. First, we find a rooted spanning tree T of G and assign to each vertex its level in T. Then, we consider the subgraph induced by the vertices at odd level; as it has maximum degree 2, we can 3-color it according to the algorithm in the proof of Lemma 4. We put in S all vertices colored 1 and we delete from G both them and their neighbors. The remaining vertices are a subset of vertices at even level in T and have maximum degree 2. We 3-color the graph induced by these vertices and we add to S all vertices colored 1.

We now prove the correctness of the algorithm.

S is an independent set. First, a 3-coloring of the subgraph induced by all the vertices at odd level is found and color 1 is an independent set S' for it. Then, the subgraph induced by all the vertices at even level that are not adjacent to any vertex in S' is considered. By 3-coloring its vertices and considering color 1, we construct an independent set S'' for this subgraph. $S = S' \cup S''$ is an independent set because, by construction, no vertex in S'' is adjacent to any vertex in S'.

S is a dominating set. The coloring algorithms ensure that each vertex colored 2 is adjacent to at least one vertex colored 1, and that each vertex colored 3 is adjacent to exactly one vertex colored 1 and one vertex colored 2. Moreover, each vertex at even level deleted after the first coloring is adjacent to at least a vertex colored 1. This guarantees that each vertex in $V - S$ is adjacent to at least one vertex colored 1.

For what concerns the complexity and the PRAM model, let us consider each step separately. Finding a spanning tree requires $O(\log n)$ time using $O(n)$ processors on a CRCW PRAM [10]. Rooting the tree and leveling its vertices can be done through the Euler Tour technique [10] in $O(\log n)$ time using $O(n)$ processors on a EREW PRAM. Finding connected components and coloring them requires $O(\log n)$ time with $O(n)$ processors on a EREW PRAM in view of Lemma 4. All the other tests and operations are performed in constant time.

Now we are ready to describe the parallel algorithm for finding a "large" cut.

As already stated, we start by any bipartition of the vertices. The first step consists in eliminating vertices of solid degree 0, if they exist. Furthermore, in order to satisfy the hypotheses of Lemma 1 and Lemma 2, we move some vertices from V_l to V_r in order to obtain $V_l = L_3$. Observe that $|E_s|$ may decrease during this step, but we need it for obtaining a stronger structure of the bipartite graph B.

At this point we eliminate 1-degree vertices from V_r; unfortunately, we are not able to ensure the extinction of 1-degree vertices from the whole graph, because they can be generated in V_l, although they completely disappear from V_r. We can however guarantee that the performance ratio of our algorithm is $1.\bar{3}$.

In the following we give the headlines of our algorithm and then we detail it step by step.

```
**ALGORITHM** Parallel-Approx-Max-Cut($G$);
**Input**: a cubic graph $G(V, E)$ with $V = \{0, 1, \ldots, n-1\}$
**Output**: A bipartition ($V_l$,$V_r$) of the vertices of $G$ such that the Max Cut is approximated by a ratio of $1.\bar{3}$
**Step 1:**
    $V_l \leftarrow \{v \in V$ such that $v$ is even$\}$
    $V_r \leftarrow \{v \in V$ such that $v$ is odd$\}$
    Eliminate-0-Degree($V_l$,$V_r$)
**Step 2:**
    Make-Left-Side-Of-Degree-3($V_l$,$V_r$)
    Eliminate-0-Degree($V_l$,$V_r$)
**Step 3:**
    Eliminate-1-Degree-From-Right-Side($V_l$,$V_r$)
    Eliminate-0-Degree($V_l$,$V_r$)
    RETURN($V_l$,$V_r$)
```

Recall that moving a node of solid degree 0 or 1 to the opposite side improves the number of solid edges since all its incident dotted edges become solid and vice versa. As our algorithm works in parallel, we must pay attention in avoiding that adjacent vertices are moved in the same parallel step, because this fact could make useless the local improvement. Hence, our algorithm makes strong use of coloring procedures to check this independence property. In the following, we detail its main steps.

- **Eliminate-0-Degree(V_l,V_r)**
  The aim of this procedure is to eliminate 0-degree vertices from G by moving some of them to the opposite side. Observe that transferring an independent dominating set of $L_0 \cup R_0$ makes $L_0 = R_0 = \emptyset$ and the solid degree of each moved vertex becomes 3. Thus Lemma 5 can be applied to find an independent dominating set whose vertices can be moved to the opposite side in a single parallel step. The procedure returns the updated sets V_l and V_r.
  This procedure can be run on a CRCW PRAM model in $O(\log n)$ time using $O(n)$ processors.
- **Make-Left-Side-Of-Degree-3(V_l,V_r)**
  In order to eliminate all vertices of solid degree 1 or 2 in the left side, let us consider the subgraph induced by $L_1 \cup L_2$ (by the previous algorithm step we can assume $L_0 = \emptyset$). If we consider a 3-coloring of such graph and move to V_r vertices of any two colors, we guarantee remaining vertices have degree 3. Since we are interested in making l_3 as large as possible (remind $|E_s| = 3l_3$), among all sets of vertices of any couple of colors we choose to move the less numerous one. A convenient 3 coloring can be found by means of the algorithm described in the proof of Lemma 4.
  This procedure can be run on a EREW PRAM model in $O(\log n)$ time using $O(n)$ processors.
- **Eliminate-1-Degree-From-Right-Side(V_l,V_r)**
  Let us consider the subgraph induced by R_1. After 3 coloring its vertices, we move those with color 1 to V_l in order to eliminate 1-degree vertices from the right side. Unlike the previous procedure, here we move only one color because we want to minimize the possible new vertices of degree 1 created in V_l (note that trying to move both a vertex colored 3 and its adjacent vertex colored 1 should imply that both of them are added to L_1). Unfortunately, not moving vertices colored 3 does not guarantee to have L_1 empty at the end of the procedure: indeed, a vertex in L_3 can become of degree 1 each time it is adjacent to two vertices in V_r colored 1.
  The running time of this procedure is the same as the previous one.

Lemma 6. *The execution of the procedure Eliminate-1-Degree-From-Right-Side (V_l, V_r) increases the cardinality of E_S by k, if k vertices are moved from R_1 to V_l.*

Proof. The assertion follows from the fact that moved vertices are independent since all of them have color 1 and from the observation of Figure 1(b).

Lemma 7. *During the execution of the procedure Eliminate-1-Degree-From--Right-Side(V_l, V_r) at least $\frac{r_1}{2} - \frac{c_r}{2}$ vertices are moved from V_r to V_l.*

Proof. Let us denote by p_o and p_e the number of vertices in R_1 involved in odd- and even-length paths in the subgraph induced by V_r, where the *length* of a path is defined to be the number of its edges. Analogously, c_o and c_e denote the number of vertices in R_1 involved in odd- and even-length cycles in the same subgraph.

As the procedure Eliminate-1-Degree-From-Right-Side(V_l, V_r) is concerned, we move from the right side to the left one the following number of vertices:

- for each path of even length l, $\frac{l}{2}$ vertices; therefore, totally at least $\frac{p_e}{2}$. Indeed, let $l_1, l_2, \ldots, l_k$ be the lengths of even-lengths paths in the subgraph induced by V_r. Then, $l_1 - 1, l_2 - 1, \ldots, l_k - 1$ are the numbers of vertices of degree 1 in such paths. Therefore, $\sum_{i=1}^{k} l_i = p_e + k$. The number of moved vertices is $\sum_{i=1}^{k} \frac{l_i}{2} = \frac{1}{2}p_e + \frac{1}{2}k \geq \frac{1}{2}p_e$;
- for each odd-length path of length l, $\frac{l-1}{2}$ vertices; therefore – with reasonings similar to the previous ones – totally exactly $\frac{p_o}{2}$;
- for each even-length cycle, exactly half of its vertices; therefore, totally exactly $\frac{c_e}{2}$;
- for each cycle of odd length l, $\frac{l-1}{2}$ vertices; therefore, totally exactly $\sum_{i=1}^{c_r} \frac{l_i-1}{2}$, where l_i is the length of the i-th odd-length cycle. This sum is equal to $\frac{c_o}{2} - \frac{c_r}{2}$.

Hence, by summing up all these contributions, the number of vertices moved from right to left is at least $\frac{p_e}{2} + \frac{p_o}{2} + \frac{c_e}{2} + \frac{c_o}{2} - \frac{c_r}{2} = \frac{r_1}{2} - \frac{c_r}{2}$.

The algorithm outputs a partition of the vertices of the graph into two sets V_l and V_r and the solution is the set E_s, i.e. the set of edges connecting V_l with V_r. The next theorem guarantees the approximation ratio $1.\bar{3}$ for our algorithm.

Theorem 8. *The performance ratio of* `Parallel-Approx-Max-Cut(`G`)` *is* $1.\bar{3}$.

Proof. Observe that the size of the optimal maximum cut cannot exceed the difference between the number of edges and the maximum number of vertex disjoint odd-length cycles, i.e. $\frac{3}{2}n - c \leq \frac{3}{2}n - c_r$.

From the idea behind the algorithm and Lemma 7 it follows that the size of our approximate solution is at least $3l_3 + \frac{r_1}{2} - \frac{c_r}{2}$.

Consequently, the approximation ratio of our algorithm is

$$R \leq \frac{\frac{3}{2}n - c_r}{3l_3 + \frac{r_1}{2} - \frac{c_r}{2}}$$

that is a function of c_r, bounded by its maximum over the definition interval of c_r.

This function always decreases since its derivative

$$\frac{-3l_3 - \frac{r_1}{2} + \frac{3}{4}n}{(3l_3 + \frac{r_1}{2} - \frac{c_r}{2})^2}$$

is always negative in view of Lemma 1 and Lemma 2:

$$-3l_3 - \frac{r_1}{2} + \frac{3}{4}n \leq -3l_3 - \frac{1}{2}(2n - 5l_3) + \frac{3}{4}n \leq -\frac{n}{8}\frac{n}{4} < 0$$

Considering the definition interval for c_r given in Lemma 3, it is easy to prove that the maximum of our function holds for $c_r = 0$. Furthermore, from Lemma 2 and Lemma 1 we have the following chain of inequalities:

$$R \leq \frac{\frac{3}{2}n}{3l_3 + \frac{r_1}{2}} \leq \frac{\frac{3}{2}n}{3l_3 + \frac{1}{2}(2n - 5l_3)} \leq \frac{\frac{3}{2}n}{n + \frac{n}{8}} = \frac{4}{3} = 1.\bar{3}$$

4 Conclusions

We presented a parallel approximation algorithm for Max Cut on cubic graphs that achieves a performance ratio equal to $1.\bar{3}$, substantially improving the best known parallel approximation ratio, i.e. 2, in the special case of cubic graphs.

Starting from any bipartition of the vertices, the general strategy of our algorithm consists of increasing the number of edges in the cut by appropriately moving vertices from one side of the bipartition to the other.

The algorithm manages with simple coloring procedures and can be efficiently implemented in $O(\log n)$ parallel time on a CRCW PRAM with $O(n)$ processors.

We consider to be interesting to test the experimental behavior of this algorithm and to compare the quality of its solutions with the quality of the solutions found by the best known sequential algorithm on cubic graphs. Moreover, a generalization of the approach to d-regular graphs should also represent a valuable contribution.

References

[1] Calamoneri, T.: *Does Cubicity Help to Solve Problems?* Ph.D. Thesis, University of Rome “La Sapienza”,XI-2-97, 1997.

[2] Cole, R. – Vishkin, U.: Deterministic Coin Tossing with applications to optimal parallel list ranking, *Information and Control*, 70(1), pp. 32-53, 1986.

[3] Delorme, C. – Poljak, S.: Combinatorial properties and the complexity of a max-cut approximation, *European Journal of Combinatorics*, 14, pp. 313-333, 1993.

[4] Fernandez de la Vega, W. – Kenyon, C.: A Randomized Approximation Scheme for Metric MAX-CUT, *IEEE Symposium on Foundations of Computer Science (FOCS98)*, 1998.

[5] Frieze, A.M. – Jerrum, M.: Improved Approximation Algorithms for MAX k-CUT and MAX BISECTION, *Algorithmica*, 18(1), pp. 67-81, 1997.

[6] Garey, M.R. – Johnson, D.S.: *Computers and Intractability: a Guide to Theory of NP-completeness*, W.H.Freeman, 1979.

[7] Garey, M.R. – Johnson, D.S. – Stockmayer, L.: Some Simplified NP Complete Graph Problems, *Theor. Comput. Sci.*, 1, pp. 237-267, 1976.

[8] Goemans, M.X. – Williamson, D.P.: .878-Approximation Algorithms for MAX CUT and MAX 2SAT, *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing (STOC94)*, pp. 422-431, 1994.

[9] Greenlaw, R. – Petreschi, R.: Cubic graphs, *ACM Computing Surveys*, 27(4), pp. 471-495, 1995.

[10] Jájá, J.: *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[11] Kann, V. – Khanna, S. – Lagergren, J. – Panconesi, A.: On the Hardness of Approximating Max k-Cut and its Dual, *Chicago Journal of Theoretical Computer Science*, 1997.

[12] Karloff, H.: How Good is the Goemans-Williamson MAX CUT Algorithm?, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing (STOC96)*, pp. 427-434, 1996.

[13] Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set, *SIAM J. on Computing*, 15, pp. 1036-1053, 1986.

[14] Nesetril, J. – Turzik, D.: Solving and Approximating Combinatorial Optimization Problems (Towards MAX CUT and TSP), *Proc. of SOFSEM97: Theory and Practics of Informatics*, LNCS 1338, pp. 70-81, 1997.

[15] Papadimitriou, C.H. and Yannakakis, M.: Optimization, Approximation, and Complexity Classes, *J. Comput. System Sci.*, 43, pp. 425-440, 1991.

[16] Poljak, S.: Integer Linear Programs and Local Search for Max-Cut, *SIAM Journal on Computing*, 24(4), pp. 822-839, 1995.

[17] Poljak, S. – Tuza, Z.: The max-cut problem- a survey Special Year on Combinatorial Optimization, *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 1995.

[18] Yannakakis, M.: Node- and Edge-Deletion NP-Complete Problems. *Proc. 10th Annual ACM Symp. on the Theory of Comp. (STOC78)*, ACM New York, 1978, pp 253–264.

Rapid Prototyping Techniques for Fuzzy Controllers

Chantana Chantrapornchai[1], Sissades Tongsima[2], and Edwin Sha[3]

[1] Faculty of Science, Silpakorn University, Nakorn Pathom 7300, Thailand
Phone: 66 (34) 253910-0 ext 2257, 2258, `ctana@su.ac.th`
[2] National Electronics and Computer Technology Center
High Performance Computing Laboratory, Bangkok 10400, Thailand
Phone: 662 642-7076 ext. 3008, `stongsim@hpcc.nectec.or.th`
[3] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA
Phone: 1-631-8803, `esha@cse.nd.edu`

Abstract. In this paper, a new framework for rapid system design and implementation for fuzzy systems is proposed. The given system specification is separated into two components: a *conceptual specification* and a *parameter specification.* The conceptual specification defines the system core, which is rarely changed during the system adjustment and may be implemented in hardware at an early stage of the design process. The parameter specification defines parameters that are often changed and are implemented in software for easy adjustment. Such a partitioning approach integrates both hardware and software capability. The presented methodology gives a rapid prototyping and efficient system tuning process which, therefore, can reduce the development cycle time and increase flexibility when designing fuzzy systems.

1 Introduction

In the design of application specific systems, the iterative process of detailing specifications, designing a possible solution, and testing and verifying the solution is repeatedly performed until the solution meets user requirements [3]. However, such a repeated cycle prolongs the system development time. The system life cycle is likely to be short due to rapid pace of improvements in current technology. The need for early prototyping, therefore, becomes critical to implement a system specification and provide customers with feedback during the design process.

Recently, fuzzy systems have been popularly used in many industrial products [17, 9]. Since such systems are designed to mimic human knowledge, the performance of the systems is as good as knowledge given by experts. Therefore, a large number of real-time experiments are required in order to verify the system performance.

Considerable work has been done on developing hardware implementations for general purposed fuzzy control systems. In [11, 4], special hardware chips have been invented to speedup the fuzzy inference engine. Other work has concentrated on design and implementation of fuzzy architectures and processors [15,16,2]. Software fuzzy systems are flexible but fail to provide high-speed result [12]. Although general-purposed fuzzy hardware can yield high-speed output, they may be able to neither meet stringent real-time requirement of specific applications nor specific needs of the applications. Kan and Shragowitz presented presented a generic development tool for fuzzy systems [7]. Their tools provide flexibility in defining membership functions, rules, and fuzzy logic operations. Nevertheless, they do not focus on constructing hardware modules from the derived system. In [10], a computer-aided design tool for implementation of fuzzy controllers on FPGAs was proposed. However, they do not provide flexibility during a prototyping phase,.e.g., membership functions are implemented as a boolean circuit. Unlike the others, our research provides rapid prototyping framework for application specific fuzzy systems as well as efficient implementation.

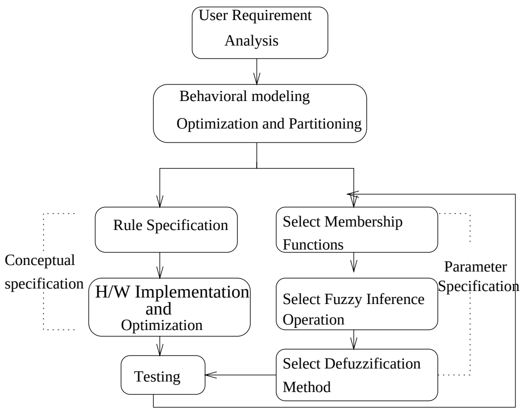

User Requirement Analysis
Behavioral modeling Optimization and Partitioning
Rule Specification
H/W Implementation and Optimization
Testing
Select Membership Functions
Select Fuzzy Inference Operation
Select Defuzzification Method
Conceptual specification
Parameter Specification

Fig. 1. Fuzzy system design flow

Since one of the goals of using fuzzy logic is to reduce computational complexity of designing systems, e.g., fuzzy control systems, fuzzy logic rule bases consist of human knowledge expressed in rule format. Figure 1 shows the proposed design approach mapped to the existing fuzzy system design flow. Based on our methodology, the system specification is characterized into two subcomponents. The *conceptual specification* consists of the rule base portion since it defines system functionality and is seldom changed or required only a few modification. In some classical fuzzy systems which have practical applications such as a temperature controller, and inverted pendulum, the rules can commonly be found [14]. For detailed fuzzy system tuning, varying parameters are considered such as the range of system operation, the shape of membership functions, type of fuzzy logic operations, defuzzification methods, etc. We collectively include these portions in the *parameter specification*. Such a specification can be implemented in software for flexible tuning.

In general, the proposed approach has the following benefits:

1. rapid specification: the conceptual specification gives designers the ability to specify the overall concept of the system which is independent of parameter setting details.
2. flexible tuning: the parameter specification and new system model allow designers to easily adjust the parameters and consider several possible solutions in order to meet user requirements.
3. hardware/software capability: the integration of hardware prototype and flexible parameterized software naturally takes advantages of the ideas of hardware/software co-design.
4. shorter development process: partitioning the system specification into two components allows the possibility of developing hardware and software in parallel and early prototyping.

Our method will be discussed in the remainder of this paper. Section 2 presents our system models. The synthesis methodology based on the models are discussed in Section 3. Section 4 presents an example of prototyping a fuzzy control system. Finally, Section 5 draws conclusion from this work.

2 Terminologies

A rule-based system consists of a collection of conditional statements such as IF-THEN rules. The variables in the if-part are called *input variables* while the variables in the then-part are called *output variables*. A particular value of an input/output variable is called an *input/output instance*. In a fuzzy system, the value of these variables specified in the rules can be *linguistic variable*s.

In the following, we present the *Conceptual State Graph* presents state-oriented behaviors of the rules. The model is close to actual implementation of a rule base.

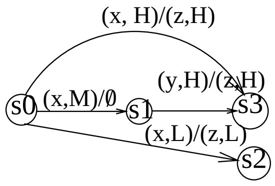
Definition 1. *A Conceptual State Graph (CSG) $G = (\mathsf{S}, \mathsf{I}, \mathsf{O}, \mathsf{E}, \omega)$ is a connected directed edge-weighted graph where S is a set of nodes $s_i \in S, 1 \leq i \leq p$, representing states of the graph, I is the set of inputs, O is the set of outputs including ϕ, $\mathsf{E} \subseteq S \times S$ is a set of edges, denoted by $s_i \overset{e}{\rightarrow} s_j$ or e_{s_i,s_j}, $s_i \in \mathsf{S}, s_j \in \mathsf{S}$, and ω is a function from E to $\mathsf{I} \times \mathsf{O}$, representing the set of edge weights, denoted by x/y, where $x \in \mathsf{I}$ and $y \in \mathsf{O}$.*

For the edge weight x/y of an edge $e_{u,v}$ or $u \overset{e}{\rightarrow} v$, if $y = \phi$, then no output is given for the edge from u to v with the input x.

As an example, Figure 2(b) shows a simple example of a rule base. In this rule base, variables x, y and z are input/output variables. The values of x, y and z can be L, M, H which stand for *low*, *medium* and *high* linguistic variables.

The rule base may represent a temperature control system whose input is the temperature obtained from an external sensor and whose output is an adjusting level for a fan speed. Input linguistic variables *low, medium,* and *high* are used to justify which rules to activate. A temperature value may be justified as a low or medium value as its meaning is fuzzy. Hence, two rules may be fired for each input.

Based on the rules in Figure 2(b), consider the CSG in Figure 2(a). I contains all combinations of values for x and y according to the rules, that is $\mathsf{I} = \{(x, L), (x, M), (x, H), (y, H)\}$. Similarly, $\mathsf{O} = \{(z, L), (z, H), \phi\}$, $\mathsf{S} = \{s_0, s_1, s_2, s_3\}$ and $\mathsf{E} = \{e_{s_0,s_1}, e_{s_0,s_2}, e_{s_0,s_3}, e_{s_1,s_3}\}$ Edges to states s_2 and s_3 have output labels (y, L) and (y, H) respectively. The output label for the intermediate edge (s_0, s_1) is ϕ. The edge weights are: $\omega(s_0, s_1) = (x, M)/\phi, \omega(s_0, s_2) = (x, L)/(z, L), \omega(s_0, s_3) = (x, H)/(z, H), \omega(s_1, s_3) = (y, H)/(z, H)$.



(a) CSG

IF $x = L$	THEN $z = L$
IF $x = M$ and $y = H$	THEN $z = H$
IF $x = H$	THEN $z = H$

(b) Rules

Fig. 2. System models

Using this model, one can easily implement a system with multiple input variables. This can be advantageous since many current fuzzy processors allow only limited number of input variables. The CSG model can be transformed to an incompletely specified FSM and therefore, transformed to a completely specified FSM to which traditional FSM synthesis can be applied [8]. The CSG is similar to the incompletely specified FSM, based on an assumption that no unspecified next state is encountered. All unspecified outputs, ϕ, convey "don't care" outputs and the set of final states are the set of states where outputs are given.

Note that one may view the CSG model similar to fuzzy automata where the state variable is fuzzy [6]. Given a certain input, two states may be fired simultaneously. Nevertheless, in this paper, we regard the CSG implementation as a traditional finite state machine rather than fuzzy finite state machine. Developing a hardware for fuzzy finite state machine is more complex and, more importantly, requires to embed membership functions representing fuzzy states. Therefore, membership functions are not explicitly separated from the CSG, i.e., the conceptual part. However, according to our approach, we intend to clearly distinguish conceptual and parameter portions and establish the flexibility in tuning membership functions in the parameter part. Thus, we apply a traditional FSM synthesis to the CSG model for rapid rule base hardware development.

3 Conceptual Specification Synthesis Algorithms

In order to construct a hardware for a conceptual specificatiton, we first construct a CSG for a rule base as in the following steps.

step 1 Convert the rule specification into a single-output rule format. A rule of the form R : If … Then $B_1, \dots, B_b$ has multiple output variables $(B_1 \dots B_b)$. This form can be broken down into a group of rules, each of which has a single output variable, e.g., R_1: If … Then B_1, …, R_b: If … Then B_b

step 2 Convert the rule into a *conjunctive form*, e.g., If X_1 **and** X_2 **and** … **and** X_m Then Y. The disjunctive clauses A and B in the form: If A **or** B Then C is broken down into two disjunctive rules: R_1 : If A Then C, and R_2 If B Then C. Further details of this conversion can be found in [5].

step 3 Construct a CSG using the following procedure.

Algorithm 1 *Construct_CSG*
Input: Rule base $R = \{r_i\}$, a current starting state s_c
Output: an update the current state graph A

```
begin
  l_max = Find_maxLabel(R)          /* find the input var. with max occurrences */
  if $l \neq (NULL)$
    then begin
          /* partition vertices into two groups $P$ and $Q$ */
          /* $P$ = vertices with input edge label l_max */
          Let $P \subset R$ be a set of rules which contain l_max
          $Q = R - P$;
          foreach $p \in P$ do
            begin
              if l_max is the only input for $p$
                then output = output($p$), $s_n = s_{\text{outlabel}}$          /* output of $p$ */
                 else output = $\phi$, $s_n = s_k$, nonoutput_state = true fi
              if $s_n \notin A.S$
                then $A.S = A.S \cup \{s_n\}$ fi
              if $e_{s_c s_n} \notin A.E$
                then $A.E = A.E \cup \{e_{s_c, s_n}\}$
                      $A.\omega = A.\omega \cup \{$l_max/output$\}$ fi  /* add weight to edge $e_{s_c s_n}$ */
            endfor od
          Ignore that input with label l_max
          if $|P| > 0$
            then if nonoutput_state
                  then $s_{next} = s_k, k = k + 1$, call Construct_CSG with $P, s_{next}$ fi fi
          if $|Q| > 0$
            then call Construct_CSG with $Q, s_c$ fi
        endif fi
end
```

Without loss of generality, assume that all system inputs are available at the same time and each clause in the if-part of each rule is conjunctive. For each n-input rule, a collection of state nodes that form an n-edge path, with input labels according to the rule can be constructed.

In Algorithm 1, the initial state is represented by s_0. All the rules are in set R. The initial call is *Construct_CSG*(R, s_0). The algorithm first counts the number of occurrences from each edge label. The maximum occurrence (edge label *l_max*) is selected. Note that *l_max* is a tuple (X, L) where X is an input variable and L is a linguistic variable with respect to the universe of X. The rules are then partitioned into two groups: P, containing rules that have an input *l_max* and Q, containing rules without *l_max*. The algorithm next examines every rule in P to determine if it has more than one input.

If during the first iteration, a node with more than one input is found, a unique edge from the current state s_c to s_n is constructed. Note that for simplicity in defining a unique state index, two kinds of state indices are introduced. All states are indexed by an integer value except the output states where symbolic names are used. The state s_n in this case is the next non-output state k. In cases where p has only one input edge, s_n is initialized to an output state where *output*(p) is its output. This state is added to the CSG if it does not already exist, and an edge is constructed between s_c and s_n if necessary. Once all rules in group P have been considered, the input with *l_max* are ignored from the set. The *nonoutput_state* flag then determines if there was a rule with more than one input. If so, a new state $s_{next} = s_k$ was constructed during the algorithm. In this case, k is incremented and the algorithm is repeated for P with a new initial state, s_{next}. The algorithm is also repeated for Q using the old initial state. These two recursions stop after all inputs of a given set of rules are considered.

After a CSG is established, a finite state machine (FSM) can simply derived. Before deriving a finite state machine, an FSM optimization may be applied. Then, the inference process can begin using the hardware rule base.

According to the fuzzy inference process, a special calculation is required to determine the output strength of each rule given the associated input strengths. We assume min operation is used for computing fuzzy rule strength. Let $\mu_{X_{ij}}(x_j)$ denote a membership function corresponding to the linguistic variable X_{ij} and the current input instance x_j for rule i. Let $\mu_{Y_i}(y)$ be a membership function corresponding to the linguistic variable Y and output instance y for rule i. Suppose a rule requires m inputs. Given input instance $x_1, \dots, x_n$, for each rule i, Equation 1 determines a modified output function $\mu'_{Y_i}(y)$ that is used in the defuzzification process

$$\mu'_{Y_i}(y) = \begin{cases} \mu_{Y_i}(y) & \text{if } \mu_{Y_i}(y) < r_i \\ r_i & \text{otherwise} \end{cases} \tag{1}$$

where the rule strength r_i, calculated by Equation 2, is the limit of the output strength Y_i.

$$r_i = \mathsf{min}_{j=1}^{n} \mu_{X_{ij}}(x_j) \tag{2}$$

If more than one rules give the same linguistic variable output Y_i and therefore generates a new function $\mu'_{Y_i}(y)$, each $\mu'_{Y_i}(y)$ is combined by using the max operation.

$$\mu''_{Y_i}(y) = \mathsf{max}_{\forall Y_k = Y_i}(\mu'_{Y_k}(y)) \tag{3}$$

Based on the above equations and the given CSG, the following algorithm presents the overall inference process.

***Algorithm 2** InferenceCSG* **Input:** *input instances* $I = \{t_1, t_2, \dots, t_n\}$
Output: *output value* o

```
1  begin
2    foreach $O_p$ where $p$ is the output linguistic variables do $O_p = 0$ od
3    foreach $t_i \in I$ do
4      foreach $l \in L_i$ do                    // for every linguistic variable of input $t_i$
5        compute $\mu_l(t_i)$ od od
7    foreach tuple $(\mu_{(x_1,l_1)}(t_1), \mu_{(x_2,l_2)}(t_2), \dots, \mu_{(x_n,l_n)}(t_n)), \forall l_1 \in L_1, l_2 \in L_2, \dots, l_n \in L_n$ do
8      $r = \min(\mu_{(x_1,l_1)}(t_1), \mu_{(x_2,l_2)}(t_2), \dots, \mu_{(x_n,l_n)}(t_n))$
9      $q$ = compute CSG using $\mu_{(x_1,l_1)}(t_1), \mu_{(x_2,l_2)}(t_2), \dots, \mu_{(x_n,l_n)}(t_n)$
10     if $q$ is not don't care variable $\phi$ then $O_q = max(O_q, r)$ fi
11   odendfor
12   $o$ = defuzzify ( $\{O_p : O_p$ is the output linguistic variables$\}$ )
13 end
```

Let $\{p\}$ be a set of output linguistic variables. O_p refers to a register which holds a current cut value for output linguistic variable p. Initially, they are set to zero. Let L_i be a set of linguistic variables for input x_i and t_i be an input instance for input x_i. In Algorithm 2, Lines 3–5 fuzzify all inputs. After that all fuzzified inputs are fed to the CSG which will direct corresponding output values (Line 9). Meanwhile, the minimum value between these fuzzified values is calculated and will be accumulated if the output is fired in Line 10.

4 Design Case Study

We now consider designing a temperature control system for the oven found in [13]. The control system has two temperature input variables x_1 and x_2 and outputs a new temperature u to which the oven temperature should be adjusted. Figure 3 shows the set of rules which are parts of conceptual specification.

1. IF x_1 = *low* (x_1, L) **and** x_2 = *low* (x_2, L)	THEN u = *high* (u, H)
2. IF x_1 = *low* (x_1, L) **and** x_2 = *medium* (x_2, M)	THEN u = *medium* (u, M)
3. IF x_1 = *low* (x_1, L) **and** x_2 = *high* (x_2, H)	THEN u = *low* (u, L)
4. IF x_1 = *medium* (x_1, M) **and** x_2 = *low* (x_2, L)	THEN u = *high* (u, H)
5. IF x_1 = *medium* (x_1, M) **and** x_2 = *high* (x_2, H)	THEN u = *low* (u, L)
6. IF x_1 = *high* (x_1, H) **and** x_2 = *low* (x_2, L)	THEN u = *high* (u, H)
7. IF x_1 = *high* (x_1, H) **and** x_2 = *medium* (x_2, M)	THEN u = *medium* (u, M)
8. IF x_1 = *high* (x_1, H) **and** x_2 = *high* (x_2, H)	THEN u = *low* (u, L)

Fig. 3. Rule set in oven example

In Figure 3, the tuple next to each clause is a shorthand notation of the clause's linguistic variable. Figure 4 shows the CSG obtained from Algorithm 1 for the above rules. S is $\{s_0, s_1, s_2, s_3, s_{(u,M)}, s_{(u,L)}, s_{(u,H)}\}$. The transitions to states $s_{(u,M)}, s_{(u,L)}$, and $s_{(u,H)}$ output temperature values *medium*, *low*, and *high* respectively.

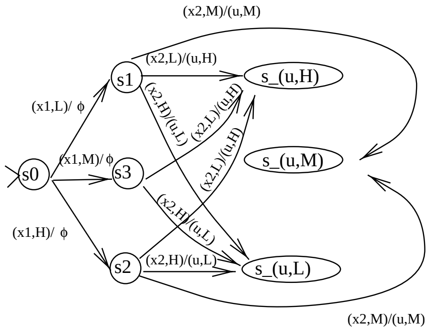


Fig. 4. Behavioral network and CSG

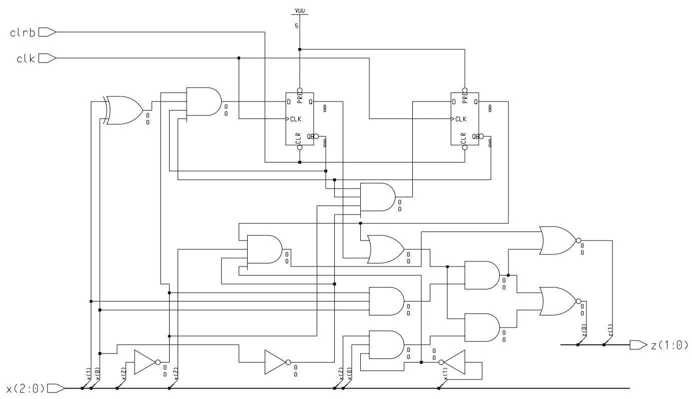


Fig. 5. Combinational circuits implementing Figure 4

Since the rule base is well-established, Figure 5 shows an ASIC implementation of the CSG in Figure 4. In general, one may use rapid prototyping technology such as an FPGA (Field Programmable Gate Arrays) to implement a set of rules. In Figure 5, linguistic variables $(x_1, L), (x_1, M), (x_1, H), (x_2, L), (x_2, M), (x_2, H)$ are represented by $000, 001, 010, 011, 100$ and 101 while output variables $(u, H), (u, M), (u, L), \phi$ are encoded as $00, 01, 10$ and 11 respectively. The initial state s_0 is encoded as 0. States s_1, s_2, and s_3 are encoded as $1, 2$, and 3 respectively. After optimization, states $s_{(u,H)}, s_{(u,M)}$ and $s_{(u,L)}$ can be merged with the initial state to save one control step and be ready to accept the next input. Thus, we only use 2 bits to encode existing states in this design. Any sequence of inputs that does not fire any rule will lead to the initial state. Figure 6 partially demonstrates the behavior of the implementation. For example,

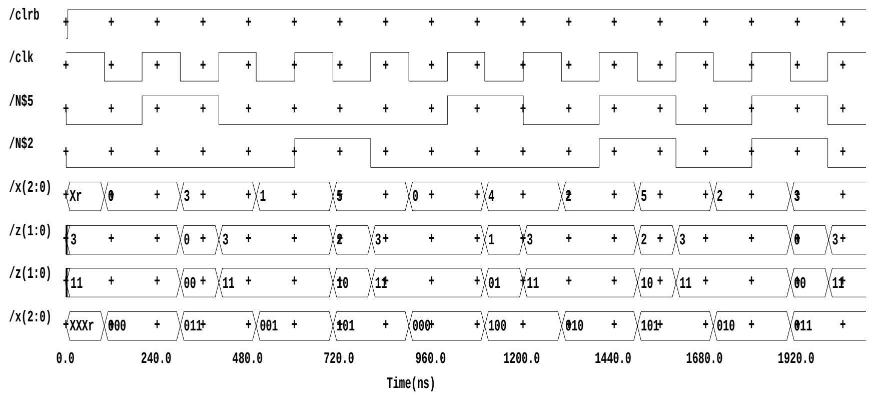


Fig. 6. Simulation results of circuits in Figure 5

in the case of (x_1, L) and (x_2, M), the circuit receives 0 and 4 respectively and produces 1 $((u, M))$ as output. If 2 and 3 are fed to the circuit, it will produce output 0 which is equivalent to rule 6 (If (x_1, H) and (x_2, L) Then (u, H)).

In this design, the parameter specification includes membership functions and defuzzification approaches. Since all input and output variables are temperatures, the same membership functions for all variables are used. The temperature domain is restricted to between 0 and 500 degrees centigrade. Figure 7(a) presents sample codes for these membership functions in C language.

For defuzzification method, we have a choice of using centroid method($z^* = \frac{\int \mu(z)zdz}{\int \mu(z)dz}$) or weighted average method ($z^* = \frac{\sum \mu(\overline{z})\overline{z}}{\sum \mu(\overline{z})}$) [13]. The codes of these methods are shown in Figure 7(b).

Figure 8 shows the integration of the conceptual and parameter components and their data paths for computing outputs. We use Mentor graphics tool [1] to synthesize this circuit. The grey-dashed components, fuzzification and defuzzification, are parameterized components where the others are fixed components. For easy implementation, the membership values are scaled from real values ranging in $[0, 1]$ to integer values between $[0, 256]$. The parameterized part presented in Figure 7 are developed in VHDL codes. Input feeder components sequentially feeds proper inputs to CSG rules (whose implementation is shown in Figure 5). The implementation of this components is simply a series of multiplexers with different inputs. Since the CSG processes one input at a time, we use only one min component to iteratively compute a rule strength. The output of the CSG is applied to control the proper accumulation of the cut on output linguistic variables. The component on the left-bottom in this picture represents the feedback calculation where control action u together with inputs are used to calculate the new inputs for the next cycle. In this example, we use the following equation.

$$x_1(t+1) = -2x_1(t) + x_2(t) + u(T), \qquad x_2(t+1) = x_1(t) - x_2(t)$$

```
float trimf(float x, VAR l)
{
  float output=0;

  switch (l) {
  case L :
    if (x <= 70) output =1;
    else if (x >= 210) output = 0;
    else output = -0.007* x + 1.5;  break;
  case M :
    if (x >= 350 || x <= 70 ) output = 0;
    else if (x <= 210) output = 0.007*x -0.47;
    else output = -0.007*x +2.47; break;
  case H :
    if (x >= 350) output = 1;
    else if (x <= 210 ) output = 0;
    else output = 0.007*x-1.45;
  }
  return  output;
}

float bellmf (float x, VAR l)
{
  float output=0;
  switch (l) {
  case L :
    if (x <= 140) output =1;
    else output =  1/( pow((1+fabs((x-70)/320)), 8) ); break;
  case M :
    output = 1/( pow((1+fabs((x-210)/320)),8) ); break;
  case H :
    if (x >= 350) output = 1;
    else output = 1/( pow((1+fabs((x-350)/320)),8) ); break;
  }
  return output;
}
```

(a) Membership functions for triangular and bell curve

```
float weighted_average (float x, float y, float z )
{
  float output;
  output = (70*x +210 *y + 355*z)/(x+y+z);
  return output;
}

float centriod (float x, float y, float z, float (*mf)(float, VAR ))
{
  int i;
  float point[3],all, sum_of_prod=0, sum=0;
  /* integral discretization */
  for (i=0; i <=500; i++) {

      point[0] = min((*mf)( (float) i,L ), x) ;
      point[1] = min((*mf)( (float) i,M ), y) ;
      point[2] = min((*mf)( (float) i,H ), z) ;
      all = max(point[0],max(point[1],point[2]));
      sum_of_prod += i*all;
      sum += all;
  }
  return sum_of_prod/sum;
}
```

(b) Defuzzification methods (3 inputs) : weighted average and centroid

Fig. 7. Sample codes for fuzzifier and defuzzifier

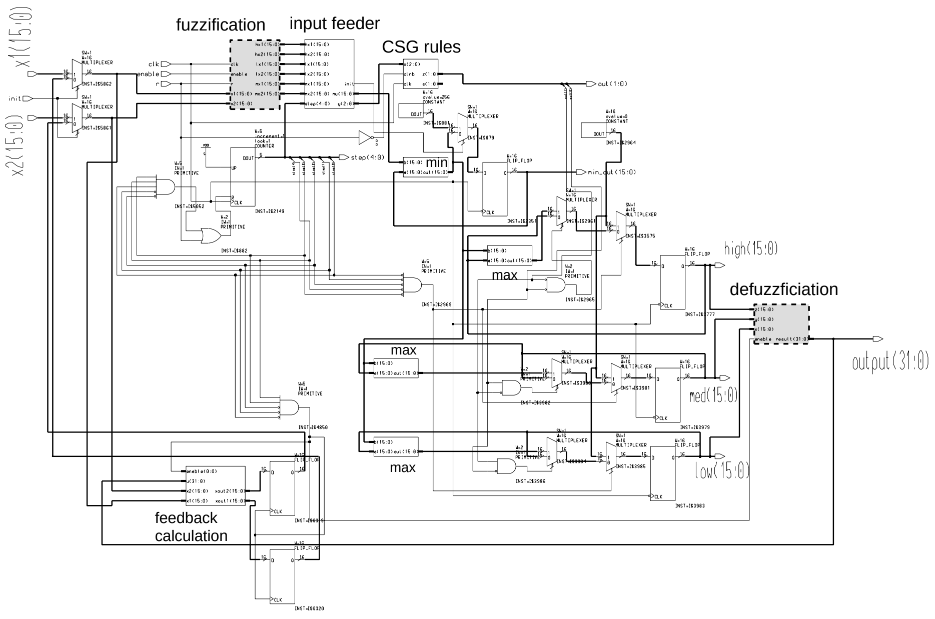


Fig. 8. Schematic of the oven example

Figure 9 presents one cycle of the simulation where initial $x_1 = 90$ and $x_2 = 100$. Assume that membership function (triangular) in Figure 7(a) is used and the weighted average is a defuzzification method. The first seven steps devote to the fuzzification part which fuzzifies $\mu_L(90) = 203, \mu_M(90) = 10$ and $\mu_L(100) = 183, \mu_M(100) = 36$. After all fuzzified values are ready, the rule strength can be calculated. This example takes 24 control steps to compute output $u = 331$. The feedback calculation results in $x_1 = 251$ and $x_2 = -10$ for the next cycle. Based on this design, one can easily adjust the parameter parts, re-insert into the components and redo the simulations. Once the parameter is finalized, the prototype can be set-up rapidly.

5 Conclusion

One of the most important issues in system design is to make the design easy to adjust for system testing and rapid implementation. In this paper, we propose a new design methodology for fuzzy systems which partitions system components into conceptual and parameterized specifications. The conceptual specification defines the core of system specification which is rarely changed and may be implemented as a hardware prototype to implement a fast prototype. Parameterized components are specified in software in order to be easily edited for system modification. This method actually integrates hardware and software capability, yielding flexibility in re-specification and shorten prototyping time.

References

[1] Mentor graphics product. http://www.mentorg.com.
[2] V. Catania et al. A VLSI fuzzy inference processor based on a discrete analog approach. *Computer*, pages 37–46, 1982.
[3] A. L. Eliason. *System development analysis, design, and implementation.* Scott Foresman Little Brown, 1990.
[4] J. W. Fattaruso, S. S. Mahant-Shetti, and J. B. Barton. A fuzzy logic inference processor. In *Proceedings of the Third International Conference on Industrial Fuzzy Control and Intelligent Systems*, pages 210–214, Houston, Texas, December 1-3 1993.
[5] M. Jamshidi, N. Vadiee, and T. J. Ross, editors. *Fuzzy Logic and Control: Software and Hardware Applications*, chapter 4-5, pages 51–111. Prentice-Hall, 1993.
[6] A. Kandel and S. C. Lee. *Fuzzy Switching and Automata.* Crane, Russak and Company Inc., New York, 1 edition, 1979.
[7] M. S. Khan and E. E. Swartzland Jr. Rapid prototyping fault-tolerant heterogeneous digital signal processing systems. In *Proceedings of the Sixth International Workshop on Rapid System Prototyping*, pages 187–193, 1995.
[8] Z. Kohavi. *Switching and Finite Automata Thoery.* McGraw-Hill, 1979.
[9] E. H. Mamdani. Advances in linguistic synthesis of fuzzy controllers. *InternationalJournalMan Machine Studies*, 8:669–678, 1976.
[10] M. A. Manzoul. CAD tool for implementation of fuzzy controllers on FPGAs. *Cybernetics and Systems*, pages 599–609, 1994.

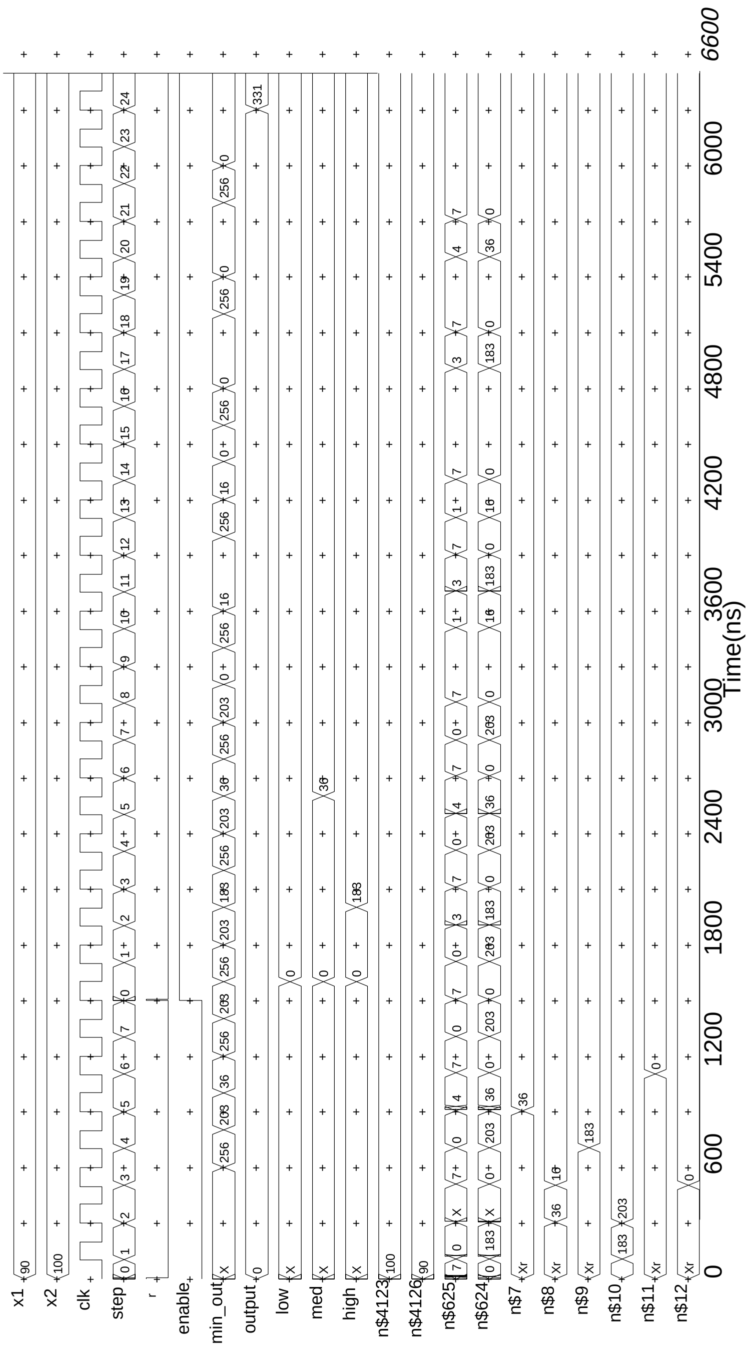


Figure 9. One cycle simulation result of the schematic in Figure 8

[11] M. A. Manzould and H. A. Serrte. Fuzzy Systolic Arrays. In *Proceedings of the 18th International Symposium on Multiple-valued Logic*, pages 106–112, Palma de Mallorca, Spain, 1988. IEEE-CS-Press.

[12] J. Moore and M. A. Manzoul. An interactive fuzzy CAD tool. *IEEE Micro* , pages 68–74, April 1996.

[13] T. J. Ross. *Fuzzy Logic with Engineering Applications*. McGrawHill, 1 edition, 1995.

[14] M. H. Smith. Tuning membership functions, tuning *and* and *or* operations, tuning defuzzication: which is the best? In *Proceedings of the North American Fuzzy Information Processing Society Biannual Conference*, pages 347–351, 1994.

[15] A. P. Ungering and K. Goser. Architecture of a 64-bit fuzzy inference processor. In *Proceedings of the International Conference on Fuzzy Systems*, volume 3, pages 1776–1780, 1994.

[16] H. Watanabe. Risc approach to design fuzzy processor architecture. In *Proceedings of the International Conference on Fuzzy Systems*, volume 3, pages 1809–1814, 1994.

[17] L. A. Zadeh. Fuzzy Logic. *Computer*, 1:83–93, 1988.

Transactional Cache Management with Aperiodic Invalidation Scheme in Mobile Environments

IlYoung Chung and Chong-Sun Hwang

Dept. of Computer Science and Engineering, Korea Univ.
5-1, Anam-Dong, SeongBuk-Ku, Seoul 136-701, KOREA
{jiy, hwang}@disys.korea.ac.kr

Abstract. In mobile client-server database systems, caching of frequently accessed data is an important technique that will reduce the contention on the narrow bandwidth wireless channel. As the server in mobile environments may not have any information about the state of its clients' cache(stateless server), using broadcasting approach to transmit the updated data lists to numerous concurrent mobile clients is an attractive approach. In this paper, a caching policy is proposed to maintain cache consistency for mobile computers. The proposed protocol adopts aperiodic broadcasting as the cache invalidation scheme, and supports transaction semantics in mobile environments. With the aperiodic broadcasting approach, the proposed protocol can improve the throughput by reducing the abortion of transactions with low communication costs. We study the performance of the protocol by means of simulation experiments.

1 Introduction

Mobile computing enables people with unrestricted mobility. It can satisfy people's information needs *at any time and in any place*. Due to the recent development of the hardware such as small portable computers and wireless communication network, data management in mobile computing environments has become an area of increased interest to the database community[2,6,9,21].

In general, the bandwidth of the wireless channel is rather limited. Thus, caching of frequently accessed data item in a mobile computer can be an effective approach to reducing contention on the narrow bandwidth wireless channel[10,14,24]. However, once caching is used, a *cache invalidation* strategy is required to ensure the cached data in mobile computers are consistent with those stored in the server[13,16,25].

Several proposals have appeared in the literature regarding the support of transactions in mobile systems[1,11,15,26]. However, most of these approaches didn't attempt to make use of a common feature in wireless systems: the ability that the server has to *broadcast* information to the mobile clients. Because the server in mobile environments may not have any information about the state of its clients' cache(stateless server), using the broadcasting approach to transmit

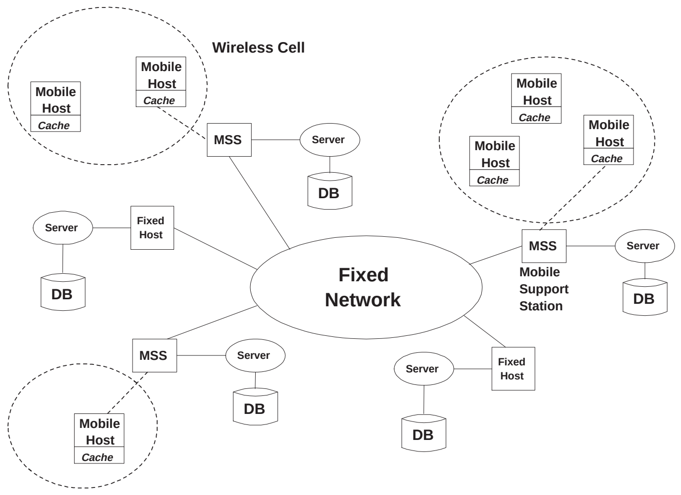


Fig. 1. The mobile database system architecture

the updated data lists to numerous concurrent mobile clients is an attractive approach[8].

Considering these limitations of the mobile environments, Barbara and Imielinski proposed an approach that a server periodically broadcasts an invalidation report that reports the data item which have been updated[7,8]. This approach is attractive in mobile environment because a server need not know the location and the connection status of its clients, and because the clients need not establish an uplink connection to invalidate their cache.

In this paper, we present a protocol that adopts *aperiodic* broadcasting, to reduce the wait time of a transaction that has requested commit, and to reduce the abortion of transactions that may show conflicts in the periodic broadcasting strategy. With aperiodic broadcasting approach, the protocol can reduce the number of broadcasting occurrence under high ratio of updates, and can reduce the abortion of transactions under low ratio of updates.

The remainder of this paper is organized as follows. Section 2 introduces the mobile transaction model. In section 3, we describe and discuss our caching protocol. Section 4 presents experiments and results, and finally we state our concluding remarks in section 5.

2 The Mobile Transaction Model

Figure 1 presents a general mobile database system model. In this model, both a database server and a database are attached to each fixed host[12,22]. Users of the mobile computers may frequently query databases by invoking a series of operations, generally referred to as a transaction. *Serializability* is widely accepted as a correctness criterion for execution of transactions[3,4,17,20]. This correctness criterion is also adopted in this paper. A database server is intended to support basic transaction operations and as resource allocation, commit, and abort.

Each mobile support station(MSS) has a coordinator which receives transaction operations from mobile hosts and monitors their execution in database servers within the fixed networks. Transaction operations are submitted by a mobile host to the coordinator in its MSS, which in turn sends them to the distributed database servers within the fixed networks for execution[2,21].

In general, there are two ways to structure a mobile transaction processing system[26]; the first situation is that the mobile host behaves like a remote I/O device, and the data must be placed in the static part of the network[9,14]. This situation arises when the amount of resources on the mobile device is small. Secondly, we can consider a mobile host as a full fledged server to manage transaction, and place data locally in mobile hosts[1,11,18].

In this paper, we adopt an way between these two situation to structure a mobile transaction processing system. That is we still keep the data locally, but we treat it as a cache rather than as a primary copy. Thus, as shown in figure 1, each mobile host can have its own cache to maintain some portion of data for later reuse. The transaction operation that accesses the data item stored in the cache can be processed without the interaction with the database server. If the data item does not exist in the cache of the mobile host, it sends an message to the server to download the appropriate data item.

3 Our Protocol

In this section, we present our concurrency control protocol, which adopts *aperiodic broadcasting* approaches, supporting the stateless server. We assume that there is a central server that holds and manages all data. Also, we assume that only one transaction may be initiated by a mobile client at any time. That is, a mobile client can initialize a transaction only after the previous transaction has finished.

3.1 Aperiodic Broadcasting

The proposed protocol adopts broadcasting approach to maintain cache consistency, and to control the concurrent transactions. With the broadcasting approach, the mobile client sends the commit request of the transaction after executing all operations, to install the updates in the central database. Then the server decides commit or abort of the requested transactions, and notifies this information to all of the clients in its cell with the broadcasting invalidation reports. The broadcasting strategy does not require high communication costs, nor require the server to maintain additional information about the mobile clients in its cell, thus is attractive in mobile databases[8,26].

Some proposals have appeared in the literature regarding the use of the broadcasting for the control of the concurrent transactions, and all of these approaches adopt synchronous(periodic) manner as the way of broadcasting the invalidation reports[7,8]. In these schemes, the server broadcasts the list of invalidating data items and the list of transactions that will be committed among those which have requested commit during the last period. These schemes present some problems which arise with the synchronous manner of broadcasting approach.

– if two or more conflicting transactions have requested commit during the same period, only one of them can commit and others have to abort.
– the mobile client is blocked on the average for the half of the broadcasting period until it decides commit or abort of the transaction.

In this paper, we adopt aperiodic broadcasting approach as the way of sending the invalidation report. Unlike the schemes using periodic broadcasting, in our scheme, invalidation reports are broadcasted immediately after a commit request arrives. Using the aperiodic broadcasting approach, most transactions that may show conflicts in the same period by synchronous broadcasting, can avoid them, thus the protocol can reduce the abortion rate of transaction processing. Also, the blocking time of the transaction that have sent the commit request can be reduced, as the server immediately broadcasts the invalidation reports which notifies commit or abort.

3.2 The Protocol

Our protocol uses a modified version of optimistic control[5,17,19,27], in order to reduce the communication costs on the wireless network. With optimistic approach, all operations of a transaction can be processed locally using cached data, and at the end of the transaction, the mobile client sends the commit request message to the server. Then the server immediately broadcasts the invalidation report including the data items that should be invalidated, and the mobile client identifier. Receiving invalidation report, the invalidating action preempts the ongoing transaction that shows conflicts with the now-committed transaction. With this approach, the mobile client can early detect the conflicts of a transaction that should be aborted later at the server.

All data items in the system are tagged with a sequence number which uniquely identifies the state of the data. The sequence number of data item is increased by the server, when the data item is updated. The mobile client includes the sequence number of its cached copy of data(if the data item is cache resident) along with the commit request message.

We now summarize our protocol for both the mobile client and the server.

Mobile Client Protocol:

– Whenever a transaction becomes active, the mobile client opens two sets, read_set and write_set. Initially, the state of a transaction is marked as *reading*. Data item is added to these sets with the sequence numbers, when the transaction requests read or write operation on that data item. The state of the transaction is changed in *updating* state, when the transaction requests write operation.

- Whenever the mobile client receives an invalidation report, it removes copies of the data items that are found in the invalidating list. And, if any of the following equations becomes true, the transaction of *reading* state is changed into *read-only* state, and the transaction of *updating* transaction is aborted.

$$\text{read\_set} \cap \text{invalidatinglist} \neq \emptyset$$
$$\text{write\_set} \cap \text{invalidatinglist} \neq \emptyset$$

- When a transaction of *read-only* state requests write operation, the mobile client aborts the transaction.
- When a transaction is ready to commit, the mobile client commits the transaction of *reading* state or *read-only* state locally. If the state of the transaction is *updating*, the mobile client sends a commit request message along with the read_set, write_set and the identification number of the mobile client.
- After that, the mobile client listens to the broadcasting invalidation report which satisfies any of the following equation.

$$\text{read\_set} \cap \text{invalidatinglist} \neq \emptyset$$
$$\text{write\_set} \cap \text{invalidatinglist} \neq \emptyset$$

  If the invalidation report, satisfying above equations, is attached with the identification number of this mobile client, the transaction is committed. Otherwise, the mobile client aborts the transaction, and removes copies of the data items that are found in invalidating list.

Server Protocol:

- Whenever the server receives a commit request from a mobile client, and if sequence numbers of all data items in read_set and write_set are identical with the server's, it broadcasts the invalidation report along with the invalidating list, which is the list of data item in the write_set of the commit request, and identification number of the mobile client. Otherwise, the server just ignores the commit request.

The protocol described above adopts aperiodic broadcasting, thus the server immediately broadcasts invalidation reports, when it receives a commit request from a mobile client. Our protocol has some advantages with the aperiodic approach. At first, when write operations are infrequent at mobile clients, the protocol can reduce the communication costs by broadcasting invalidation reports only when updating transaction occurs. It is unnecessary to send invalidation reports periodically without data items that should be invalidated. On the other hand, when updating transactions occur frequently, the protocol can avoid many

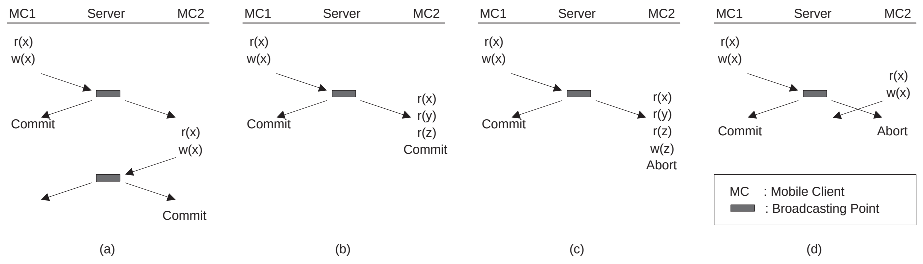


Fig. 2. Example execution

aborts, by reducing the conflicts between updating transactions. In synchronous broadcasting approach, when two or more updating transactions are conflicting in a period, only one of them can commit, as invalidation reports are broadcasted once for a period. Our protocol can avoid much of these aborts, because mobile clients are informed the list of updated data items immediately.

In this case, the increased number of broadcasting is not an additional communication overhead that may degrades the throughput of transaction processing, as the server initiates only one broadcasting for a committing transaction. As shown in the above protocol, no notification is required for an aborted transaction. The server just ignores the commit request with sequence number which has fallen behind the server's number.

Example execution under the protocol is shown in Figure 2. This example uses two mobile clients and a server. In this example, r(x) and w(x) denote a read and write operation performed by a transaction on data item x.

In (a), the transaction initiated in mobile client 2 can be committed by the server, as it can read the data item x that has been updated by the transaction in mobile client 1, by the immediate invalidation report. With the synchronous broadcasting approach, if the commit requests of the two transactions arrive in the identical period, both transactions cannot be committed. In (b) and (c), the state of the transaction in mobile client 2 is changed from reading state to read-only state, when the mobile client receives the broadcasting invalidation report. In case of (b), the transaction can commit locally, as the state of the transaction remains read-only state, when the transaction is ready to commit. However, in (c), the transaction will be aborted, because it requests write operation in read-only state. In example (d), the mobile client 2 sends the commit request slightly before it receives the invalidation report that indicates the updates of the data item x. The server ignores the commit request, as the sequence number of data item x in the message is lower than the server's. No broadcasting message is required for the aborted transaction, and the transaction in mobile client 2 is aborted, receiving the invalidation report of data item x, with the identification of mobile client 1. Thus, in our protocol, the server initiates one broadcasting for each commit of transactions.

4 Performance

In this section, we develop the simulation model and present the results of experiments, in order to evaluate the performance of the proposed algorithm.

4.1 Simulation Model

This section describes our simulation model. We can divide our simulation model into three parts: database model, transaction model, and system model. The database model captures the characteristics of the database, such as the database size and object attribute. The transaction model captures the data object reference behavior of transactions in a workload. And the system model captures the characteristics of the system's hardware and software. Figure 1 in section 2, shows the physical structure of the modeled system. Modeled system is composed of a database server and mobile clients, connected by a wireless network.

Database Model Database is composed of multiple data objects which have the same attributes. There are two attributes for each data object: identifier and sequence number. The Database model parameters are summarized in Table 1. The number of objects in the database, *Nobject*, was chosen to be relatively small in order to model contention. For caching clients, the cache size, *CachePercent*, is a constant. The content of each client's cache is fixed at the start of the simulation by choosing objects uniformly from the database.

Table 1. Database parameters

Parameter	Meaning
NObject	Number of objects in the database
CachePercent	Percentage of cache size

Transaction Model The transaction model supports the following operations.

- *ReadObject* : Reads an object from its client cache. If the object does not exist in the cache, reads it from the server database.
- *WriteObject* : Updates an object in the client cache and increase the sequence number.
- *CommitTR* : Commit a transaction.
- *AbortTR* : Abort a transaction.

A transaction is modeled by a finite loop of *ReadObject* and *WriteObject* operations, which are followed by *CommitTR* or *AbortTR* operation. Table 2 summarizes the parameters that characterize a transaction type. The number of *ReadObject* and *WriteObject* operations in a transaction is called *TRSize*, which is uniformly distributed between *MinTRSize* and *MaxTRSize*. The parameter *ProbWrite* indicates the probability that *WriteObject* operations occur in a transaction. The delay parameters are exponentially distributed delay times used to model interactive system.

Table 2. Transaction parameters

Parameter	Meaning
TRSize	Number of operations in a transaction
ProbWrite	Probability that write operation occurs
ReadDelay	Average delay of a read operation
WriteDelay	Average delay of a write operation
TRState	State of transactions that are processed

System Model The system model consists of a network manager and clients and server modules. The parameter for all the modules are summarized in Table 3.

Table 3. System Parameters

Parameter	Meaning
NClient	Number of mobile clients
NetDelay	Average communication delay on the wireless network
DBAccessDelay	Average delay to access database
CacheAccessDelay	Average delay to access cache
ReadHitProb	Probability of read hit at mobile client

4.2 Experiments

In this section, we present results from several performance experiments involving the protocol described in section 3. The main performance metric presented is system *throughput*, measured in committed transactions per second. The throughput results are, of course, dependent on the particular settings chosen for the various physical system resource parameters. Thus while the throughput results show performance characteristics in what consider to be a reasonable environment, we also present other performance measures, *the number of messages* and *the percentage of aborts*, to provide additional insights into the fundamental tradeoffs between protocols. Table 4 shows the values of the simulation parameters used for these experiments.

Table 4. Simulation Parameter Settings

Parameter	Value
NObject	1,000
CachePercent	5%
TRSize(Max)	15
TRSize(Min)	3
ProWrite	0, 2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20, 22.5, 25%
ReadDelay	10 ms
WriteDelay	40 ms
NClient	20
NetDelay	20 ms
DBAcessDelay	50 ms
CacheAccessDelay	10 ms
ReadHitProb	0.5

4.3 Results

In this section, we present the results from several performance experiments. We ran a number of simulations to compare the behavior of the proposed protocol and the protocol with periodic broadcasting.

Figure 3 shows the required number of messages to commit a transaction with the proposed protocol and the synchronous one. The message counts of each protocol increases in a sublinear fashion in whole range of update ratio. When the update ratio is lower than about 13%, our protocol using aperiodic broadcasting approach requires less messages than the synchronous protocol, because broadcasting is rare with low ratio of updating transactions. If the period is longer than 13%, the frequently broadcasted invalidation reports are the main factor that cause our protocol to require more messages. In the synchronous

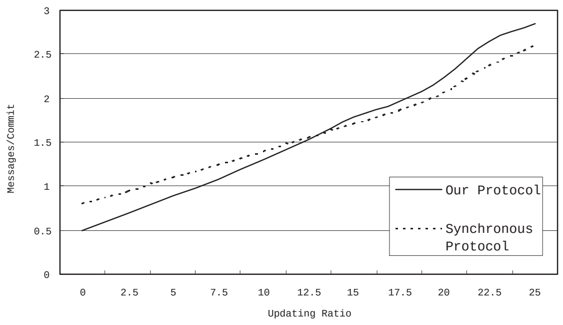


Fig. 3. Impact of updating ratio on message counts

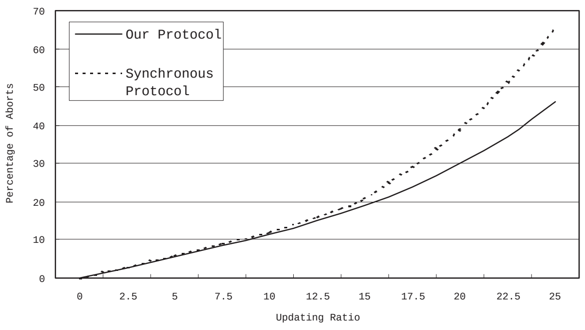


Fig. 4. Impact of updating ratio on aborts of transactions

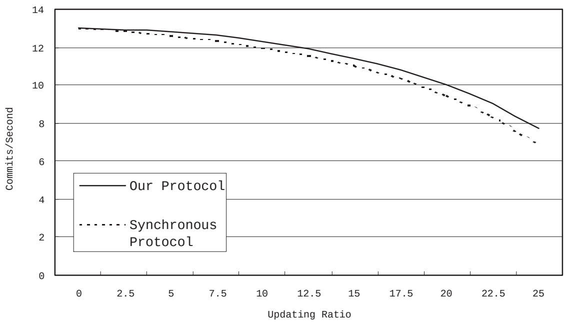


Fig. 5. Impact of updating ratio on throughput

protocol, the message costs does not increase so rapidly as our protocol, because the invalidation reports are broadcasted periodically.

Figure 4 presents the percentage of aborts of transaction that take place both at the client side and at the server. As can be seen, when updating ratio increases, the percentage gets to be significant. When the updating ratio is lower, most aborts are due to the transaction's read operation on the stale cached data. It occurs when an invalidation report is not delivered by disconnection, or when a mobile client requests commit of a transaction before it receives the invalidation report which include the data item referenced by the transaction. As the updating ratio gets higher, Aborts increase with both protocols, mainly because of the increased conflicts between transactions. In case of the synchronous protocol, the percentage increases more rapidly, as the number of transactions that show conflicts in the same broadcasting period increases.

Figure 5 shows the throughput results for two protocols. As shown in this figure, the throughput degrades with increasing write operations, because of the message costs for updating transactions. When the updating ratio is very low, two protocols show almost the same performance, because there are few transactions that are aborted by the server, in both cases. However, the throughput of the synchronous protocol degrades more rapidly as the updating ratio increases, because of increasing aborts of transactions. With synchronous approach, lots of aborts caused by conflicts may happen, as the server checks the conflicts between transactions once in a period. In our protocol, most of such aborts can be avoided by broadcasting invalidation reports immediately after receiving a commit request.

5 Conclusions

We have presented a new protocol to support transactions in mobile client-server environments. The proposed protocol adopts broadcasting cache invalidation strategy to support the stateless server scheme, a common feature in mobile environments. In this paper, we adopt aperiodic approach as the way of broadcasting invalidation reports. With this approach, our protocol can reduce communication costs when updating is rare, and can avoid most aborts of adjacent conflicting transactions without additional communications on the wireless network. Simulations were conducted to evaluate the performance of the protocol. Our performance experiments show that relative merit of the proposed protocol under various updating ratio.

References

1. A. Elmagarmid, J. Jing and O. Bukhres: An Efficient and Reliable Reservation Algorithm for Mobile Transactions. Proc. of International Conference on Information and Knowledge Management (1995)
2. A. Elmagarmid, J. Jing and T. Furukawa: Wireless Client/Server Computing for Personal Information Services and Applications. ACM SIGMOD Record Vol.2 (1997)
3. A. J. Bernstein and P/ M. Lewis: Concurrency in Programming and Database Systems. Jones and Bartlett Publishers (1993)
4. A. S. Tannenbaum: Distributed Operating Systems. Prentice Hall (1995)
5. B. Liskov, M. Day and L. Shrira: Distributed Object Management in Thor. Proc. of the International Workshop on Distributed Object Management (1992) (Published as Distributed Object Management. Ozsu, Dayal, Valduriez: Morgan Kaufmann 1994)
6. B. R. Badrinath and T. Imielinsky: Replication and Mobility. Workshop on the Management of Replicated Data (1992)
7. D. Barbara: Certification Reports: Supporting Transactions in Wireless Systems. Proc. of IEEE International Conference on Distributed Computing (1997)
8. D. Barbara and T. Imielinsky: Sleepers and Workaholics: Caching in Mobile Environments. Proc. of ACM SIGMOD Conference on Management of Data (1994)
9. E. Pitoura and B. Bhargava: Building Information Systems for Mobile Environments. Proc. of International Conference on Information and Knowledge Management (1994)

10. E. Pitoura and B. Bhargava: Maintaining Consistency of Data in Mobile Distributed Environments: Proc. of International Conference on Distributed Computing Systems (1995)
11. J. Jing O. Bukhres and A. Elmagarmid: Distributed Lock Management for Mobile Transactions. Proc. of International Conference on Distributed Computing systems (1995).
12. K. L. Wu, P. S. Yu and M. S. Chen: Energy-efficient Caching for Wireless Mobile Computing. Proc. of the International Conference on Data Engineering (1996)
13. K. Wilkinson and M. Neimat: Maintaining Consistency of Client Cached Data. Proc. of the International Conference on Very Large Data Bases (1990)
14. M. H. Dunham and A. S. Helal: Mobile Computing and Databases: Anything New?. ACM SIGMOD Record (1995)
15. M. H. Wong and W. M. Leung: A Caching Policy to Support Read-only Transactions in a Mobile Computing Environment. Technical Report, The Chinese Univ. of Hong Kong, Dept. of Computer Science (1995).
16. M. J. Franklin: Caching and Memory Management in Client-Server Database Systems. Ph.d. Thesis, Dept. of Computer Science, University of Wisconsin (1993)
17. M. J. Franklin and M. J. Carey: Client-Server Caching Revisited. Proc. of the International Workshop on Distributed Object Management (1992) (Published as Distributed Object Management. Ozsu, Dayal, Valduriez: Morgan Kaufmann 1994)
18. M. J. Franklin, M. J. Carey and M. Livny: Global Memory Management in Client-Server DBMS Architecture. Proc. of the International Conference on the Very Large Data Bases (1992)
19. M. J. Franklin, M. J. Carey and M. Livny: Local Disk Caching for Client-Server Database Systems. Proc. of the International Conference on Very Large Data Bases (1993).
20. P. A. Berstein, V. Hadzilacos and N. Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
21. R. Alonso and H. Korth: Database System Issues in Nomadic Computing. Proc. of the ACM SIGMOD Conference on Management of Data (1993)
22. S. Acharya and R. Alonso: The Computational Requirements of Mobile Machines. Proc. of the International Conference of Engineering of Complex Computer Systems (1995)
23. S. I. Chu and M. Winslett: Minipage Locking Support for Object-Oriented Page-Server DBMS. Proc. International Conference on Information and Knowledge Management (1994)
24. T. Imielinsky and B. R. Badrinath: Data Management for Mobile Computing ACM SIGMOD Record (1993)
25. V. Gottemukkala, E. Omiecinski and U. Ramachandran: Relaxed Consistency for a Client-Server Database. Proc. of International Conference on Data Engineering (1996)
26. V. R. Narasayya: Distributed Transactions in a Mobile Computing System. Proc. of International Conference on Parallel and Distributed Systems (1993)
27. Y. Wang and L. Rowe: Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. Proc. of the ACM SIGMOD International Conference on the Management of Data (1991)

Observable Sharing for Functional Circuit Description

Koen Claessen and David Sands

Department of Computing Sciences,
Chalmers University of Technology and Göteborg University, Sweden.
`www.cs.chalmers.se/~{koen,dave}`

Abstract. Pure functional programming languages have been proposed as a vehicle to describe, simulate and manipulate circuit specifications. We propose an extension to Haskell to solve a standard problem when manipulating data types representing circuits in a lazy functional language. The problem is that circuits are finite graphs – but viewing them as an algebraic (lazy) datatype makes them indistinguishable from potentially infinite regular trees. However, implementations of Haskell do indeed represent cyclic structures by graphs. The problem is that the sharing of nodes that creates such cycles is not observable by any function which traverses such a structure. In this paper we propose an extension to call-by-need languages which makes graph sharing observable. The extension is based on non updatable reference cells and an equality test (sharing detection) on this type. We show that this simple and practical extension has well-behaved semantic properties, which means that many typical source-to-source program transformations, such as might be performed by a compiler, are still valid in the presence of this extension.

1 Introduction

In this paper we investigate a particular problem of embedding a hardware description language in a lazy functional language – in this case Haskell. The "embedded language" approach to domain-specific languages typically involves the designing a set of combinators (higher-order reusable programs) for an application area, and by constructing individual applications by combining and coordinating individual combinators. See [Hud96] for examples of domain-specific languages embedded in Haskell. In the case of hardware design the objects constructed are descriptions of circuits; by providing different interpretations of these objects one can, for example, simulate, test, model-check or compile circuits to a lower-level description. For this application (and other embedded description languages) we motivate an extension to Haskell with a feature which we call *observable sharing*, that allows us to detect and manipulate cycles in datastructures – a particularly useful feature when describing circuits containing feedback. Observable sharing is added to the language by providing immutable reference cells, together with a reference equality test. In the first part of the paper we present the problem and motivate the addition of observable sharing.

A problem with *observable sharing* is that it is not a conservative extension of a pure functional language. It is a "side effect" – albeit in a limited form – for which the semantic implications are not immediately apparent. This means that the addition of such a feature risks the loss of many of the desirable semantic features of the host language. O'Donnell [O'D93] considered a form of observable sharing (Lisp-style pointer equality `eq`) in precisely the same context (i.e., the manipulation of hardware descriptions) and dismissed the idea thus:

> " This ⟨pointer equality predicate⟩ is a hack that breaks referential transparency, destroying much of the advantages of using a functional language in the first place."

But how much is actually "destroyed" by this construct? In the second part of this paper we show – for our more constrained version of pointer equality – that in practice almost nothing is lost.

We formally define the semantics of the language extensions and investigate their semantic implications. The semantics is an extension to a call-by-need abstract machine which faithfully reflects the amount of sharing in typical Haskell implementations.

Not all the laws of pure functional programming are sound in this extension. The classic law of beta-reduction for lazy functional programs, which we could represent as: $\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N = N[{}^{M}\!/_{x}] \quad (x \notin M)$ does *not* hold in the theory. However, since this law could duplicate an arbitrary amount of computation (via the duplication of the sub-expression M, it has been proposed that this law is not appropriate for a language like Haskell [AFM^{+}95], and that more restrictive laws should be adopted. Indeed most Haskell compilers (and most Haskell programmers?) do not apply such arbitrary transformations – for efficiency reasons they are careful not to change the amount of sharing (the internal graph structure) in programs. This is because all Haskell implemetations use a *call-by-need* parameter passing mechanism, whereby the argument to a function in a given call is evaluated at most once.

We develop the theory of operational equivalence for our language, and demonstrate that the extended language has a rich equational theory, containing, for example, all the laws of Ariola et al's call-by-*need* lambda calculus [AFM^{+}95].

2 Functional Hardware Description

We deal with the description of synchronous hardware circuits in which the behaviour of a circuit and also its components can be modelled as *functions* from streams of inputs to streams of outputs. The description is realised using an embedded language in the pure functional language Haskell. There are good motivations in literature for being able to use higher-order functions, polymorphism and laziness to describe hardware [She85, O'D96, CLM98, BCSS98].

Describing Circuits The approach of modelling circuits as functions on streams was taken as early as in the days of μFP [She85], and later modernised in systems like Hydra [O'D96] and Hawk [CLM98]. The following introduction to functional circuit description owes much to the description in [O'D93].

Here are some examples of primitive circuit components modelled as functions. We assume the existence of a datatype `Signal`, which represents an input, output or internal wire in a circuit.

```
inv   :: Signal -> Signal    and :: Signal -> Signal -> Signal
latch :: Signal -> Signal    xor :: Signal -> Signal -> Signal
```

We can put these components together in the normal way we compose functions; by abstraction, application, and local naming. Here are two examples of circuits. One consists of just an and-gate and an xor-gate, which is used as a component in the other.

```
halfAdd a b   = (xor a b, and a b)
fullAdd a b c = let  (s1, c1) = halfAdd a b
                     (s2, c2) = halfAdd s1 c in  (s2, xor c1 c2)
```

We use local naming of results of subcomponents using a `let` expression. The types of these terms are:

```
halfAdd :: Signal -> Signal           -> (Signal, Signal)
fullAdd :: Signal -> Signal -> Signal -> (Signal, Signal)
```

Here is a third example of a circuit. It consists of an inverter and a latch, put together with a loop, also called *feedback*. The result is a circuit that toggles its output.

```
toggle :: Signal
toggle = let  output = inv (latch output) in  output
```

Note how we express the loop; by naming the wire and using it recursively.

Simulating Circuits By interpreting the type `Signal` as streams of bits, and the primitive components as functions on these streams, we can run, or *simulate* circuit descriptions with concrete input.

Here is a possible instantiation, where we model streams by Haskell's lazy lists.

```
type Signal = [Bool] -- possibly infinite
inv   bs = map not bs       and as bs = zipWith (&&) as bs
latch bs = False : bs       xor as bs = zipWith (/=) as bs
```

We can simulate a circuit by applying it to inputs. The result of evaluating `fullAdd [False,True] [True,True] [True,True]` is `[(False,True),(True, True)]`, while the result of `toggle` is `[True,False,True,False,True,` ...

As parameters we provide lists or streams of inputs and as result we get a stream of outputs. Note that the toggle circuit does not take any parameter and results in an infinite stream of outputs. The ability to both specify and execute (and perform other operations) hardware as a functional program is a claimed strength of the approach.

Generating Netlists Simulating a circuit is not enough. If we want to implement it, for example on an FPGA, or prove properties about it, we need to generate a *netlist* of the circuit. This is a description of the all components of the circuit, and how they are connected.

We can reach this goal by *symbolic evaluation.* This means that we supply variables as inputs to a circuit rather than concrete values, and construct an expression representing the circuit. In order to do this, we have to reinterpret the `Signal` type and its operations.

A first try might be along the following lines. A signal is either a variable name (a wire), or the result of a component which has been supplied with its input signals.

```
type Signal = Var String | Comp String [Signal]
inv   b = Comp "inv"   [b]     and a b = Comp "and" [a, b]
latch b = Comp "latch" [b]     xor a b = Comp "xor" [a, b]
```

Now, we can for example symbolically evaluate `halfAdd (Var "a") (Var "b")`

```
(Comp "xor" [Var "a", Var "b"], Comp "and" [Var "a", Var "b"])
```

And, similarly a full adder. But what happens when we try to evaluate `toggle`?

```
Comp "inv" [Comp "latch" [Comp "inv" [Comp "latch" ...
```

Since the `Signal` datatype is essentially a tree, and the toggle circuit contains a cycle, the result is an infinite structure. This is of course not usable as a symbolic description in an implementation. We get an infinite data structure representing a finite circuit.

We encounter a similar problem when we provide inputs to the a circuit which are themselves output wires of another circuit. The `Signal` type is a tree, which means that when a result is used twice, is has to be copied. This shows that trees are inappropriate for modelling circuits, because physically, circuits have a richer graph-like structure.

2.1 Previous Solutions

One possible solution, proposed by O'Donnell [O'D93], is to give every use of component a unique tag, explicitly. The signal datatype is then still a tree, but when we then traverse that tree, we can keep track of what tags we have already encountered, and thus avoid cycles and detect sharing.

In order to do this, we have to change the signal datatype slightly by adding a *tag* to every use of a component, for example as follows.

```
data Signal = Var String | Comp Tag String [Signal]
```

When we define a circuit, we have to explicitly label every component with a unique tag. O'Donnell then introduces some syntactic sugar for making it easier for the programmer to do this.

Though presented as "the first real solution to the problem of generating netlists from executable circuit specifications [...] in a functional language", it is awkward to use. A particular weakness of the abstraction is that it does not enforce that two components with the same tag are actually identical; there is nothing to stop the programmer from mistakenly introducing the same tag on different components.

But if explicit tagging is not the desired solution, why not let some underlying machinery *guarantee* that all the tags are unique? *Monads* are a standard approach for such problems (see e.g., [Wad92]). In functional programming, a monad is a data structure that can abstract from an underlying computation model. A very common monad is the *state monad*, which threads a changing piece of state through a computation. We can use such a state monad to generate fresh tags for the signal datatype. This monadic approach is taken in Lava [BCSS98].

Introducing a monad implies that the types of the primitive components and circuit descriptions become *monadic*, that is, their result type becomes monadic. A big disadvantage of this approach is not only that we must change the types, but also the syntax. We can no longer use normal function abstraction, local naming and recursion anymore, we have to express this using monadic operators. All this turns out to be very inconvenient for the programmer.

What we are looking for is a solution that does *not* require a change in the natural circuit description style of using local naming and recursion, but allows us to detect sharing and loops in a description from *within* the language.

3 Proposed Solution

The core of the problem is: a description of a circuit is basically a graph, but we cannot observe the sharing of the nodes from within the program. The solution we propose is to make the graph structure of a program *observable*, by adding a new language construct.

Objects with Identity The idea is that we want the weakest extension that is still powerful enough to observe if two given objects have actually previously been created as one and the same object.

The reason for wanting as weak an extension as possible is that we want to retain as many semantic properties from the original language as possible. This is not just for the benefit of the programmer – it is important because compilers make use of semantic properties of programs to perform program transformations, and because we do not want to write our own compiler to implement this extension.

Since we know in advance what kind of objects we will compare in this way, we choose to be explicit about this at *creation* time of the object that we might end up comparing. In fact, one can view the objects as *non-updatable references.* We can create them, compare them for equality, and dereference them.

Here is the interface we provide to the references. We introduce an abstract type `Ref`, with the following operators:

```
type Ref a = ...                      ref   :: a -> Ref a
(<=>) :: Ref a -> Ref a -> Bool       deref :: Ref a -> a
```

The following two examples show how we can use the new constructs to detect sharing: (i) **let** `x = undefined` **in** `(`**let** `r = ref x` **in** `r <=> r)`
(ii) **let** `x = undefined` **in** `ref x <=> ref x`

In (i) we create one reference, and compare it with itself, which yields `True`. In (ii), we create two *different* references to the same variable, and so the comparison yields `False`.

Thus, we have made a *non conservative extension* to the language; previously it was not possible to distinguish between a shared expression and two different instances of the same expression. We call the extension *observable sharing.* We give a formal description of the semantics in section 4.

3.1 Back to Circuits

How can we use this extension to help us to symbolically evaluate circuits? Let us take a look at the following two circuits.

```
circ1 = let  output = latch output in  output
circ2 = let  output = latch (latch output) in  output
```

In Haskell's denotational semantics, these two circuits are identified, since `circ2` is just a recursive unfolding of `circ1`. But we would like these descriptions to represent different circuits; `circ1` has one latch and a loop, where as `circ2` has two latches and a loop. If the signal type includes a reference, we could compare the identities of the latch components and conclude that in `circ1` all latches are identical, where as in `circ2` we have two *different* latches.

We can now modify the signal datatype in such a way that the creation of identities happens transparently to the programmer.

```
data Signal = Var String  | Comp (Ref (String, [Signal]))
comp name args = Comp (ref (name, args))

inv b   = comp "inv"   [b]     and a b = comp "and"   [a, b]
latch b = comp "latch" [b]     xor a b = comp "xor"   [a, b]
```

In this way, a circuit like `toggle` still creates a cyclic structure, but it is now possible to define a function which *observes* this cyclicity and therefore terminates when generating a netlist for the circuit.

3.2 Other Possible Solutions

We briefly discuss two other solutions, both of which more or less well known extensions to functional programming languages.

Pointer Equality The language is extended with an operator `(>=<) :: a -> a -> Bool` that investigates if two expressions are *pointer equal*, that is, they refer to the same bindings.

In our extension, we basically provide pointer equality in a more controlled way; you can only perform it on references, not on expressions of any type. This means we can implement our references using a certain kind of pointer equality. The other way around is not possible however, which shows that our extension is weaker.

Gensym The language is extended with a new type `Sym` of abstract symbols with equality, and an operator that generates fresh such symbols, `gensym`. It is possible to define `gensym` in terms of our `Ref`s, and also the other way around. With the reference approach however, by get an important law by *definition*, which is: `r1 <=> r2` $= \mathit{True} \;\Rightarrow$ `deref r1` $=$ `deref r2`

4 The Semantic Theory

In this section we formally define the operational semantics of observable sharing, and study the induced notion of operational equivalence. For the technical development we work with a de-sugared core language based on an untyped lambda calculus with recursive lets and structured data.

The language of terms, Λ_{ref} is given by the following grammar[1]:

$$L, M, N ::= x \mid \lambda x.M \mid M\,x \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mid \mathsf{ref}\,x \mid \mathsf{deref}\,M \mid M \rightleftharpoons N$$

Note that we work with a restricted syntax in which the arguments in function applications and the arguments to constructors are always variables (c.f, [PJPS96, PJS98, Lau93, Ses97]. It is trivial to translate programs into this syntax by the introduction of let bindings for all non-variable arguments.

The set of *values*, Val $\subseteq \Lambda_{\mathsf{ref}}$, ranged over by V and W are the lambda-expressions $\lambda x.M$. We will write $\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N$ as a shorthand for $\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N$ where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (i.e., all lets are potentially recursive).

The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

4.1 The Abstract Machine

The semantics for the standard part of the language presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [Ses97]. Following [MS99], we believe an abstract machine semantics is well suited as the basis for studying operational equivalence.

Transitions in this machine are defined over configurations consisting of (i) a *heap*, containing a set of bindings, (ii) the expression currently being evaluated,

[1] In the full version of the paper we also include constructors and a case expression, as well as a strict sequential composition operator.

and (iii) a *stack*, representing the actions that will be performed on the result of the current expression.

There are a number of possible ways to represent references in such a machine. One straightforward possibility is to use a global reference-environment, in which evaluation of the ref operation creates a fresh reference to its argument. We present an equivalent but syntactically more economical version. Instead of reference environment, references are represented by a new (abstract) constructor (i.e. a constructor which is not part of Λ_{ref}), which we denote by $\underline{\mathsf{ref}}$.

Let $\Lambda_{\underline{\mathsf{ref}}} \overset{\text{def}}{=} \Lambda_{\mathsf{ref}} \cup \{\underline{\mathsf{ref}}\, x \mid x \in \mathit{Var}\}$, and $\mathrm{Val}_{\underline{\mathsf{ref}}} \overset{\text{def}}{=} \mathrm{Val} \cup \{\underline{\mathsf{ref}}\, x \mid x \in \mathit{Var}\}$. We write $\langle\, \Gamma,\ M,\ S\,\rangle$ for the abstract machine configuration with heap Γ, expression $M \in \Lambda_{\underline{\mathsf{ref}}}$, and stack S. A *heap* is a set of bindings from variables to terms of $\Lambda_{\underline{\mathsf{ref}}}$; we denote the empty heap by $\emptyset$, and the addition of a group of bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$.

A stack is a list of stack elements. The stack written $b : S$ will denote the a stack S with b pushed on the top. The empty stack is denoted by ϵ, and the concatenation of two stacks S and T by ST (where S is on top of T). Stack elements are either:

- a variable x, representing the argument to a function,
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap,
- a pending reference equality-test of the form $(\rightleftharpoons M)$, or $(\underline{\mathsf{ref}}\, x \rightleftharpoons)$,
- a dereference deref, indicating that the reference which is produced by the current computation should be dereferenced.

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. Since we cannot have more than one binding occurrence of a variable, a configuration is deemed *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle\, \Gamma,\ M,\ S\,\rangle$ to be closed, any free variables in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$.

For sets of variables P and Q we will write $P \perp Q$ to mean that P and Q are disjoint, *i.e.*, $P \cap Q = \emptyset$. The free variables of a term M will be denoted $\mathsf{FV}\,(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}\,(\vec{M})$. The abstract machine semantics is presented in figure 4.1; we implicitly restrict the definition to well-formed configurations. The first collection of rules are standard. The second collection of rules concern observable sharing. Rule (*RefEq*) first forces the evaluation of the left argument, and (*Ref1*) switches evaluation to the right argument; once both have been evaluated to ref constructors, variable-equality is used to implement the pointer-equality test.

4.2 Convergence, Approximation, and Equivalence

Two terms will be considered equal if they exhibit the same behaviours when used in any program context. The behaviour that we use as our test of equiv-

$$\langle\, \Gamma\{x = M\},\ x,\ S\,\rangle \to \langle\, \Gamma,\ M,\ \#x : S\,\rangle \qquad (\mathit{Lookup})$$
$$\langle\, \Gamma,\ V,\ \#x : S\,\rangle \to \langle\, \Gamma\{x = V\},\ V,\ S\,\rangle \qquad (\mathit{Update})$$
$$\langle\, \Gamma,\ M\,x,\ S\,\rangle \to \langle\, \Gamma,\ M,\ x : S\,\rangle \qquad (\mathit{Unwind})$$
$$\langle\, \Gamma,\ \lambda x.M,\ y : S\,\rangle \to \langle\, \Gamma,\ M[{}^{y}\!/\!_{x}],\ S\,\rangle \qquad (\mathit{Subst})$$
$$\langle\, \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\ S\,\rangle \to \langle\, \Gamma\{\vec{x} = \vec{M}\},\ N,\ S\,\rangle \quad \vec{x} \perp \mathrm{dom}(\Gamma, S) \qquad (\mathit{Letrec})$$

$$\langle\, \Gamma,\ \mathsf{ref}\,M,\ S\,\rangle \to \langle\, \Gamma\{x = M\},\ \underline{\mathsf{ref}}\,x,\ S\,\rangle \quad x \notin \mathrm{dom}(\Gamma, S) \qquad (\mathit{Ref})$$
$$\langle\, \Gamma,\ \mathsf{deref}\,M,\ S\,\rangle \to \langle\, \Gamma,\ M,\ \mathsf{deref}\ : S\,\rangle \qquad (\mathit{Deref1})$$
$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,x,\ \mathsf{deref}\ : S\,\rangle \to \langle\, \Gamma,\ x,\ S\,\rangle \qquad (\mathit{Deref2})$$
$$\langle\, \Gamma,\ M \rightleftharpoons N,\ S\,\rangle \to \langle\, \Gamma,\ M,\ (\rightleftharpoons N) : S\,\rangle \qquad (\mathit{RefEq})$$
$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,x,\ (\rightleftharpoons N) : S\,\rangle \to \langle\, \Gamma,\ N,\ (\underline{\mathsf{ref}}\,x \rightleftharpoons) : S\,\rangle \qquad (\mathit{Ref1})$$
$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,y,\ (\underline{\mathsf{ref}}\,x \rightleftharpoons) : S\,\rangle \to \langle\, \Gamma,\ b,\ S\,\rangle \quad b = \begin{cases} \mathsf{true} & \text{if } x = y \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad (\mathit{Ref2})$$

Fig. 1. Abstract machine semantics

alence is simply termination. Termination behaviour is formalised by a convergence predicate:

Definition 4.1 (Convergence) *A closed configuration* $\langle\, \Gamma,\ M,\ S\,\rangle$ converges*, written* $\langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow}$*, if there exists heap* Δ *and value* V *such that*

$$\langle\, \Gamma,\ M,\ S\,\rangle \to^* \langle\, \Delta,\ V,\ \epsilon\,\rangle.$$

We will also write $M{\Downarrow}$, identifying closed M with the initial configuration $\langle\, \emptyset,\ M,\ \epsilon\,\rangle$. Closed configurations which do not converge are of four types: they either (i) reduce indefinitely, or get stuck because of (ii) a type error, (iii) a case expression with an incomplete set of alternatives, or (iv) a *black-hole* (a self-dependent expression as in $\mathsf{let}\ x = x\ \mathsf{in}\ x$). All non-converging closed configurations will be semantically identified.

Let $\mathbb{C}$, $\mathbb{D}$ range over *contexts* – terms containing zero or more occurrences of a *hole*, $[\cdot]$ in the place where an arbitrary subterm might occur. Let $\mathbb{C}[M]$ denote the result of filling all the holes in $\mathbb{C}$ with the term M, possibly causing free variables in M to become bound.

Definition 4.2 (Operational Approximation) *We say that* M operationally approximates N*, written* $M \precsim N$*, if for all* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are closed,* $\mathbb{C}[M]{\Downarrow}$ *implies* $\mathbb{C}[N]{\Downarrow}$*.*

We say that M and N are *operationally equivalent*, written $M \cong N$, when $M \precsim N$ and $N \precsim M$. Note that equivalence is a non-trivial equivalence relation. Below we present a sample of basic laws of equivalence. In the statement of all laws, we follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables.

$$
\begin{aligned}
(\lambda x.M)\, y &\cong M[{}^{y}\!/_{x}] \\
\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] &\cong \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[V]\}\ \mathsf{in}\ \mathbb{C}[V] \\
\mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] &\cong \mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z] \\
\mathsf{let}\ \{x = z, \vec{y} = \vec{M}\}\ \mathsf{in}\ N &\cong \mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{z}\!/_{x}]\}\ \mathsf{in}\ N[{}^{z}\!/_{x}] \\
\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N &\cong N, \quad \text{if } \vec{x} \perp \mathsf{FV}(N) \\
\mathbb{C}[\mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ M] &\cong \mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[M] \\
M \rightleftharpoons N &\cong N \rightleftharpoons M
\end{aligned}
$$

Remark: The fact that the reference constructor $\underline{\mathsf{ref}}$ is abstract (not available directly in the language) is crucial to the variable-inlining properties. For example a (derivable) law like $\mathsf{let}\ \{x = z\}\ \mathsf{in}\ N \cong N[{}^{z}\!/_{x}]$ would fail if terms could contain $\underline{\mathsf{ref}}$. This failure could be disastrous in some implementations, because in effect a configuration-level analogy of this law is applied by some garbage collectors.

4.3 Proof Techniques for Equivalence

We have presented a collection of laws for approximation and equivalence – but how are they established? The definition of operational equivalence suffers from the standard problem: to prove that two terms are related requires one to examine their behaviour in *all* contexts. For this reason, it is common to seek to prove a *context lemma* [Mil77] for an operational semantics: one tries to show that to prove M operationally approximates N, one only need compare their immediate behaviour. The following context lemma simplifies the proof of many laws:

Lemma 1 (Context Lemma). *For all terms M and N, $M \sqsubseteq N$ if and only if for all Γ, S and substitutions σ, $\langle\, \Gamma,\ M\sigma,\ S\,\rangle\Downarrow$ implies $\langle\, \Gamma,\ N\sigma,\ S\,\rangle\Downarrow$*

It says that we need only consider configuration contexts of the form $\langle\, \Gamma,\ [\cdot],\ S\,\rangle$ where the hole $[\cdot]$ appears only once. The substitution σ from variables to variables is necessary here, but since laws are typically closed under such substitutions, so there is no noticeable proof burden.

The proof of the context lemma follows the same lines as the corresponding proof for the *improvement theory* for call-by-need [MS99], and it involves uniform computation arguments which are similar to the proofs of related properties for call-by-value languages with state [MT91].

In the full paper we present some key technical properties and a proof that the compiler optimisation performed after so-called strictness analysis is still sound in the presence of observable sharing.

4.4 Relation to Other Calculi

Similar languages have been considered by Odersky [Ode94] (call-by-name semantics) and Pitts and Stark [PS93] (call-by-value semantics). A reduction-calculus approach to call-by-need was introduced in [AFM$^+$95], and extended to deal with mutable state in recent work of Ariola and Sabry [AS98]. The reduction-calculi approach in general has been pioneered by Felleisen et al (e.g. [FH92]), and its advantage is that it builds on the idea of a core calculus of equivalences (generated by a confluent rewriting relation on terms); each language extension is presented as a conservative extension of the core theory. The price paid for this modularity is that the theory of equality is rather limited. The approach we have taken – studying operational equivalence – is exemplified by Mason and Talcott's work on call-by-value lambda calculi and state [MT91]. An advantage of the operational-equivalence approach is that it is a much richer theory, in which induction principles may be derived that are inexpressible in reduction calculi. Our starting point has been the call-by-need *improvement theory* introduced by Moran and Sands [MS99]. In improvement theory, the definition of operational equivalences includes an observation of the number of reduction steps to convergence. This makes sharing observable – although slightly more indirectly.

We have only scratched the surface of the existing theory. Induction principles would be useful – and also seem straightforward to adapt from [MS99]. For techniques more specific to the subtleties of references, work on parametricity properties of local names e.g., [Pit96], is likely to be relevant.

5 Conclusions

We have motivated a small extension to Haskell which provides a practical solution to a common problem when manipulating data structures representing circuits. We have presented a precise operational semantics for this extension, and investigated laws of operational approximation. We have shown that the extended language has a rich equational theory, which means that the semantics is robust with respect to program transformations which respect sharing properties.

The extension we propose is small, and turns out to be easy to add to existing Haskell compilers/interpreters in the form of an abstract data-type (a module with hidden data constructors). In fact similar functionality is already hidden away in the nonstandard libraries of many implementations.[2] A simple implementation using the Hugs-GHC library extensions is given in the full version of the paper.

The feature is likely to be useful for other embedded description languages, and we briefly consider two such applications in the full paper: writing parsers for left-recursive grammars, and an optimised representation of decision trees.

[2] www.haskell.org/implementations/

References

[AFM$^+$95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proc. POPL'95*, ACM Press, 1995.

[AS98] Z. M. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. POPL'98*, pages 62–74. ACM Press, 1998.

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP'98.* ACM Press, 1998.

[CLM98] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems.* Marstrand, Sweden, 1998.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103:235–271, 1992.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, December 1996.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, 1993.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *TCS* 4:1–22, 1977.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, ACM Press, 1999.

[MT91] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[O'D93] J. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Springer-Verlag Workshops in Computing, pages 178–194, 1993.

[O'D96] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languages in Education*, LNCS vol 1125, pages 221–234. Springer Verlag, 1996.

[Ode94] Martin Odersky. A functional theory of local names. In *POPL'94*, pages 48–59, ACM Press, 1994.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*, pages 152–163. IEEE Computer Society Press, 1996.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, 1996.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[PS93] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that create local names, or: What's new? In *MFCS'93*, LNCS vol 711, pages 122–141, Springer-Verlag, 1993.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[She85] M. Sheeran. Designing regular array architectures using higher order functions. In *FPCS'95*, LNCS vol 201, Springer Verlag, 1985.

[Wad92] P. Wadler. Monads for Functional Programming. In *Lecture notes for Marktoberdorf Summer School on Program Design Calculi*, NATO ASI Series F: Computer and systems sciences. Springer Verlag, August 1992.

Relating May and Must Testing Semantics for Discrete Timed Process Algebras

Luis Fernando Llana Díaz and David de Frutos Escrig

Dept. Sistemas Informáticos y Programación.
Universidad Complutense de Madrid.
{llana,defrutos}@eucmax.sim.ucm.es

Abstract. In this paper we prove that for timed algebras *may* testing is much stronger than it could be expected. More exactly, we prove that the *may* testing semantics is equivalent to the *must* testing semantics for a rather typical discrete timed process algebra when considering divergence-free processes. This is so, because for any adequate test we can define a dual one in such a way that a process passes the original test in the *must* sense if and only if it does not pass the dual one in the *may* sense. It is well known that in the untimed case by *may* testing we can (partially) know the possible behaviors of a process after the instant at which it diverges, which is not possible under *must* semantics. This is also the case in the timed case.
Keywords: process algebra, time, testing semantics, *must*, *may*.

1 Introduction and Related Work

Testing semantics is introduced in [DH84, Hen88] in order to have an abstract semantics induced by the operational one, which allows us to compare processes in a natural way. Besides, it is a formalization of the classical notion of testing of programs. Tests are applied to processes generating computations that either have success or fail. But as processes are non-deterministic it is possible that the same process sometimes passes a test and sometimes fails to do it. This leads us to two different families of tests associated to each process: those that sometimes are passed, and those that always are passed. Two processes are equivalent if they pass the same tests, but as we have two different ways to pass tests, we obtain two different testing semantics that are called *may* and *must* semantics. In this paper we will study testing semantics for timed process algebras, which is also the subject of the Ph.D. Thesis of the first author [Lla96]. As far as we know, there has been not too much previous work on the subject, but [HR95] is an interesting related reference.

In the untimed case the *may* semantics is just trace semantics, while the *must* semantics is more involved, and so we need acceptance trees [Hen88] in order to characterize it. But when time is introduced the *may* semantics also becomes more complex, since, as usual, we are assuming the ASAP rule, which allows us to detect by means of tests not only the actions that have been executed, but also those that could have been chosen instead. To be exact, we will prove in this paper that for non-divergent processes the *may* testing semantics is equivalent to the *must* testing semantics for a rather typical timed process algebra. This is so, because for any adequate test we can define a dual one in such a way that a process passes the original test in the *must* sense if and only if it does not pass the dual one in the *may* sense.

The situation is much more complicated in the case of non-divergent free processes. It is well known that in the untimed case by *may* testing we can (partially) know the possible behaviors of a process after the instant at which it diverges, what is not possible under *must* semantics. This is also the case in the timed case. In fact, we will present a couple of examples showing that in the general case *may* and *must* testing orderings are incomparable each other.

A related result can be found in [Sch95]; there an operational semantics for Timed *CSP* [RR88, Ree88] is presented, and it is proved that the timed failures model is fully abstract with respect to the *may* testing semantics. Another interesting result can be found in [BDS95], where they prove that the failures semantics for *TE-LOTOS* is the biggest congruence contained in the trace semantics. In our case trace semantics is far to be equal to *may* semantics, but it is interesting to observe that it is not a congruence, also for our language, in spite of the fact that we have no problems with internal actions in the context of the choice operator, like in *CCS* or *LOTOS*.

In fact, in a previous (erroneous) version of this paper we tried to prove that *may* testing equivalence was not a congruence for the parallel operator, but if we consider the biggest congruence contained in it, for non-divergent processes we just obtain the *must* equivalence. We are very grateful to Lars Jenner for pointing us the mistake in that paper, and also for communicating us that he had also obtained the same equivalence [Jen98] between *may* and *must* testing semantics for the case of Petri Nets. It would be very interesting to study the relation between both results, mainly because the way in which the semantics are defined and the equivalence result is proven are not trivially connected. More in general, it would be nice to study which are the properties that a model of real time processes has to fulfill in order that the corresponding *may* and *must* testing semantics will be dual each other.

2 Syntax and Operational Semantics

In this section we describe the syntax of the language we will consider. In order to focus on the main characteristics and problems of timed algebras, we introduce a simple timed process algebra which however contains the main operators that characterize such an algebra. More exactly, we are talking about those we consider to be the main operators of a *high level* timed process algebra. So, we do not consider *tic* actions measuring time in a explicit way. Nevertheless it is possible to translate our high level operators to a low level language containing such kind of operators, and in this way similar results to those in this paper could be obtained for a language such as the one in [HR95]. In our language, time is introduced via the prefix operator; actions must be executed at the indicated

$$\mathsf{Upd}(t,P) = \begin{cases} P & \text{if } P = \mathtt{STOP} \text{ or } P = \mathtt{DIV} \\ \mathtt{STOP} & \text{if } P = et' \,;\, P_1 \;\text{ and }\; t' < t \\ e(t'-t) \,;\, P_1 & \text{if } P = et' \,;\, P_1 \;\text{ and }\; t' \geq t \\ \mathsf{Upd}(t,P_1)\ op\ \mathsf{Upd}(t,P_2) & \text{if } P = P_1\ op\ P_2,\ \ op \in \{\Box, \sqcap, \|_A\} \\ \mathsf{Upd}(t,P_1) \setminus A & \text{if } P = P_1 \setminus A \\ \mathtt{DIV} & \text{if } P = x \\ \mathsf{Upd}(t,P_1)[\mathtt{REC}\,x.P_1/x] & \text{if } P = \mathtt{REC}\,x.P_1 \end{cases}$$

[DIV]	$\mathtt{DIV} \xrightarrow{\tau 0} \mathtt{DIV}$	
[PRE]	$et \,;\, P \xrightarrow{et} P$	
[ELI]	$P \sqcap Q \xrightarrow{\tau 0} P$	$P \sqcap Q \xrightarrow{\tau 0} Q$
[CH1]	$\dfrac{P \xrightarrow{at} P',\ \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{P \Box Q \xrightarrow{at} P'}$	$\dfrac{P \xrightarrow{at} P',\ \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{Q \Box P \xrightarrow{at} P'}$
[CH2]	$\dfrac{P \xrightarrow{\tau t} P',\ \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{P \Box Q \xrightarrow{\tau t} P' \Box \mathsf{Upd}(Q,t)}$	$\dfrac{P \xrightarrow{\tau t} P',\ \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{Q \Box P \xrightarrow{\tau t} \mathsf{Upd}(Q,t) \Box P'}$
[INT]	$\dfrac{P \xrightarrow{et} P',\quad \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{P \|_A Q \xrightarrow{et} P' \|_A \mathsf{Upd}(t,Q)}\ e \notin A$	$\dfrac{P \xrightarrow{et} P',\quad \forall t' < t:\ Q \not\xrightarrow{\tau t'}}{Q \|_A P \xrightarrow{et} \mathsf{Upd}(t,Q) \|_A P'}\ e \notin A$
[SYN]	$\dfrac{P \xrightarrow{at} P',\quad Q \xrightarrow{at} Q'}{P \|_A Q \xrightarrow{at} P' \|_A Q'}\ a \in A$	
[HD1]	$\dfrac{P \xrightarrow{et} P',\quad \forall t' < t, a \in A:\ P \not\xrightarrow{at'}}{P \setminus A \xrightarrow{et} P' \setminus A}\ e \notin A$	
[HD2]	$\dfrac{P \xrightarrow{at} P',\quad \forall t' < t, a' \in A:\ P \not\xrightarrow{a't'}}{P \setminus A \xrightarrow{\tau t} P' \setminus A}\ a \in A$	
[REC]	$\mathtt{REC}\,x.P \xrightarrow{\tau 0} P[\mathtt{REC}\,x.P/x]$	

Table 1. Operational Semantics.

time, and we will consider a discrete time domain $\mathcal{T}$. We consider a finite set of actions Act, an internal event $\tau \notin Act$, and the set of events $\mathcal{E} = Act \cup \{\tau\}$. We denote by $\mathcal{P}$ the set of terms generated by the following grammar:

$$P ::= \mathtt{STOP} \mid \mathtt{DIV} \mid et \,;\, P \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid P \|_A Q \mid P \setminus A \mid x \mid \mathtt{REC}\,x.P,$$

where x is a variable process, $A \subseteq Act$, $e \in \mathcal{E}$, and $t \in \mathcal{T}$. A process is a closed term generated by the previous grammar. We denote the set of processes by $\mathcal{CP}$.

In order to define the operational semantics of the language we need an auxiliary function $\mathsf{Upd}(t,P)$, which represents the pass of t units of time on the process P. This function is defined in Table 1. Looking at the operational semantics, we observe that the function $\mathsf{Upd}(P,t)$ is only used when $P \not\xrightarrow{\tau t'}$ for $t' < t$; so, the way $\mathsf{Upd}(\tau t' \,;\, P, t)$ is defined when $t' < t$ is not important, and it is only included for completeness.

The operational semantics of $\mathcal{CP}$ is given by the relation $\xrightarrow{(\cdot)} \subseteq \mathcal{CP} \times (\mathcal{E} \times \mathcal{T}) \times \mathcal{CP}$ defined by the rules in Table 1. The intuitive meaning of $P \xrightarrow{et} Q$ is that P executes event e at time t to become Q. Time is always relative to the previous executed action. So rule **[PRE]** indicates that the only event that process $et ; P$ can execute is e and the time when it is executed is t. The negative premises in the rules are included to ensure the urgency of internal actions. Note that non-guarded recursive processes execute infinitely many internal actions in a row, all of them at local time 0, just like process `DIV`. We say that such processes are *divergent*.

Since some rules have negative premises we have to provide a way to guarantee that the generated transition system is consistent. This is achieved by defining a *stratification*, as detailed in [Gro93]. We consider the following function

$$f(P \xrightarrow{et} Q) = \textit{Number of operators in } P$$

that is not difficult to check that is indeed a stratification.

The main characteristics of the operational semantics considered are *urgency* and *finitely branching*. In fact, the results of this paper can be also obtained for any process algebra whose operational semantics fulfills these two properties. First, internal actions are urgent: $P \xrightarrow{\tau t} P' \;\Rightarrow\; P \overset{et'}{\not\longrightarrow} \; \forall e \in \mathcal{E}$ and $t' > t$. Nevertheless internal actions have no greater priority than observable actions to be executed at the same time (although, as in the untimed case, the execution of the internal actions that are offered at the same time that them could preclude, in a nondeterministic way, the execution of those observable actions). Finally, observable actions to be executed before any internal action are not affected by the existence of later internal actions.

Another important fact is that this operational semantics is *finitely branching*. In an untimed process algebra this property could be set as: *for a given process P there exist a finitely many number of transitions that P can execute.* This property is also satisfied by the timed process algebra that we consider, but, as in the future we desire to extend the results in this paper to more general timed process algebras, we have modified this property in the adequate way. For instance, if we would allow intervals in the prefix operator, as in $a[0..\infty]; P$, then the above property would be no longer true. Thus we consider instead the following one: *Given* $e \in \mathcal{E}$ *and* $t \in \mathcal{T}$, *the set* $\{P' \mid P \xrightarrow{et} P'\}$ *is finite.*

To conclude this section we define the set of timed actions that a process P can execute, $\mathsf{TA}(P) = \{at \mid a \in Act,\ \exists P' :\ P \xrightarrow{at} P'\}$.

3 Testing Semantics

In this section we define the testing semantics induced by the operational semantics above. Tests are just finite processes[1] but defined by an extended grammar where we add a new process, `OK`, which expresses that the test has been passed. More exactly, tests are generated by the following B.N.F. expression:

$$T ::= \texttt{STOP} \mid \texttt{OK} \mid et\,;T \mid T_1 \,\square\, T_2 \mid T_1 \sqcap T_2 \mid T_1 \parallel_A T_2 \mid T \setminus a.$$

The operational semantics of tests is defined in the same way as for processes, but only adding the following rule for the test `OK`[2]: **[OK]** $\quad \texttt{OK} \xrightarrow{\texttt{OK}} \texttt{STOP}$. Finally, we define the composition of a test and a process: $P \mid T = (P \parallel_{Act} T) \setminus Act$.

[1] We could also allow recursive processes as tests, but they are not allowed, since this would not add any additional power.

[2] To be exact, we should extend the definition of the operational semantics for processes and tests to mixed terms defining their composition, since these mixed terms

Definition 1. *Given a computation of* $P \mid T$

$$P \mid T = P_1 \mid T_1 \xrightarrow{\tau t_1} P_2 \mid T_2 \cdots P_k \mid T_k \xrightarrow{\tau t_k} P_{k+1} \mid T_{k+1} \cdots$$

we say that it is Complete *if it is finite and blocked (no more steps are allowed), or infinite. It is* Successful *if there exists some k such that* $T_k \xrightarrow{\texttt{OK}}$.

Now we can give the basic definitions of *may* and *must* test passing.

Definition 2.

– *We say that P* must pass *the test T (P must T) iff any complete computation of $P \mid T$ is successful.*
– *We say that P* may pass *the test T (P may T) iff there exists a successful computation of $P \mid T$.*
– *We write $P \sqsubseteq_{\text{must}} Q$ iff whenever P must T we also have Q must T.*
– *We write $P \sqsubseteq_{\text{may}} Q$ iff whenever P may T we also have Q may T.*
– *Finally, we write $P \simeq_{\text{may}} Q$ when $P \sqsubseteq_{\text{may}} Q$ and $Q \sqsubseteq_{\text{may}} P$, and similarly for the* must *case.*

4 States, b-Traces, and Barbs

In this section we will recall some definitions and results introduced in [LdN96]. In order to characterize the testing semantics we will consider some special kind of sets of timed actions which we call *states*. They represent any of the possible *local configurations* of a process. In a *state* we have the set of timed actions that are offered, and the time, if any, at which the process becomes divergent. We consider that a process is divergent if it can execute in a row an infinite number of internal actions, all of them at time 0. Note that divergent processes only *must pass* trivial tests, that is, those initially offering the action OK. In order to capture divergence, we have introduced a new element $\Omega \notin Act$ that represents undefinition.

Definition 3. *We say that $A \subseteq (Act \cup \{\Omega\}) \times \mathcal{T}$ is a* state *if*

– *There is at most a single $\Omega t \in A$, i.e.,* $\Omega t,\ \Omega t' \in A \ \Rightarrow\ t = t'$.
– *If $\Omega t \in A$ then t is the maximum time in A, i.e.,* $\Omega t,\ at' \in A \ \Rightarrow\ t' < t$.

We will denote by $\mathcal{ST}$ the set of states.

Therefore, a state contains a set of timed actions. If a process P is in a state A and $at \in A$ then we have that P can execute action a at time t. When $\Omega t \in A$ we have that the process would become divergent at (local) time t, if it has not executed before any of the actions in the state.

are neither processes nor tests, but since this extension is immediate we have preferred to omit this formal definition.

Example 1. Let $P = (a0\,;a1\,;\texttt{STOP}) \sqcap \big((b1\,;a0\,;\texttt{STOP}) \,\square\, (c0\,;\texttt{STOP})\big)$. As indicated by Definition 5, it has (initially) two states: $\{a0\}$ and $\{b1, c0\}$. If we consider the process $Q = P \,\square\, \tau 1\ ;\texttt{DIV}$, it has as states $\{a0, \Omega 1\}$ and $\{c0, \Omega 1\}$. Note that action b at time 1 is not included in the second state. This is so, because states are used to characterize the *must* semantics and we can easily check that $(b1\,;\texttt{STOP}) \,\square\, (\tau 1\,;\texttt{DIV}) \simeq_{\text{must}} \tau 1\,;\texttt{DIV}$. Nevertheless, as we will see in Example 2, this action will appear in a b-trace as an executed action of this process.

Next we give some auxiliary definitions over states that will be used later:

– We define the function $\mathsf{nd}(\cdot) : \mathcal{ST} \mapsto \mathcal{T} \cup \{\infty\}$, which gives us the time at which a state becomes undefined (<u>n</u>ot <u>d</u>efined function), by $\mathsf{nd}(A) = t$ if $\Omega t \in A$ and $\mathsf{nd}(A) = \infty$ otherwise.
– For each time $t \in \mathcal{T}$, we define the *time addition* and *time restriction* operators over a set of states by $A + t = \{a(t + t') \,|\, at' \in A\}$ and $A \upharpoonright t = \{at' \,|\, at' \in A \text{ and } t' < t\}$.
– If $A \in \mathcal{ST}$, we define its set of timed actions by $\mathsf{TA}(A) = A \upharpoonright \mathsf{nd}(A) = A \cap \{at \,|\, at \in A,\ a \neq \Omega\}$.
– If $A \subseteq \mathcal{ST}$ and $t \in \mathcal{T}$, we write $A < t$ (resp. $A \leq t$) iff for all $at' \in A$ we have $t' < t$ (resp. $t' \leq t$).

A barb is a generalization of an acceptance [Hen88]. Additional information must be included in order to record the actions that the process offers *before* any action has been executed. First we introduce the concept of b-trace, which is a generalization of the notion of trace. A b-trace, bs, is a sequence $A_1 a_1 t_1 A_2 a_2 t_2 \cdots A_n a_n t_n$ that represents the execution of the sequence of timed actions $a_1 t_1 a_2 t_2 \cdots a_n t_n$ in such a way that after the execution of each prefix $a_1 t_1 \cdots a_{i-1} t_{i-1}$ the timed actions in A_i were offered (but not taken) before accepting $a_i t_i$. Then a barb is a b-trace followed by a final state, that represents the reached configuration of the process after executing the b-trace. The time values that appear in barbs and b-traces are relative to the previous executed action.

Definition 4.

– b-traces *are finite sequences, $bs = A_1 a_1 t_1 \cdots A_n a_n t_n$, where $n \geq 0$, $a_i Act$, $t_i \in \mathcal{T}$, $A_i \subseteq Act \times \mathcal{T}$, and if $a't' \in A_i$ then $t' < t_i$. We take* $\mathsf{length}(b) = n$*; when $n = 0$ we have the empty b-trace denoted by ϵ.*
– *A* barb *b is a sequence $b = bs \cdot A$ where bs is a b-trace and A is a state. We will represent the barb $\epsilon \cdot A$ simply by A, and so we can consider states as (initial) barbs.*

The states of a process P are computed from its complete computations of internal actions. For any computation we record the actions that were offered *before* the execution of the next internal action. For divergent computations we also record the *time of divergence*.

Definition 5. *For a process P, the set $\mathcal{A}(P)$ is the set of states $A \in \mathcal{ST}$ that are generated from the* complete *(initial) computations of internal actions of P as described below:*

– *Each infinite computation* $P = P_1 \xrightarrow{\tau t_1} P_2 \xrightarrow{\tau t_2} \cdots P_k \xrightarrow{\tau t_k} P_{k+1} \cdots$ *generates the state* $A \in \mathcal{A}(P)$ *given by*

$$A = \bigcup_{i \in \mathbb{N}} \big((\mathsf{TA}(P_i) \upharpoonright t_i) + t^i\big) \cup \begin{cases} \{\Omega t\} & \textit{if } t = \sum_{i=1}^{\infty} t_i < \infty,^3 \\ \varnothing & \textit{otherwise.} \end{cases}$$

– *Each finite blocked computation* $P = P_1 \xrightarrow{\tau t_1} P_2 \xrightarrow{\tau t_2} \cdots P_{n-1} \xrightarrow{\tau t_{n-1}} P_n$ *generates the state* $(\mathsf{TA}(P_n) + t^n) \cup \bigcup_{i=1}^{n} \big((\mathsf{TA}(P_i) \upharpoonright t_i) + t^i\big) \in \mathcal{A}(P)$.

where, in both cases, we take $t^i = \sum_{j=1}^{i-1} t_j$.

In order to define the b-traces of a process we introduce the following notation: Let $t \in T$, $bs = A_1 a_1 t_1 \cdot bs_1$ be a b-trace, and $A \subseteq Act \times \mathcal{T}$ be a set of timed actions such that $A < t$, then we take $(A, t) \sqcup bs = (A \cup (A_1 + t))a(t_1 + t) \cdot bs_1$.

Definition 6. *Let* P*,* P' *be processes and* bs *a b-trace. We define the relation* $P \overset{bs}{\Longrightarrow} P'$ *as follows:*

– $P \overset{\epsilon}{\Longrightarrow} P$.
– *If* $P \xrightarrow{\tau t} P_1$*, and* $P_1 \overset{bs'}{\Longrightarrow} P'$ *with* $bs' \neq \epsilon$ *then* $P \xRightarrow{(\mathsf{TA}(P) \upharpoonright t, t) \sqcup bs'} P'$.
– *If* $P \xrightarrow{at} P_1$*, and* $P_1 \overset{bs'}{\Longrightarrow} P'$ *then* $P \xRightarrow{(\mathsf{TA}(P) \upharpoonright t) at \cdot bs'} P'$.

We define set of barbs of P *by* $\mathsf{Barb}(P) = \{bs \cdot A \mid P \overset{bs}{\Longrightarrow} Q \text{ and } A \in \mathcal{A}(Q)\}$.

States, barbs and b-traces are closely related. The states of a process are its initial barbs, those whose b-trace is empty. If $P \xRightarrow{bs \cdot Aat} Q$ then there exists a state A' such that $bs \cdot A' \in \mathsf{Barb}(P)$ and $A' \upharpoonright t = A$. Finally, if $bs \cdot A \in \mathsf{Barb}(P)$ and $at \in A$ then there exists a process Q such that $P \xRightarrow{bs \cdot (A \upharpoonright t) at} Q$

Example 2. Let us consider the processes P and Q introduced in Example 1. The barbs of P are $\{a0\}$, $\{b1, c0\}$, $\varnothing a0\{a1\}$, $\varnothing a0 \varnothing a1 \varnothing$, $\varnothing c0 \varnothing$, $\{c0\}b1\{a0\}$, and $\{c0\}b1\varnothing a0\varnothing$. The barbs of Q are those of P removing $\{b1, c0\}$ and adding $\{\Omega 1, c0\}$. The fact that $\{c0\}b1\varnothing a0\varnothing \in \mathsf{Barb}(P)$ indicates the that P can execute the b-trace $\{c0\}b1\varnothing a0$ and it reaches a state that offers $\varnothing$, i.e., the process deadlocks. The execution of the b-trace $\{c0\}b1\varnothing a0$ indicates that the process can execute the action b at time 1 and then, immediately, the action a (local time 0, but global time 1); but before executing action b the process could also execute the action c at time 0. If we know that this b-trace has been executed, we can conclude that c was not offered at time 0 by the environment. Note that $\{c0\}b1\{a0\} \in \mathsf{Barb}(Q)$, in spite of the fact that $\{c0, b1\} \notin \mathsf{Barb}(Q)$.

We will use barbs and b-traces to characterize the testing semantics. This will be done by defining a preorder between sets of barbs and another one between sets of b-traces. In order to define them, we need the following ordering relations between b-traces and barbs:

[3] Note that $\sum_{i=1}^{\infty} t_i < \infty$ iff there exists some n_0 such that $t_i = 0$ for all $i \geq n_0$.

Definition 7.

- *We define the relation* $\ll$ [4] *between b-traces as the least relation that satisfies: 1.* $\epsilon \ll \epsilon$, *2. If* $bs' \ll bs$ *and* $A' \subseteq A$ *then* $A'at \cdot bs' \ll Aat \cdot bs$.
- *We define the relation* $\ll$ *between barbs as the least relation that satisfies:*
  1. *If* bs, bs' *are b-traces such that* $bs' \ll bs$, *and* A, A' *are states such that* $\mathsf{nd}(A') \le \mathsf{nd}(A)$ *and* $\mathsf{TA}(A') \subseteq A$, *then* $bs' \cdot A' \ll bs \cdot A$.
  2. *If* A' *is a state,* $b = A_1 a_1 t_1 \cdot b'$ *is a barb such that* $\mathsf{nd}(A') \le t_1$ *and* $\mathsf{TA}(A') \subseteq A_1$, *and* $bs' \ll bs$ *then* $bs' \cdot A' \ll bs \cdot (A_1 a_1 t_1 \cdot b')$.

Intuitively, a b-trace bs is *worse* than another one bs', if the actions that appear in both b-traces are the same, and the intermediate sets A_i that appear in bs are smaller than those A'_i appearing in bs'. For barbs, we must notice that whenever a process is in an undefined state, which means $t = \mathsf{nd}(A) < \infty$, it *m*ust pass no test *after* that time. Barbs and b-traces are introduced to characterize the testing semantics. As shown in [LdN96], to characterize the *must* testing semantics it is enough to extend the preorder above to sets of barbs and b-traces, as follows:

Definition 8. *Let* B_1 *and* B_2 *be sets of barbs, we say that* $B_1 \ll B_2$ *iff for any* $b_2 \in B_2$ *there exists* $b_1 \in B_1$ *such that* $b_1 \ll b_2$.

The preorder $\ll$ can be used to characterize that induced by the *must* testing semantics, which means that we can prove the following *Must Characterization Theorem* (proved in detail in [LdN96]): $P \sqsubseteq_{\text{must}} Q \iff \mathsf{Barb}(P) \ll \mathsf{Barb}(Q)$. In the following sections we will characterize the *may* testing semantics.

5 Divergence Free Processes

First, the relationship between *may* and *must* testing semantics will be studied in the case when the involved processes are divergence free. We will show that, rather surprisely, in this case, both relations are symmetric each other:

$$P \sqsubseteq_{\text{must}} Q \quad \iff \quad Q \sqsubseteq_{\text{may}} P$$

Even more, we will show that *may* testing can be viewed as the *dual* relation of *must* testing, in the sense that we can find a family of *standard tests* characterizing *must* semantics, such that if we define the *dual* of a test, in a very simple and natural way (to be precised later), we have that a process passes a test of the family in the *must* sense iff it does not pass its *dual test* in the *may* sense.

Intuitively, a process P is *divergent*, and then we write $P \Uparrow$, if P can execute in a row infinitely many internal actions, all of them at (local) time 0. We say that P is *divergence free* when no execution of P leads us to a divergent process P'. For instance, the process $P = a1;\mathtt{STOP}$ is divergence free, while $Q = a1;\mathtt{DIV}$ is not, because of the existence of the computation $Q \xrightarrow{a1} \mathtt{DIV}$. We could formally define divergence freedom directly from the operational semantics, but since this definition is a little cumbersome, we present instead the following equivalent characterization in terms of barbs:

[4] Note that the symbol $\ll$ is overloaded, since it is used for both b-traces and barbs.

Definition 9. *We say that a process P is* divergence free *iff for each barb $bs \cdot A \in \mathsf{Barb}(P)$ we have $\mathsf{nd}(A) = \infty$.*

Next we will prove the left to right arrow: $P \sqsubseteq_{\text{must}} Q \;\Rightarrow\; Q \sqsubseteq_{\text{may}} P$. We will use the following characterization of the *must* testing ordering, that we have proved in [LdN96]: $P \sqsubseteq_{\text{must}} Q \iff \mathsf{Barb}(P) \ll \mathsf{Barb}(Q)$. So it is enough to prove:

Proposition 1. $\mathsf{Barb}(Q) \ll \mathsf{Barb}(P) \quad \Rightarrow \quad P \sqsubseteq_{\text{may}} Q$.

Proof. (sketch) Let T be a test such that P may T; then there exist some process P' and some test T' such that $T' \xrightarrow{\mathtt{OK}}$ and there exists a computation from $P \,|\, T$ to $P' \,|\, T'$. So, there exist P'' and T'' and a couple of b-traces $bs = A_1 a_1 t_1 \cdots A_n a_n t_n$ and $bs' = A_1' a_1 t_1 \cdots A_n' a_n t_n$ such that $P \xRightarrow{bs} P''$, $T \xRightarrow{bs'} T''$, $A_i \cap A_i' = \varnothing$, and there exists a computation from $P \,|\, T$ to $P'' \,|\, T''$ and another one from $P'' \,|\, T''$ to $P' \,|\, T'$.

Now from the computations of P'' and T'' to P' and T' we get two states $A \in \mathcal{A}\big(P''\big)$ and $A' \in \mathcal{A}\big(T''\big)$ satisfying $A \cap A' = \varnothing$.

Since $\mathsf{Barb}(Q) \ll \mathsf{Barb}(P)$ and P and Q are divergence free, there exist some barb $b'' = bs'' \cdot A'' \in \mathsf{Barb}(Q)$ with $bs'' = A_1'' a_1 t_1 \cdots A_n'' a_n t_n$ and some processes Q' and Q'' such that $A_i'' \subseteq A_i$, $A'' \subseteq A$, $Q \xRightarrow{bs''} Q''$ and there is a computation from Q'' to Q', that generates the state A'' by applying Definition 5. So we have $A_i'' \cap A_i' = \varnothing$ and $A'' \cap A' = \varnothing$, so we have a computation from $Q \,|\, T$ to $Q'' \,|\, T''$. Since Q is divergence free, we also obtain a computation from $Q'' \,|\, T''$ to $Q' \,|\, T'$.

Next we prove the right to left side of the equivalence. For it, we introduce a new family of *standard tests* characterizing *must* semantics, which means that whenever we have $P \not\sqsubseteq_{\text{must}} Q$ we can find some test T in the family such that P must T but Q m$\!\!\!/$ust T. These tests are similar, but not exactly the same, that those with the same name and property that we have used in [LdN96]. Although in order to relate $\sqsubseteq_{\text{may}}$ and $\sqsubseteq_{\text{must}}$ we could also use here those tests, we have preferred to use instead the new family, because by considering the corresponding *dual tests* we can directly obtain that relationship, thus emphasizing the duality between *must* and *may* passing of tests. The tests that constitute the new family of *standard tests* are those obtained by applying the following definition:

Definition 10. *Given a barb b and a set of barbs B such that there is no barb $b' \in B$ such that $b' \ll b$, we say that a test T is* well formed with respect to B and b *when it can be derived by applying the following rules:*

- *If $b = A$, we take any finite set $A_1 \subseteq Act \times \mathcal{T}$ such that for any $A' \in B$ we have $A_1 \cap A' \neq \varnothing$. Then, taking*
  $$T_2 = \Big(\Box_{at \in A_1} at \,;\mathtt{OK}\Big) \,\Box\, \tau t \,;\mathtt{STOP}\;, \quad \text{where } t > \max\{t' \mid at' \in A_1\}$$
  we have that $T = T_1 \,\Box\, T_2$ is well formed with respect to B and b.
- *If $b = Aat \cdot b_1$, we consider any finite set $A_1 \subseteq Act \times \mathcal{T}$ with $A_1 \cap A = \varnothing$ and such that any barb $A'at \cdot b_1' \in B$ either satisfies $A' \cap A_1 \neq \varnothing$ or $b_1' \not\ll b_1$. When $A_1 \neq \varnothing$, we consider the test*
  $$T_2 = \Box_{at \in A_1 \setminus A} at \,;\mathtt{OK}.$$

Besides, taking the set of barbs $B_1 = \{b' \mid A' \subseteq A$ and $A'at \cdot b' \in B\}$, when $B_1 \neq \varnothing$, we can take as T_1 any well formed test with respect to B_1 and b_1. Then we have that

$$T = \begin{cases} T_1 \,\Box\, T_2 \,\Box\, it\,;\mathtt{OK} & \text{if } A_1 \neq \varnothing,\ B_1 \neq \varnothing \\ T_1 & \text{if } A_1 \neq \varnothing,\ B_1 = \varnothing \\ T_2 \,\Box\, it\,;\mathtt{OK} & \text{if } A_1 = \varnothing,\ B_1 \neq \varnothing \end{cases}$$

is well formed test with respect to B and b.

Given an arbitrary set of barbs B and a barb b, it is possible that there is no well formed test T with respect to B and b, because the finite set A_1 required in the first part of the definition might not exist. But, as the operational semantics is finitely branching, for any $B = \mathsf{Barb}(P)$ and $b \in \mathsf{Barb}(Q)$ such that there is no $b' \in B$ with $b' \ll b$, there is some well formed test T with respect to B and b. Finally, dual tests are defined as expected:

Definition 11. *Let T be a test, we define its dual test, T^*, by interchanging the tailing occurrences of* `STOP` *and* `OK` *in T.*

The well formed tests and their duals satisfy the following:

Proposition 2. *Let B be a set of barbs a b a barb such that and there is no $b' \in B$ such that $b' \ll b$. Let us consider T a* well formed test with respect to B and b*, then:*

$$\begin{aligned} b \in \mathsf{Barb}(Q) &\Rightarrow Q \ \cancel{must}\ T \text{ and } Q \ may\ T^*. \\ B = \mathsf{Barb}(P) &\Rightarrow P \ must\ T \text{ and } P \ \cancel{may}\ T^*. \end{aligned}$$

Proposition 3. $P \sqsubseteq_{\text{may}} Q \quad \Rightarrow \quad Q \sqsubseteq_{\text{must}} P$.

Proof. Let us suppose that $Q \not\sqsubseteq_{\text{must}} P$, then there must exist some $b \in \mathsf{Barb}(P)$ such that there is no $b' \in \mathsf{Barb}(Q)$ verifying $b' \ll b$. Then we take a well formed test T with respect to $\mathsf{Barb}(Q)$and b such that we have Q must T and P $\cancel{\text{must}}$ T. Then by applying the previous proposition, we have P may T^* and Q $\cancel{\text{may}}$ T^*. This contradicts our hypothesis, $P \sqsubseteq_{\text{may}} Q$.

Finally, as a consequence of Propositions 1 and 3, we obtain the *may testing characterization theorem* for divergence-free processes that we were looking for:

Theorem 1. $P \sqsubseteq_{\text{may}} Q \quad \Longleftrightarrow \quad Q \sqsubseteq_{\text{must}} P$

6 Processes with Divergences

Once we have studied the relation $\sqsubseteq_{\text{may}}$ for non divergent processes we proceed to study it in the general case. Since in the untimed case we already had $\simeq_{\text{must}} \neq \simeq_{\text{may}}$ when divergences appear, we could expect that in the untimed case we would have the same result. This is indeed the case, as the following example shows1 :

Example 3. Let us consider the processes $P = \mathtt{DIV}$ and $Q = \mathtt{DIV} \sqcap a1\,;\mathtt{STOP}$. It is easy to show that P and Q are equivalent under *must* testing, i.e., $P \sqsubseteq_{\text{must}} Q$ and $Q \sqsubseteq_{\text{must}} P$; but, on the other hand, we have $Q \not\sqsubseteq_{\text{may}} P$.

We also have $\simeq_{\text{must}} \not\supseteq \simeq_{\text{may}}$, what means that we really need divergence freedom in order to prove Proposition 3. To show it, let us consider

$$P = ((a0\,;\mathtt{STOP}) \,\Box\, (\tau 2\,;\mathtt{DIV})) \sqcap ((a0\,;\mathtt{STOP}) \,\Box\, (b0\,;\mathtt{STOP}) \,\Box\, (\tau 1\,;\mathtt{DIV})) \quad \text{and}$$
$$Q = ((a0\,;\mathtt{STOP}) \,\Box\, (\tau 2\,;\mathtt{DIV})) \sqcap ((a0\,;\mathtt{STOP}) \,\Box\, (b0\,;\mathtt{STOP}) \,\Box\, (\tau 2\,;\mathtt{DIV})).$$

It is not difficult to show, by using Theorem 2, that under *may* testing semantics both processes are equivalent, i.e., $P \simeq_{\text{may}} Q$; but we have $Q \not\sqsubseteq_{\text{must}} P$, and so $P \not\simeq_{\text{must}} Q$.

In order to characterize $\sqsubseteq_{\text{may}}$ in the general case, we introduce the following

Definition 12. *Let $bs \cdot A$ and $bs' \cdot A'$ be barbs, we say $bs' \cdot A' \ll_{\text{may}} bs \cdot A$ iff $bs' \ll bs$, $\mathsf{nd}(A') \geq \mathsf{nd}(A)$, and $A' \upharpoonright \mathsf{nd}(A) \subseteq A$. Then we define the relation $\ll_{\text{may}}$ between sets of barbs, by saying $B_1 \ll_{\text{may}} B_2$ iff for any $b_1 \in B_1$ there exists $b_2 \in B_2$ such that $b_2 \ll_{\text{may}} b_1$.*

The rest of the section is devoted to prove the *May Characterization Theorem* for the general case: $P \sqsubseteq_{\text{may}} Q \iff \mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q)$. It is an immediate consequence of Propositions 4 and 5 below. In the following, $\mathsf{t_{bs}}(bs)$ stands for the *duration* of bs, defined by taking $\mathsf{t_{bs}}(\epsilon) = 0$ and $\mathsf{t_{bs}}(A_1 a_1 t_1 \cdot bs_1) = t_1 + \mathsf{t_{bs}}(bs_1)$. To prove the left to right arrow of the theorem, we need a special kind of tests:

Definition 13. *Let b be a barb, we inductively define the test $T(b)$ as follows:*

$$T(\epsilon \cdot A) = \tau t\,; \mathtt{OK} \,\Box \underset{a't' \notin A,\ t'<t}{\Box} a't'\,; \mathtt{STOP} \qquad \big(t = \mathsf{nd}(A)\big)$$
$$T(A'a't' \cdot b, t) = a't'\,; T(b, a, t) \,\Box \underset{a''t'' \notin A',\ t''<t}{\Box} a''t''\,; \mathtt{STOP}$$

For these tests it is easy to check the following:

Lemma 1. *P may $T(b)$ iff there exists $b' \in \mathsf{Barb}(P)$ such that $b' \ll_{\text{may}} b$.*

Proposition 4. *$P \sqsubseteq_{\text{may}} Q \;\Rightarrow\; \mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q)$.*

Proof. Let us consider $b = bs \cdot A \in \mathsf{Barb}(P)$. Then we have P may $T(b)$. As $P \sqsubseteq_{\text{may}} Q$ we also have Q may $T(b)$, and by applying the previous lemma we get the desired result.

Proposition 5. *$\mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q) \;\Rightarrow\; P \sqsubseteq_{\text{may}} Q$.*

Proof. (sketch) The proof of this proposition is very similar to that of Proposition 1. The only difference is the way that the adequate barb $b'' = bs'' \cdot A'' \in \mathsf{Barb}(Q)$ is found. To do it in this case we have to use the fact that $P \ll_{\text{may}} Q$. From this fact we can also conclude that this barb verifies $bs'' \ll bs$, $A'' \upharpoonright \mathsf{nd}(A) \subseteq A$ and $\mathsf{nd}(A'') \geq \mathsf{nd}(A)$. Then for the corresponding processes Q' and Q'' we obtain a successful computation from $Q \mid T$ to $Q' \mid T'$.

Finally we have the desired result:

Theorem 2. *$P \sqsubseteq_{\text{may}} Q \iff \mathsf{Barb}(P) \ll_{\text{may}} \mathsf{Barb}(Q)$.*

7 Conclusions and Discussion

In this paper we have defined and characterized *may* testing semantics for timed process algebras. We have distinguished the cases when the involved processes are divergence free or not. When considering divergence free processes we have proved that the *may* testing semantics is the inverse preorder of *must* testing semantics. Moreover, for a representative class of tests we have define the notion of *dual test.* This class of tests is powerful enough to distinguish non-related processes, and for any test T in that class it holds that P must T if and only if P m$\not{a}$y T^*, where T^* is the dual test of T. So we can conclude that, in some sense, the *may* testing semantics is dual to the *must* testing semantics. The problems that appear when considering non-divergence free processes are similar to those appearing in the untimed case.

It would be interesting to compare in detail our results with those in [Sch95]. The time model used there is continuous ($\mathbb{R}$ in fact), but this point deserves no special attention. In fact it, some people working on this area claim that continuous time is *more general* than discrete time. If that were the case, any result obtained for a reasonable continuous timed process algebra could be transfered to the *associated* discrete timed algebra. Applying this *argument* we could conclude from our results and those in [Sch95] that *must* testing semantics is equivalent to failures semantics. In fact, we conjecture that this is indeed the case, but this cannot be concluded just by applying this naive argument.

Reasonable continuous and discrete timed models are not trivially comparable. For instance, process $at;P$ cannot be modeled in Schneider's framework. The candidate to *simulate* in that model such a process would be $\texttt{WAIT}\ t; (a \rightarrow P \overset{0}{\triangleright} \texttt{STOP})$. But in this way we do not obtain an exact translation, since the meaning (using our syntax) of the latter term is the same as that of $a0\,; P \sqcap \tau 0\,; \texttt{STOP}$. The reason why this happens is that, in Schneider's model or any other reasonable continuous time model, when a process rejects an action at time 0, it must also do it at some instant $\epsilon > 0$. This naturally reflects the continuous character of real numbers. As a consequence, we cannot assert any property related to a concrete instant of time. Therefore, when comparing continuous and discrete timed models much more care is needed.

References

[BDS95] J. Bryans, J. Davies, and S. Schneider. Towards a denotational semantics for ET-LOTOS. In *CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*, pages 249–263. Springer-Verlag, 1995.

[DH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Gro93] J. F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.

[Jen98] Lars Jenner. Further studies on time testing of concurrent systems. Technical report, Institut für Informatik, Universitat Augsburg, 1998.

[LdN96] L. F. Llana-Díaz, D. de Frutos, and M. Núñez. Testing semantics for urgent process algebras. In *Third AMAST Workshop in Real Time Programming*, pages 33–46, 1996.

[Lla96] L. F. Llana-Díaz. *Jugando con el Tiempo.* PhD thesis, Universidad Complutense de Madrid, 1996.
Available in `http://dalila.sip.ucm.es/miembros/luis/ps/tesis.ps.gz`.

[Ree88] G. M. Reed. *A Uniform Mathematical Theory for Real-Time Distributed Computing.* PhD thesis, Oxford University, 1988.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[Sch95] S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

A Framework for Formal Reasoning about Open Distributed Systems[*]

Lars-åke Fredlund and Dilian Gurov

Swedish Institute of Computer Science,
Box 1263, SE-164 29 Kista, Sweden,
fred|dilian@sics.se

Abstract. We present a framework for formal reasoning about the behaviour of distributed programs implementing open distributed systems (ODSs). The framework is based on the following key ingredients: a specification language based on the μ-calculus, a hierarchical transitional semantics of the implementation language used, a judgment format allowing parametrised behavioural assertions, and a proof system for proving validity of such assertions which includes proof rules for property decomposition. This setting provides the expressive power for behavioural reasoning required by the complex open and dynamic nature of ODSs. The utility of the approach is illustrated on a prototypical ODS.

1 Introduction

For a few years now, the Formal Design Techniques group at the Swedish Institute of Computer Science has pursued a programme aimed at enabling formal verification of complex open distributed systems (ODSs) through program code verification. While previous work by the group has been predominantly directed towards establishing the mathematical machinery [5], basic tool support [3], and performing case studies [2], the present paper focuses on methodological aspects by motivating the chosen verification framework and by showing on an example proof how suitable this framework is in practice for formal reasoning about the behaviour of ODSs.

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Large-scale open distributed systems, for instance in telecom applications, must accommodate complex functionality such as dynamic addition of new components, modification of interconnection structure, and replacement of existing components without affecting overall system behaviour adversely. To this effect it is important that component interfaces are clearly defined, and that systems can be put together relying only on component behaviour along these interfaces. That is, behaviour specification,

[*] Work partially supported by the Computer Science Laboratory of Ericsson Utvecklings AB, Stockholm, the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre, and a Swedish Foundation for Strategic Research Junior Individual Grant.

and hence verification, needs to be *parametric* on subcomponents. But almost all prevailing approaches to verification of concurrent and distributed systems rely on the assumption that process networks are static, or can safely be approximated as such, as this assumption opens up for the possibility of bounding the space of global system states. Clearly such assumptions square poorly with the dynamic and parametric nature of open distributed systems.

The decision to focus on verification of actual program code rather than addressing the easier task of verifying specifications comes from the observation that still, after all these years of advocating formalised specifications as a means to improve the quality of products, in industry today only rarely does one find such formalised specifications.

We summarise the framework in Section 2 as it has developed throughout the project, and then illustrate its merits in Section 3 by focusing on a prototypical distributed systems example where a set data structure is implemented through the coordination of a dynamically changing number of processes. The example is programmed in the Erlang language [1], a functional programming language with support for distribution and concurrency, that is nowadays used in numerous telecommunication products developed by the Ericsson corporation. To illustrate the verification method we formulate and sketch a proof of a key property of the set implementation.

2 Verification of Open Distributed Systems

First we examine the characteristics of programming platforms for open distributed systems, and from this description derive requirements on the formal machinery necessary to permit verification of open distributed systems.

2.1 Programming Platforms for ODSs

Programming platforms provide the necessary functionality for open distributed systems. To name but a few services typically provided:

1. The basic building blocks that can execute concurrently (processes and/or threads, concurrent objects).
2. Facilities for dynamically creating new executing entities.
3. Means for coordination of, and communication between, concurrently executing entities. For example through semaphores, a shared memory, remote method calls, or asynchronous message passing.
4. Support for implicitly or explicitly grouping executing entities into more complex structures such as process groups, rings of processes or hypercubes.
5. Support for fault detection and fault recovery.

Like large software systems in general, ODSs are usually built from libraries of *components*. These ideally use encapsulation to provide clean *interfaces* to the components to prevent their improper use.

2.2 A Framework for Formal Reasoning about ODS Behaviour

Semantics of ODSs. To reason in a formal fashion about the behaviour of an ODS, a formal semantics of the design language in which the system is described is needed. This can be done in different styles, depending on the intended style of reasoning. Our methodology is mainly tailored to *operational semantics*, although other formal notions of behaviour are derivable in our framework, supporting reasoning in different flavours. Operational semantics are usually presented by transition rules involving labelled transitions between structured states [11]. A natural approach to handling the different conceptual layers of entities in the language, supporting modular (i.e. compositional) reasoning, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. This approach will be illustrated in the example of the Erlang programming language in Section 2.3.

Specification Language. Reasoning about complex systems requires *compositional reasoning*, i.e. the capability to reduce arguments about the behaviour of compound entities to arguments about the behaviours of its parts. To support compositional reasoning, a specification language should capture the labelled transitions at each layer of the transitional semantics. Poly-modal logic is particularly suitable for the task, employing box and diamond *modalities* labelled by the transition labels: a structured state s satisfies formula $\langle\alpha\rangle\Phi$ if there is an α-derivative of s (i.e. a state s' such that $s \xrightarrow{\alpha} s'$ is a valid labelled transition) satisfying Φ, while s satisfies $[\alpha]\Phi$ if all α-derivatives of s (if any) satisfy Φ. Additionally, *state predicates* are needed to capture the "local", unobservable characteristics of structured states, such as e.g. the value of a local variable. The presence of recursion on different layers requires also the specification language to be recursive. Adding recursion in the form of least and greatest fixed-points to the modalities described above results in a powerful specification language, broadly known as the *μ-calculus* [10, 8]. Roughly speaking, least fixed-point formulas $\mu X.\phi$ express eventuality properties, while greatest fixed-point formulas $\nu X.\phi$ express invariant properties. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed.

Parametricity. As explained above, reasoning about open systems requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen-style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of assertions. A judgment is deemed *valid* if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $x : \Psi \vdash P(x) : \Phi$

states that object P has property Φ provided the parameter x of P satisfies property Ψ.

Compositionality. Reducing an argument about the behaviour of compound entities to arguments about the behaviours of its parts can be achieved through parametricity: We can relativise an assertion $P[Q/x] : \Phi$ about the compound object P to a certain property Ψ of its component Q by considering Q as a parameter for which property Ψ is assumed, provided we can show that Q indeed satisfies the assumed property Ψ. Technically this can be achieved through a *term-cut* proof rule of the shape:

$$\frac{\Gamma \vdash Q : \Psi, \Delta \qquad \Gamma, x : \Psi \vdash P : \Phi, \Delta}{\Gamma \vdash P[Q/x] : \Phi, \Delta}$$

Recursion. When reasoning about programs in the presence of recursion on different layers, one traditionally relies on different forms of *inductive reasoning*, such as mathematical induction, complete induction and well-founded induction. Of these, the latter is the most general one. Through a sophisticated mechanism for generalized-loop detection, fixed-point approximation, and discharge, our proof method supports well-founded inductive reasoning, as well as proofs by *co-induction* [9] which is needed when reasoning about entities of non-well-founded nature such as infinite streams. Recursion on any layer like data, functions, and processes is treated uniformly in this framework.

The mechanism itself is presented and studied in detail in [5]; here we only give an idea of the approach. Assume that we are to prove that repeated popping of elements from a stack must eventually fail unless interleaved with the pushing of new elements. The initial proof goal will roughly have the shape *s:stack* $\vdash$ *reppop(s):terminates*, where *stack* is the type of stack expressed as a formula describing the transitional semantics of stacks, *reppop* is a function implementing repeated popping, and *terminates* is a formula expressing termination of computation. Since the stack definition and the formulas are recursive, in the process of proof construction we will eventually reach a point where we have to prove the same termination property, but for a modified stack. In fact, this new proof goal will be an *instance* of the more general initial goal, and in this way we will have discovered a generalised loop in the proof tree. But along this loop we will have made progress in that we will have decreased the value of an ordinal approximating a least fixed-point formula describing the stack. This fact will allow the new goal to be discharged with respect to the initial goal, analogously to the way assumptions are discharged in natural deduction, thus terminating successfully the respective branch in the proof tree.

2.3 Programming ODSs in Erlang

Erlang [1] is at its core a functional programming language, extended with a notion of processes and primitives for message passing. Erlang has a small set of powerful constructs, and is therefore suitable as both a modeling as well as an implementation language for ODSs consisting of a high number of light-weight processes. It is especially suitable for telecommunications software. In contrast to most other functional programming languages, Erlang has seen heavy use in industry. In a recent project at Ericsson where a state-of-the-art high-speed ATM switch was developed [4], figures of 480 000 lines of Erlang source code have been reported, compared to 330 000 lines of C code (most of it in the form of imported protocol libraries), and approximately 5 000 lines of Java code. A frequently voiced opinion is that a chief reason for the quick development of this product, and with resulting excellent quality, is the fast code-debug-replace cycle made possible through the introduction of Erlang. Another important reason for the success of Erlang in such projects clearly are the accompanying libraries which provide support for many aspects of developing and maintaining large telecommunications applications. There is for instance support for distributed data base access, error recovery, and code replacement during runtime. A brief overview of the Erlang fragment used in this paper can be found in Appendix A.1.

Formal Semantics Our semantics for Erlang is a small-step operational one [6]. The basic message of the previous section with respect to language semantics was the desirability to mimic the conceptual view that a programmer has of a system built using Erlang in the language semantics. The semantics developed here matches closely the hierarchic structure of the Erlang language. First the Erlang expressions are provided with a semantics that does not require any notion of processes. The actions here are a computation step τ, an output $\mathit{pid}!v$, $\mathit{read}(q, v)$ for reading a value v from the queue of the process in which context the expression executes, and $f(v_1, \ldots, v_n) \leadsto v$ for calling a builtin function (like `spawn` for process spawning) with side-effects on the process level state. An example of an expression level transition rule is:

$$\mathsf{fun}_1 \; \frac{}{f(\tilde{v}) \xrightarrow{f(\tilde{v}) \leadsto v} v}$$

The transitional behaviors of Erlang systems are captured separated into two cases: (i) a single process constraining the behaviors of an Erlang expression as illustrated in the following rule for process spawning:

$$\mathsf{spawning} \; \frac{e \xrightarrow{\mathtt{spawn}(\mathit{module}, f, v) \leadsto \mathit{pid}'} e' \quad \mathit{pid}' \neq \mathit{pid}}{\mathtt{proc}\,\langle e, \mathit{pid}, q\rangle \longrightarrow \mathtt{proc}\,\left\langle e', \mathit{pid}, q\right\rangle \parallel \mathtt{proc}\,\left\langle \mathit{module} : f(v), \mathit{pid}', \mathsf{eps}\right\rangle}$$

and (ii) the (parallel) composition of two Erlang systems into a single one exemplified in the following rule for interleaving:

$$\mathsf{interleave}_0 \; \frac{s_1 \xrightarrow{\tau} {s_1}' \quad \mathit{wellformed}({s_1}' \parallel s_2)}{s_1 \parallel s_2 \xrightarrow{\tau} {s_1}' \parallel s_2}$$

where $\mathit{wellformed}(s)$ requires that process identifiers of processes in s are unique. The system actions are computation steps τ, input $\mathit{pid}?v$ and output $\mathit{pid}!v$.

2.4 Verifying ODSs in EVT

The Erlang Verification Tool (EVT for short) is a proof editing tool implementing the above described framework: providing a property specification language and an embedding of an operational semantics for Erlang, combined with a general proof system based on the classical first-order sequent calculus.

```
         F ::=  tt | ff | T = T | F /\ F | F => F | F \/ F | not F
             |  forall Var : Type . F | exists Var : Type . F
             |  λVar:Type.F | F T | T : F
             |  [Action]F | <Action>F

PredicateDef ::=  Name : PropType DefSymbol F
    PropType ::=  prop | Type -> PropType
   DefSymbol ::=  => | <= | =
```

Fig. 1. The syntax of logic formulae and definitions

The syntax of the specification logic of EVT is illustrated in Figure 1. In addition to the usual connectives of predicate logic the $\langle\alpha\rangle$F and $[\alpha]$F modalities are available with their usual meaning, defined by referring to the transition relations of the embedded operational semantics. The T : F construct expresses the proposition "T satisfies F" (or "T has type F"). In the following we will refer to a number of predefined types such as `erlangValue` (ground values), `erlangExpression` (expressions), `erlangSystem` (systems), `erlangAction` (actions ranging over computation steps `tau`, output *pid*`!`*v* and input *pid*`?`*v*), etc. In the definition of a predicate (PredicateDef) the `=>` symbol selects the greatest fixed point, the `<=` symbol the least fixed point, while `=` is for non-recursive definitions (shorthands). Recursive occurrences of predicates are only permitted under an even number of negations to ensure monotonicity.

3 Verifying a Prototypical Open Distributed System

Active data structures, i.e., collections of processes that by coordinating their activities mimic in a concurrent way some data structure, are frequently used in telecommunication software. In a previous study [2] a protocol for responding to database queries, directed to the distributed database manager Mnesia, was verified. Internally the protocol built up a ring like structure of connected processes in order to answer queries efficiently. In the current example we examine a scheme for a set implementation inspired by a set-as-process example of Hoare [7]. Here the active data structure is a linked list, but the similarities with the database query example are striking.

3.1 An Implementation of a Persistent Set

As an abstract mathematical notion, a *set* is simply a collection of objects (taken out of an universe of objects), characterized by the membership relation "$\in$": if s is an object and S is a set, then the statement $s \in S$ is either true or false. Using the membership relation, one can define sets as unions, intersections, or differences of other sets, or in other ways.

Computer scientists have also another view of sets, namely as *mutable* objects: a set, when manipulated by adding or removing elements, still keeps its "identity", e.g. through an identifier. Any data-structure for manipulating collections of objects, which does not impose an order on its elements (i.e. hides this order through its interface), can be understood as implementing a set.

The objects to be manipulated can be distributed in space, and if the objects themselves are large, it is conceivable, that we might want each object to be maintained by a separate process. A further reason for implementing a set as an active data structure is to permit concurrent access to multiple elements.

A complete implementation of a set, without a possibility to remove elements, by means of a collection of interacting processes is given in Appendix A.2, where a module `persistent_set_adt` is defined. Internally the module implements two functions - one for maintaining of single elements, and one for the empty set. A set is identified by an Erlang process identifier. When creating a new set, it initially consists of a single process executing the `empty_set` function; it is the process identifier of this process by which the set is to be identified from hence on. When an element is added, a new process is spawned off to store the element if it is not already present in the set. Internally, when a new element is added to a set, it is "pushed downwards" through the list of processes representing set elements, until it reaches the emptyset process, which spawns off another emptyset process, and becomes itself a process maintaining the new element. So, as a result, a set is implemented as a unidirectional linked collection of processes referenced by a process identifier.

To encapsulate the set against improper use, we provide a controlled interface to the set module, consisting of a function for set creation `mk_empty`, testing for membership `is_member`, addition of elements `add_element`, etc. The set creation function, for example, spawns off a process executing the `empty_set` function, and returns the process identifier of the newly spawned process. This process identifier has then to be provided as an argument to all the other interface functions. The implementation of the two set functions and the interface prevents the user of the set module from having to notice that sets are internally represented by processes, and moreover prevents direct access to any other process identifiers created internal to the linked list of processes.

Note however that any process, given knowledge of the process identifier of a persistent set, can choose to circumvent the interface functions and directly communicate (through message passing) with the set process. As we shall see in the proof such "protocol abuse" can lead to program errors.

3.2 A Persistent Set Property

To check the correctness of a persistent set implementation, we have to specify those properties of sets which we consider paramount for correct behaviour. Ideally, one would like such a specification to be complete, i.e. a system should satisfy the specification exactly when it implements such a set. Completeness, however, is usually difficult to achieve in practice, since such a specification would be very detailed and the resulting proofs could easily become unmanageably complex.

One crucial property of persistent sets is naturally that they retain any element added to them. For simplicity, we will here prove a simpler property, that once any element has been added to such a set the set will forever be non-empty. The main predicates are:

```
ag_non_empty: erlangPid -> erlangSystem -> prop =>
  \SetPid:erlangPid. \SetSys:erlangSystem.
    ((SetSys : non_empty SetPid) /\
     (SetSys : forall Alpha:erlangAction.[Alpha](ag_non_empty SetPid)));

persistently_non_empty: erlangPid -> erlangSystem -> prop =>
  \SetPid:erlangPid. \SetSys:erlangSystem.
    (((SetSys : non_empty SetPid) /\ (SetSys : ag_non_empty SetPid)) \/
      (SetSys : empty SetPid) /\ (SetSys : forall Alpha:erlangAction.
                                   [Alpha](persistently_non_empty SetPid)));
```

Intuitively the `persistently_non_empty` predicate expresses an automaton that, when applied to a process identifier `SetPid` and an Erlang system `SetSys` representing a set, checks that `empty SetPid` remains true until `non_empty SetPid` becomes true, after which `non_empty SetPid` must remain continuously true forever (definition `ag_non_empty`). Note that this is, in some respect, a challenging property since it contains both a safety part (*non-empty sets never claim to be empty*) and a liveness part (*all sets eventually answer queries whether they are empty*).

We advocate an observational approach to specification, through invocation of the interface functions, as evidenced in the definition of the `empty` predicate:

```
empty: erlangPid -> erlangSystem -> prop =
  \SetPid:erlangPid. \SetSys:erlangSystem.
    (forall Pid:erlangPid.
     ((not (Pid = SetPid)) =>
      (proc<is_empty(SetPid), Pid, eps> || SetSys : (evaluates_to Pid true))));
```

The `empty` predicate expresses that `proc<is_empty(SetPid), Pid, eps>`, an observer process, will eventually (in a finite number of steps) terminate with the value `true`, if executing concurrently with the observed set `SetSys`. For lack of space the definition of the `evaluates_to` predicate has been omitted.

3.3 A Proof Sketch

Expressed in the syntax of EVT the main proof obligation becomes:

```
prove "declare P:erlangPid in |- proc<empty_set(), P, eps> : persistently_non_empty P";
```

That is the Erlang system `proc<empty_set(), P, eps>`, an initially empty set, satisfies the `persistently_non_empty P` property. In fact we will prove a slightly stronger property:

```
Goal #0: not(add_in_queue Q) |- proc<empty_set(), P, Q> : persistently_non_empty P;
```

where the `not(add_in_queue Q)` assumption expresses that the queue `Q` does not contain an `add_element` message. This proof goal is reduced by unfolding the definition of the `persistently_non_empty` predicate, choosing to show that the set process will signal that it is empty when queried, and performing a few other trivial proof steps. There are two resulting proof goals:

```
#1: not(add_in_queue Q) |- proc<empty_set(), P, Q> : empty P
#2: not(add_in_queue Q) |- proc<empty_set(), P, Q> :
          forall Alpha:erlangAction. [Alpha](persistently_non_empty P)
```

Goal `#1` reduces to (after unfolding `empty` and rewriting):

```
not(add_in_queue Q), not(P=P') |- proc<is_empty(P), P', eps> || proc<empty_set(), P, Q> :
                                  evaluates_to P' true
```

That is, an observer process calling the interface routine `is_empty` with the set process identifier `P` as argument will eventually (in a finite number of steps) evaluate to the value `true` (meaning that the set is empty). Here the proof strategy is to symbolically “execute” the two processes together with the formula, and observe that in all possible future states the observer process terminates with `true` as the result. Note however that the assumption `not(add_in_queue Q)` is crucial due to the Erlang semantics of queue handling. If the queue `Q` contains an `add_element` message the observer process will instead return `false` as a result, since its `is_empty` message would be stored after the `add_element` message in the queue and thus be serviced only after an element is added to the set.

The second proof goal `#2` is reduced by eliminating the universal quantifier, and computing the next state under all possible types of actions. Since the process is unable to perform an output action there are two resulting goals, one which corresponds to the input of a message `V` (note the resulting queue `Q@V`) and the second a computation step (applying the `empty_set` function).

```
#3: not(add_in_queue Q) |- proc<empty_set(), P, Q@V> : persistently_non_empty P
#4: not(add_in_queue Q) |- proc<receive ... end, P, Q> : persistently_non_empty P
```

Proceeding with goal `#4` either the first message to be read from the queue is `is_empty` or `is_member` (the possibility of an `add_element` message can be discarded due to the queue assumption). Handling these two new goals presents no major difficulties.

Goal `#3` is reduced by analysing the value of `V`. If it is not an `add_element` message then we can easily extend the assumption about non-emptiness of `Q`:

```
#5: not(add_in_queue Q@V) |- proc<empty_set(), P, Q@V> : persistently_non_empty P
```

Goal #5 is clearly an instance of goal #0, i.e., we can find a substitution of variables that when applied to the original goal will result in the current proof goal (the identity substitution except that it maps the queue Q to the queue Q@V). Since we have at the same time unfolded a greatest fixed point on the right hand side of the turnstile (the definition of persistently_non_empty) we are allowed to discharge the current proof goal at this point. If, on the other hand, V is an add_element message the next goal becomes:

```
#6: add_in_queue Q@V |- proc<empty_set(), P, Q@V> : persistently_non_empty P
```

At this point we cannot discharge the proof goal, since there is no substitution from the original proof goal to the current one. Instead we repeat the steps of the proof of goal #0 but taking care to show non_empty P instead of empty P. Also, we cannot discard the possibility of receiving an add_element message and the resulting goal is (after weakening out the queue assumption):

```
#7: |- proc<set(Element,mk_empty(...)),P,Q'> : ag_non_empty P
```

By repeating the above pattern of reasoning with regards to goal #7 we eventually reach the proof state:

```
#8: not(P=P') |- proc<set(Element,P'),P,Q''> || proc<empty_set,P',eps> : ag_non_empty P
```

The Erlang components of the proof states of the proof, up to the point of the spawning off of the new process, are illustrated in Figure 2.

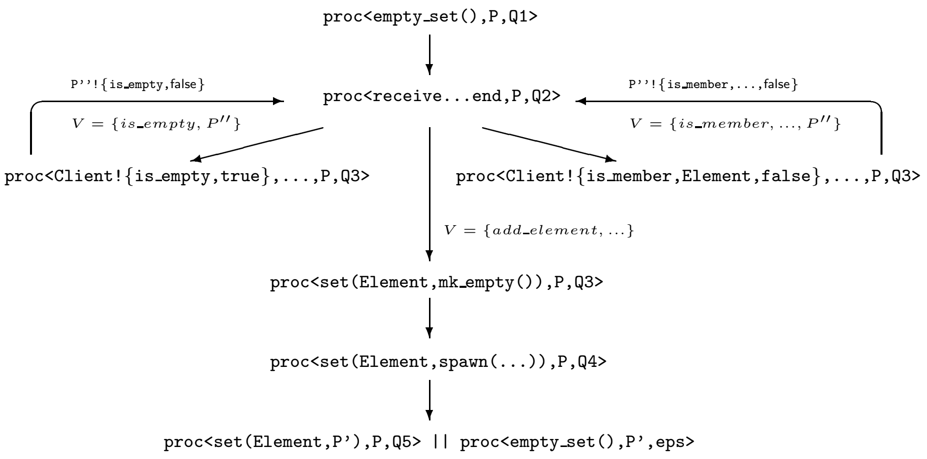

proc<empty_set(),P,Q1>
P''!{is_empty,false}
proc<receive...end,P,Q2>
P''!{is_member,...,false}
V = {is_empty, P''}
V = {is_member, ..., P''}
proc<Client!{is_empty,true},...,P,Q3>
proc<Client!{is_member,Element,false},...,P,Q3>
V = {add_element, ...}
proc<set(Element,mk_empty()),P,Q3>
proc<set(Element,spawn(...)),P,Q4>
proc<set(Element,P'),P,Q5> || proc<empty_set(),P',eps>

Fig. 2. Erlang components of initial proof states

At this point we have reached a critical point in the proof where some manual decision is required. Clearly we can repeat the above proof steps forever, never being able to discharge all proof goals, due to the possibility of spawning new processes. Instead we apply the *term-cut* proof rule, to abstract the freshly spawned processes with a formula `psi` ending up with two new proof goals:

```
#9: not(P=P') |- proc<empty_set(), P', eps> : psi P P'
#10: not(P=P'), X:psi P P' |- proc<set(Element,P'), P, Q''> || X : ag_non_empty P
```

How should we choose `psi`? The cut formula must be expressive enough to characterise the `proc<empty_set(), P', eps>` process, in the context of the second process and for the purpose of proving the formula `ag_non_empty P`. Here it turns out that the following formula is sufficient:

```
psi: erlangPid -> erlangPid -> erlangSystem -> prop =>
  \P:erlangPid . \P':erlangPid .
  (   (forall P:erlangPid . forall V:erlangValue . [P?V]((not(is_empty V)) => psi P P'))
   /\ (forall P:erlangPid . forall V:erlangValue . [P!V](not(is_empty V)))
   /\ (converges P P') /\ (foreign P) /\ (local P'));
```

Intuitively `psi` expresses:

– Whenever a new message is received, and it is not an `is_empty` message then `psi` continues to hold.
– An `is_empty` reply is never issued.
– The predicated system can only perform a finite number number of internal and output steps (definition of `converges` omitted).
– Process identifier `P` is foreign (does not belong to any process in the predicated system) and process identifier `P'` is local (definitions omitted).

The proof of goal `#9` is straightforward up to reaching the goal:

```
#11: not(P'=P'') |- proc<set(Element,P''), P', Q'''> || proc<empty_set(), P'', eps>: psi P P'
```

Here we once again apply the term-cut rule to obtain the following goals:

```
#12: not(P'=P'') |- proc<empty_set(), P'', eps> : psi P' P''
#13: Y:psi P' P'' |- proc<set(Element,P''),P',Q'''>||Y : psi P P'
```

Goal `#12` can be discharged immediately due to the fact that it is an instance of goal `#9`. Goal `#13` involves symbolically executing the `proc<set(Element,P''),P',Q'''>` process together with the (abstracted) process variable `Y`, thus generating their combined proof state space. Since both these systems generate finite proof state spaces this construction will eventually terminate. The proof of goal `#10` is highly similar to the proof of goal `#13` above, and is omitted for lack of space.

3.4 A Discussion of the Proof

The proof itself represented a serious challenge in several respects:

– The modelled system is an open one in which at any time additional set elements can be added outside of the control of the set implementation itself. The state space of the set implementation is clearly non finite state: both the number of processes and the size of message queues can potentially grow without bound.

– The queue semantics of Erlang has some curious effects with regards to an observer interacting with the set implementation. It is for instance not sufficient to consider only the program states of the set process to determine whether an observer will recognise a set to be empty or not; also the contents of the input message queue of the set process has to be taken into account.

Although the correctness of the program may at first glance appear obvious, a closer inspection of the source code through the process of proving the implementation correct revealed a number of problems.

For instance, in an earlier version of the set module the guards `pid(Client)` in the `empty_set` and `set` functions were missing. These guards serve to ensure that any received `is_empty` or `is_member` message must contain a valid process identifier. Should these guards be removed a set process will terminate due to a runtime (typing) error if, say, a message `{is_empty,21}` is sent to it.

In most languages adding such guards would not be needed since usage of the interface functions should ensure that these kinds of "typing errors" can never take place. In Erlang, in contrast, it is perfectly possible to circumvent the interface functions and communicate directly with the set implementation.

4 Conclusion

We have introduced an ambitious proof system based verification framework that enables formal reasoning about complex open distributed systems (ODSs) as programmed in real programming languages like Erlang. The proof method was illustrated on a prototypical example where a set library was implemented as a process structure, and verified with respect to a particular correctness property formulated in an expressive specification logic. Parts of the proof were checked using the Erlang Verification Tool, a proof editing tool with specific knowledge about Erlang syntax and operational semantics.

In conclusion we have clearly illustrated the great potential of the approach: we were able to verify non-trivial properties of real code in spite of difficulties such as the essentially non-finite state nature of the class of open distributed systems studied in the example. In addition the approach is promising because of its generality: we are certainly not tied for all future to the currently studied programming language (Erlang) but can, by providing alternative operational semantics, easily target other programming languages. Still numerous improvements of the framework are necessary, perhaps at the moment most importantly with respect to the interaction with the proof editing tool. We are currently forced to reason at a detail level where too many manual proof steps are required to complete proofs. How to rectify this situation by providing high-level automated proof tactics remains an area of active research.

Acknowledgements Many thanks are due to Thomas Arts at the Ericsson Computer Science Laboratory and Gennady Chugunov and Mads Dam at the Swedish Institute of Computer Science.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.
[2] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. To appear in *Proc. Formal Methods Europe'99*, 1999.
[3] T. Arts, M. Dam, L.-å. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98,* Lecture Notes in Artificial Intelligence*, vol. 1421, pp. 38–41*, 1998.
[4] S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.
[5] M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference,* H. Langmaack, A. Pnueli and W.-P. de Roever (eds.), Springer, 1536:150–185, 1998.
[6] L. Fredlund. Towards a semantics for Erlang. Unpublished manuscript*, Swedish Institute of Computer Science*, 1999.
[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
[8] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.
[9] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.
[10] D. Park. Fixpoint induction and proof of program semantics. *Machine Intelligence*, **5**:59–78, 1970.
[11] G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.

A Appendix

A.1 A Short Introduction to Erlang

An Erlang *process*, here written $\texttt{proc}\,\langle e, pid, q\rangle$, is a container for evaluation of functional expressions e, that can potentially have side effects (e.g., communicate). A process has a unique process identifier (pid) which is used to identify the recipient process in communications. Communication is always binary, with one (anonymous) party sending a message (a value) to a second party identified by its process identifier. Messages sent to a process are put in its mailbox q, queued in arriving order. The empty queue will be denoted with `eps`, and q@v is a queue composed of the subqueue q and the value v. To express the concurrent execution of two sets of processes s_1 and s_2, the syntax $s_1 \parallel s_2$ is used.

The functional sublanguage of Erlang is rather standard: atoms, integers, lists and tuples are value constructors; $e_1(\tilde{e_2})$ is a function call; e_1, e_2 is sequential composition; $\texttt{case}\ e\ \texttt{of}\ p_1[\texttt{when}\ \ e_{1g}]\texttt{->}e_1; \ldots ; p_n[\texttt{when}\ \ e_{1g}]\texttt{->}e_n\ \texttt{end}$ is matching: the value that e evaluates to is matched sequentially against patterns (values that may contain unbound variables) p_i, respecting the optional guard expressions e_{ig}. $e_1!e_2$ is sending whereas $\texttt{receive}\ m\ \texttt{end}$ inspects the process mailbox q and retrieves (and removes) the first element in q that matches any pattern in m. Once such an element v has been found, evaluation proceeds analogously to $\texttt{case}\ v\ \texttt{of}\ m$. Finally $\texttt{if}\ e_1\texttt{->}\tilde{e'_1}; \cdots e_n\texttt{->}\tilde{e'_n}\ \texttt{end}$ is sequential choice.

Expressions are interpreted relative to an environment of "user defined" function definitions $f(\tilde{p_1})$ `->`$\tilde{e_1}$; $\cdots$; $f(\tilde{p_k})$ `->`$\tilde{e_k}$. The Erlang functions are placed in *modules* using the `-module` construct. Functions not explicitly exported using the `-export` construct are unavailable outside the body of the module.

A number of builtin functions are used in the paper. self returns the pid of the current process. spawn(e_1, e_2, e_3) creates a new process, with empty mailbox, executing the expression $e_2(\tilde{e_3})$ in the context of module e_1. The pid of the new process is returned. pid(e) evaluates to `true` if e is a pid and `false` otherwise. Syntactical equality (inequality) is checked by `==` (`/=`).

A.2 The Set Erlang Module

```
-module(persistent_set_adt).
-export([mk_empty/0, is_empty/1, is_member/2, add_element/2, empty_set/0]).

empty_set () ->
    receive
        {is_empty, Client} when pid(Client) ->
            Client ! {is_empty, true}, empty_set ();
        {is_member, Element, Client} when pid(Client) ->
            Client ! {is_member, Element, false}, empty_set ();
        {add_element, Element}->
            set (Element, mk_empty ())
    end.

set (Element, Set) ->
    receive
        {is_empty, Client} when pid(Client) ->
            Client ! {is_empty, false}, set (Element, Set);
        {is_member, SomeElement, Client} when pid(Client) ->
            if
                SomeElement == Element ->
                    Client ! {is_member, SomeElement, true}, set (Element, Set);
                SomeElement /= Element ->
                    Set ! {is_member, SomeElement, Client}, set (Element, Set)
            end;
        {add_element, SomeElement} ->
            if
                SomeElement == Element ->
                    set (Element, Set);
                SomeElement /= Element ->
                    Set ! {add_element, SomeElement}, set (Element, Set)
            end
    end.

%% MODULE INTERFACE FUNCTIONS

mk_empty () -> spawn (persistent_set_adt, empty_set, []).

is_empty (Set) ->
    Set ! {is_empty, self ()},
    receive
        {is_empty, Value} -> Value
    end.

is_member (Element, Set) ->
    Set ! {is_member, Element, self ()},
    receive
        {is_member, Element, Value} -> Value
    end.

add_element (Element, Set) -> Set ! {add_element, Element}.
```

Structural Sharing and Efficient Proof-Search in Propositional Intuitionistic Logic

Didier Galmiche and Dominique Larchey-Wendling

LORIA - Université Henri Poincaré
Campus Scientifique, BP 239
Vandœuvre-lès-Nancy, France

Abstract. In this paper, we present a new system for proof-search in propositional intuitionistic logic from which an efficient implementation based on structural sharing is naturally derived. The way to solve the problem of formula duplication is not based on logical solutions but on an appropriate representation of sequents with a direct impact on sharing and therefore on the implementation. Then, the proof-search is based on a finer control of the resources and has a $\mathcal{O}(n \log n)$-space complexity. The system has the subformula property and leads to an algorithm that, for a given sequent, constructs a proof or generates a counter-model.

1 Introduction

In recent years there was a renewed interest in proof-search for constructive logics like intuitionistic logic (IL), mainly because of research in intuitionistic type theories and their relationships with programming through proof-search [7]. Nowadays, theorem provers are more often used into the formal development of software and systems and have to be integrated into various environments and large applications. Then, to provide a simple implementation of a logical calculus (and connected proof-search methods) in imperative programming languages like C or Java, we have to consider new techniques leading to the clarity of the implementation [8] but also to a good and efficient explicit management of formulae, considered here as resources. The potential support for user interfaces or for a natural parallelisation of the proof-search process could be expected.
A recent work on efficient implementation of a theorem prover for Intuitionistic Propositional Logic (IPL) has emphasised the interest of finer control and management of the formulae involved in the proof-search, with an encoding of the sequents at the implementation step [2]. But to solve the problem of duplication of some formulae we need to reconsider the global problem of proof-search in this setting, knowing that a usable proposal is necessary for a good space complexity and an efficient implementation based on resources sharing. A logical solution based on the introduction of new variables and of some adequate strategies is given in [3]. Our goal in this paper is to propose a solution at a structural level (no new formulae and only modifications of the sequents structure) including a finer management of formulae (as resources). Therefore, we define, from an

analysis of previous works, a new logical system, that has some similarities with the one of [3]. With such a system we can naturally derive, from the logical rules and a particular representation of sequents (with specific tree structures), some new rules that act on trees to perform proof-search. Moreover, these ideas can also be applied to a refutability system for IPL to generate counter-models in case of unprovability [5]. With such an approach the depth of the search tree is in both systems linearly bounded and the control on resources leads to an efficient implementation in imperative languages.

2 Proof-Search in Intuitionistic Logic

We focus on IPL for which the LJ sequent calculus is a simple formulation having the subformula property. On the other hand, it lacks termination and thus implementing this calculus involves some kind of loop-checking mechanism to ensure the termination of computation. A solution is the use of the contraction-free sequent calculus LJT [1] that has the nice property that backward search does not loop. There exists other proposals for proof-search in IL based for instance on resolution [7], constraints [9] or skolemization [6].
Let us start to recall the main characteristics of the LJT system. It is is sound and complete w.r.t. provability in LJ and a backward search does not loop and always terminates [1]. For that, the left-implication rule of LJ is replaced by four rules depending on the form of the premiss of the principal formula. Then, the treatment of the implication is becoming finer and the termination is obtained from a well-founded ordering on sequents. Moreover the contraction rule is no longer a primitive rule but is nevertheless admissible. The corresponding multi-conclusion calculus is also a good basis for a non-looping method to construct Kripke trees as counter-models for non-provable formulae of IPL [5]. It provides a calculus, called CRIP, in which refutations are inductively defined and Kripke counter-models can be extracted from these refutations. The rules of LJT are given in figure 1, X representing atomic formulae, A, B, G representing formulae and Γ being a multiset of formulae. This system provides a decision procedure in which the deductions (proofs) may be of exponential depth in the size of their endsequent. Moreover in rules like $\rightarrow$-L$_3$, some formulae of the conclusion appear twice (duplication problem). We also observe that, for instance in the $\rightarrow$-L$_3$ rule, the premises are not smaller (in the usual meaning) than the conclusion, even if they are with a multiset ordering (see [1]). The LJT prover, implemented in Prolog, is based on an analytic approach where the size of the formulae decreases but a potential big number of subformulae might appear. One one has no explicit knowledge on the formulae used during the proof-search or on the depth of proofs. Moreover, LJT does not satisfy the subformula property: for instance, in rule $\rightarrow$-L$_4$, the formula $B \rightarrow C$ is introduced in the left premise even if it is not a subformula of $(A\rightarrow B)\rightarrow C$. Some rules duplicate formulae as in rule $\rightarrow$-L$_3$ where C is duplicated and in rule $\rightarrow$-L$_4$ where B is duplicated. Therefore, the idea to share formulae naturally arises so that the duplication would be harmless in terms of space. But we have to know which are the formulae that might appear

$$\frac{}{\Gamma, X \vdash X}\ \mathrm{Id} \qquad \frac{}{\Gamma \vdash \mathsf{T}}\ \mathsf{T} \qquad \frac{}{\Gamma, \mathsf{F} \vdash G}\ \mathsf{F}$$

$$\frac{\Gamma, A, B \vdash G}{\Gamma, A \wedge B \vdash G}\ \wedge\text{-L} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-R} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-R}_1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-R}_2 \qquad \frac{\Gamma, A \vdash G \qquad \Gamma, B \vdash G}{\Gamma, A \vee B \vdash G}\ \vee\text{-L} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow\text{-R}$$

$$\frac{\Gamma, X, B \vdash G}{\Gamma, X, X \rightarrow B \vdash G}\ \rightarrow\text{-L}_1 \qquad \frac{\Gamma, A \rightarrow (B \rightarrow C) \vdash G}{\Gamma, (A \wedge B) \rightarrow C \vdash G}\ \rightarrow\text{-L}_2$$

$$\frac{\Gamma, A \rightarrow C, B \rightarrow C \vdash G}{\Gamma, (A \vee B) \rightarrow C \vdash G}\ \rightarrow\text{-L}_3 \qquad \frac{\Gamma, B \rightarrow C \vdash A \rightarrow B \qquad \Gamma, C \vdash G}{\Gamma, (A \rightarrow B) \rightarrow C \vdash G}\ \rightarrow\text{-L}_4$$

Fig. 1. The LJT calculus

during such backward proof-search. Then we have recently proposed an explicit management of the formulae (considered as resources) and a new proof-search method for LJT based on new concepts and results such as a new notion of subformula [2].

In the LG system, independently proposed in [3], the length of the deductions is linear bounded in terms of the size of the endsequent. A backward application of the rules gives rise to an $\mathcal{O}(n \log n)$-space decision algorithm for IPL. In this work, the strategy used to avoid the duplication of subformulae is different for the $\rightarrow$-L$_3$ and $\rightarrow$-L$_4$ rules: for the first one, it corresponds to add new propositional variables and to replace complex formulae by such new variables. For the second one, it forces the sequents to keep a convenient form in order to apply some transformations or reductions. In fact, in this system, the duplication problem is solved at a logical level by the introduction of new variables and the use of particular sequents [3]. But it is complicated and several implementation problems arise. Our aim, in this paper, is to propose an alternative (not logical but structural) solution based on a fine management of new structures appropriate for an efficient implementation.

Let us mention the work of Weich [10] that is based on constructive proofs of the decidability of IPL by simultaneously constructing either a proof, taking up the approach of [3] or a counter-model refining the idea of [4]. His algorithm is faster than the refutability algorithm of [5]. When restricted to provability it coincides with the one of [3] including the same logical solution for the duplication problem.

Our proposal is based on a new logical system that has some similarities with the formulations of LG dedicated to the implementation [3]. The duplication problem is in our case treated by a structural approach in which no new variable or formula is introduced and such that we only modify the structure of the sequents. It will lead to a more direct and clear implementation in imperative languages. The main characteristics coming from the use of appropriate sharing techniques are: no dynamic memory allocation, a finer control on the resources and a $\mathcal{O}(n \log n)$-space algorithm for provability. A similar approach can be applied to define and then implement an algorithm for refutability with the generation of counter-models in a same space complexity, i.e., with the depth of search tree being linearly bounded.

$$\frac{\Gamma^\star, A, B \vdash \blacksquare}{\Gamma^\star, A \land B \vdash \blacksquare}\,[\land_L] \qquad \frac{\Gamma^\star, A \vdash \blacksquare \qquad \Gamma^\star, B \vdash \blacksquare}{\Gamma^\star, A \lor B \vdash \blacksquare}\,[\lor_L]$$

$$\frac{\Gamma^\star, X, C \vdash \blacksquare}{\Gamma^\star, X, X \to C \vdash \blacksquare}\,[X{\to}] \qquad \frac{}{\Gamma, X, X^\star \to C \vdash \blacksquare}\,[X^\star{\to}]$$

$$\frac{\Gamma, A, B^\star \to C \vdash \blacksquare \qquad \Gamma^\star, C \vdash \blacksquare}{\Gamma^\star, (A \to B) \to C \vdash \blacksquare}\,[{\to}{\to}] \qquad \frac{\Gamma, A, B^\star \to C \vdash \blacksquare}{\Gamma, (A \to B)^\star \to C \vdash \blacksquare}\,[{\to}^\star{\to}]$$

$$\frac{\Gamma^\star, A \to (B \to C) \vdash \blacksquare}{\Gamma^\star, (A \land B) \to C \vdash \blacksquare}\,[\land{\to}] \qquad \frac{\Gamma, A^\star \to C \vdash \blacksquare \qquad \Gamma, B^\star \to C \vdash \blacksquare}{\Gamma, (A \land B)^\star \to C \vdash \blacksquare}\,[\land^\star{\to}]$$

$$\frac{\Gamma^\star, A \to C, B \to C \vdash \blacksquare}{\Gamma^\star, (A \lor B) \to C \vdash \blacksquare}\,[\lor{\to}] \qquad \frac{\Gamma, A^\star \to C, B \to C \vdash \blacksquare}{\Gamma, (A \lor B)^\star \to C \vdash \blacksquare}\,[\lor_l^\star{\to}]$$

$$\frac{}{\Gamma, X \vdash X}\,[\mathrm{Ax}] \qquad \frac{\Gamma, A \to C, B^\star \to C \vdash \blacksquare}{\Gamma, (A \lor B)^\star \to C \vdash \blacksquare}\,[\lor_r^\star{\to}]$$

$$\frac{\Gamma, A, B \vdash G}{\Gamma, A \land B \vdash G}\,[\land_L] \qquad \frac{\Gamma, A \vdash G \qquad \Gamma, B \vdash G}{\Gamma, A \lor B \vdash G}\,[\lor_L]$$

$$\frac{\Gamma, X, C \vdash G}{\Gamma, X, X \to C \vdash G}\,[X{\to}] \qquad \frac{\Gamma, A, B^\star \to C \vdash \blacksquare \qquad \Gamma, C \vdash G}{\Gamma, (A \to B) \to C \vdash G}\,[{\to}{\to}]$$

$$\frac{\Gamma, A \to (B \to C) \vdash G}{\Gamma, (A \land B) \to C \vdash G}\,[\land{\to}] \qquad \frac{\Gamma, A \to C, B \to C \vdash G}{\Gamma, (A \lor B) \to C \vdash G}\,[\lor{\to}]$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \land B}\,[\land_R] \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \lor B}\,[\lor_{R_l}]$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,[{\to}_R] \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \lor B}\,[\lor_{R_r}]$$

Fig. 2. The SLJ-system

3 A New System for Intuitionistic Proof-Search

We present in this section a new logical system, called SLJ, that leads to a natural use of sharing techniques on structures adapted to an efficient implementation in imperative languages. It is derived from the ideas of LJT [1] and of LG [3].

3.1 The SLJ System

The SLJ-system, that is given in figure 2, includes two kinds of sequents, like in [3]. We have the usual intuitionistic sequent $\Gamma \vdash A$ which is a multiset Γ of formulae forming the hypothesis together with a unique formula A called the conclusion. The intuitive meaning of a sequent is that the conclusion A logically follows from the hypothesis in Γ. Moreover, we have a second kind of sequent which we will call *boxed sequent*: $\Gamma, \alpha^\star \to C \vdash \blacksquare$ which intuitively means $\Gamma, \alpha \to C \vdash \alpha$. The special $\blacksquare$-notation indicates that the conclusion α is shared with the left side of an implication in the hypothesis: $\alpha \to C$. For example, in rule $[{\to}{\to}]$, B is shared and not duplicated. There is a mark $\star$ to determine which formula is concerned. In this second kind of sequents, there is exactly one marked formula on the left of an implication of the hypothesis. We will denote $\Gamma^\star$ to point out the fact that Γ contains exactly one formula of type $\alpha^\star \to C$. This will happen when this last formula is not the active one like in

rule $[X{\rightarrow}]$ for example. This idea of marking α was introduced in [3] to avoid one particular kind of duplication of formula in LJT and is also useful for our purpose. It is important to notice that when we try to prove a boxed sequent $\Gamma, \alpha^\star \rightarrow C \vdash \blacksquare$, the sequents encountered upper in the proof are always boxed: only boxed sequents appear in the proof of an initial boxed sequent.
The reader may also notice that the rules $[\wedge_L]$, $[\vee_L]$, $[X{\rightarrow}]$, $[{\rightarrow}{\rightarrow}]$, $[\wedge{\rightarrow}]$ and $[\vee{\rightarrow}]$ are duplicated. There is one version for boxed sequents and another version for standard sequents. In the case of boxed sequents, the marked (or starred) formula is not active. We have chosen not to distinguish the names because only the nature of the sequent (boxed or not) differs for each pair of rules. On the contrary, if we compare the rules $[{\rightarrow}{\rightarrow}]$ and $[{\rightarrow}^\star{\rightarrow}]$, then it appears that they are very different. In the case the marked formula is active, the rules do not have the same meaning at all.
This system is very interesting because all the duplications appearing in LJT have been removed except in the rules $[\vee{\rightarrow}]$, $[\vee_l^\star{\rightarrow}]$ and $[\vee_r^\star{\rightarrow}]$. The problem of duplication in the rule ($\rightarrow$-L$_4$) of LJT has then disappeared. We will see in section 4 that the last duplication cases can be addressed by changing the structure of sequent to represent sharing.

3.2 Kripke Models

Kripke models are a very general model representation for logics like intuitionistic or modal logics. Here, we will adopt the same notation as in [10] for Kripke trees. A *Kripke tree* is a pair $\mathcal{K} = (\mathcal{S}_\mathcal{K}, [\mathcal{K}_1, \ldots, \mathcal{K}_p])$ where $\mathcal{S}_\mathcal{K}$ is a finite set of logical variables and $[\mathcal{K}_1, \ldots, \mathcal{K}_p]$ is a finite list of Kripke trees. Moreover, we suppose that for each i, $\mathcal{S}_\mathcal{K} \subseteq \mathcal{S}_{\mathcal{K}_i}$. This monotony condition is typical to intuitionistic logic as opposed to modal logics for example. This is an inductive definition of Kripke trees and the base case is when the list $[\mathcal{K}_1, \ldots, \mathcal{K}_p]$ is empty, i.e. $p = 0$. In fact, Kripke trees are regular (oriented) trees with each node tagged with a set of variables that grows along the paths from the root to the leaves.
We then introduce the notion of *forcing*. Let $\mathcal{K} = (\mathcal{S}_\mathcal{K}, [\mathcal{K}_1, \ldots, \mathcal{K}_p])$ be a Kripke tree and F be a logical formula. We say that $\mathcal{K}$ forces F and write $\mathcal{K} \vDash F$. The inductive definition is the following:

$$\mathcal{K} \vDash X \text{ iff } X \in \mathcal{S}_\mathcal{K} \qquad \mathcal{K} \vDash A \rightarrow B \text{ iff } \mathcal{K} \vDash A \text{ implies } \mathcal{K} \vDash B \text{ and } \forall i, \mathcal{K}_i \vDash A \rightarrow B$$
$$\mathcal{K} \vDash A \wedge B \text{ iff } \mathcal{K} \vDash A \text{ and } \mathcal{K} \vDash B \qquad \mathcal{K} \vDash A \vee B \text{ iff } \mathcal{K} \vDash A \text{ or } \mathcal{K} \vDash B$$

The only difference with classical logic here is that the forcing of logical implication $A \rightarrow B$ is inherited in the sons $\mathcal{K}_i$ of $\mathcal{K}$ and so in all its subtrees. In fact, this inheritance is extended to all formulae with the following result, a proof of which can be found in [10].

Lemma 1. *If F is a logical formula and $\mathcal{K} = (\mathcal{S}_\mathcal{K}, [\mathcal{K}_1, \ldots, \mathcal{K}_p])$ is a Kripke tree forcing F, i.e. $\mathcal{K} \vDash F$, then for all i, $\mathcal{K}_i \vDash F$.*

Kripke trees are models of intuitionistic logic. Let $A_1, \ldots, A_n \vdash B$ be a sequent, we say that a Kripke tree $\mathcal{K}$ is a model of $A_1, \ldots, A_n \vdash B$ if $\mathcal{K} \vDash A_1$ and ... and $\mathcal{K} \vDash A_n$ implies $\mathcal{K} \vDash B$. We will often write $\mathcal{K} \vDash A_1, \ldots, A_n \vdash B$.

Theorem 2 (Soundness). *If $\Gamma \vdash A$ is provable in intuitionistic logic and $\mathcal{K}$ is a Kripke tree then $\mathcal{K} \vDash \Gamma \vdash A$.*

Models are very often used in a negative way to provide a synthetic argument to the unprovability of some formula or sequent. Therefore, we say that $\mathcal{K}$ is a counter-model of the sequent $A_1, \ldots, A_n \vdash B$ if $\mathcal{K} \vDash A_1$ and ... and $\mathcal{K} \vDash A_n$ and $\mathcal{K} \nvDash B$. We write $\mathcal{K} \nvDash A_1, \ldots, A_n \vdash B$. There is an constructive version of the completeness theorem, see [10] for example.

Theorem 3 (Completeness). *Let $\Gamma \vdash A$ be a sequent, then either it is intuitionistically provable, or there exists a Kripke tree $\mathcal{K}$ such that $\mathcal{K} \nvDash \Gamma \vdash A$.*

3.3 Completeness of SLJ

Before proving the theorem we have to precise how Kripke trees can model boxed sequents. $\mathcal{K}$ is a counter-model to $\Gamma, \alpha^\star \to C \vdash \blacksquare$ if $\mathcal{K}$ is a counter-model to $\Gamma, \alpha \to C \vdash \alpha$, so $\mathcal{K} \vDash \Gamma, \alpha \to C$ and $\mathcal{K} \nvDash \alpha$. But we do not know whether $\mathcal{K}$ forces C or not at the level of $\mathcal{K}$. However, if one of its sons $\mathcal{K}_i$ forces α then $\mathcal{K}_i$ also forces C. The completeness result for SLJ can be decomposed into two parts: one for boxed sequents and one for standard sequents. Here we only prove the part for boxed sequents. The other part is very similar. The reader is reminded that the following proofs are inspired from [5,10].

Theorem 4 (Soundness). *If $\Gamma, \alpha^\star \to C \vdash \blacksquare$ is a provable sequent in SLJ, then $\Gamma, \alpha \to C \vdash \alpha$ is intuitionistically provable (or provable in LJT).*

To obtain this result, it is sufficient to show that all the boxed rules[1] of SLJ are admissible. For example, let us explore the case of $[\wedge^\star \to]$. We have to prove that the following rule is admissible in IPL:

$$\frac{\Gamma, A \to C \vdash A \qquad \Gamma, B \to C \vdash B}{\Gamma, (A \wedge B) \to C \vdash A \wedge B}$$

From the premisses, we obtain $\Gamma \vdash (A \to C) \to A$ and $\Gamma \vdash (B \to C) \to B$. The sequent $(A \to C) \to A, (B \to C) \to B, (A \wedge B) \to C \vdash A \wedge B$ is also provable. By two applications of the (Cut) rule, we obtain $\Gamma, \Gamma, (A \wedge B) \to C \vdash A \wedge B$. We finally get the conclusion by application of the contraction rule. More details can be found in [3].

Theorem 5 (Completeness). *Let $\Gamma, \alpha^\star \to C \vdash \blacksquare$ be a boxed sequent, either it admits a proof in SLJ, or it admits a Kripke tree counter-model.*

The proof is done by induction on $\Gamma, \alpha \to C$, but it is not a simple induction on the size of $\Gamma, \alpha \to C$, see [1]. First, we point out the fact that some rules have the important invertibility property: any counter-model to one of the premisses is a counter-model to the conclusion. This is the case for the $[\wedge_L]$, $[\vee_L]$, $[X\to]$,

[1] Boxed rules are the rules of figure 2 for which the conclusion is a boxed sequent.

$[X^\star{\rightarrow}]$, $[{\rightarrow}^\star{\rightarrow}]$, $[\wedge{\rightarrow}]$, $[\wedge^\star{\rightarrow}]$ and $[\vee{\rightarrow}]$ rules. Then if Γ has one of the following form,

$$\begin{array}{lll} \Gamma \equiv \Delta, A \wedge B & \Gamma \equiv \Delta, A \vee B & \Gamma \equiv \Delta, X, X \rightarrow C \\ \Gamma \equiv \Delta, (A \vee B) \rightarrow C & \Gamma \equiv \Delta, (A \wedge B) \rightarrow C & \end{array}$$

one of the above rule can be applied. The induction hypothesis gives us either one proof for each premiss or else one counter-model to one of the premisses, and thus we either have a proof or a counter model of $\Gamma, \alpha^\star \rightarrow C \vdash \blacksquare$. This is also the case when $\alpha \equiv A \rightarrow B$, $\alpha \equiv A \wedge B$ or $\alpha \equiv X$ and $X \in \Gamma$. Then the problem is reduced to the following case:

$$\Gamma \equiv \begin{cases} X_1, \ldots, X_n, Y_1 \rightarrow D_1, \ldots, Y_k \rightarrow D_k, \\ (A_1 \rightarrow B_1) \rightarrow C_1, \ldots, (A_p \rightarrow B_p) \rightarrow C_p \end{cases}$$

with $\{X_1, \ldots, X_n\} \cap \{Y_1, \ldots, Y_k\} = \emptyset$. Moreover, either $\alpha \equiv A \vee B$, or $\alpha \equiv X$ and $X \notin \{X_1, \ldots, X_n, Y_1, \ldots, Y_k\}$. We will suppose that $\alpha \equiv A \vee B$ for the rest of the proof — the other case is simpler. We introduce $\Delta_i = \Gamma - \{(A_i \rightarrow B_i) \rightarrow C_i\}$. Then, by induction hypothesis, either one of the $\Delta_i, A_i, B_i^\star \rightarrow C_i, \alpha \rightarrow C \vdash \blacksquare$ is provable, or there exists a counter-model for each of them.
In the first case, suppose that $\Delta_{i_0}, A_{i_0}, B_{i_0}^\star \rightarrow C_{i_0}, \alpha \rightarrow C \vdash \blacksquare$ has a proof in SLJ. Then consider the rule

$$\frac{\Delta_{i_0}, A_{i_0}, B_{i_0}^\star \rightarrow C_{i_0}, \alpha \rightarrow C \vdash \blacksquare \qquad \Delta_{i_0}, C_{i_0}, \alpha^\star \rightarrow C \vdash \blacksquare}{\Gamma, \alpha^\star \rightarrow C \vdash \blacksquare} \; [{\rightarrow}{\rightarrow}]$$

This rule is right invertible: a counter-model of the second premiss is a counter-model of the conclusion. On the other hand, a proof of the second premiss would give a proof of $\Gamma, \alpha^\star \rightarrow C \vdash \blacksquare$. Thus, we obtain a proof or a counter-model of the conclusion.
In the second case, for each $i \in [1, p]$ let $\mathcal{K}_i$ be a counter-model of $\Delta_i, A_i, B_i^\star \rightarrow C_i, \alpha \rightarrow C \vdash \blacksquare$. Let us consider $\alpha \equiv A \vee B$.[2] By induction hypothesis, either one sequent among $\Gamma, A^\star \rightarrow C, B \rightarrow C \vdash \blacksquare$ and $\Gamma, A \rightarrow C, B^\star \rightarrow C \vdash \blacksquare$ is provable, or we have two counter-models $\mathcal{K}_A$ and $\mathcal{K}_B$. In the first subcase, we obtain a proof of $\Gamma, (A \vee B)^\star \rightarrow C \vdash \blacksquare$ applying $[\vee_l^\star{\rightarrow}]$ or $[\vee_r^\star{\rightarrow}]$. In the second subcase, we define $\mathcal{K} = (\{X_1, \ldots, X_n\}, [\mathcal{K}_1, \ldots, \mathcal{K}_p, \mathcal{K}_A, \mathcal{K}_B])$. It is only routine to check that $\mathcal{K}$ is a counter-model to $\Gamma, \alpha^\star \rightarrow C \vdash \blacksquare$. We now give some details of the proof. As $X_j \in \Delta_i$, we get $\mathcal{K}_i \vDash X_j$ and so $X_j \in \mathcal{S}_{\mathcal{K}_i}$ for any $j \in [1, n]$ and $i \in \{1, \ldots, p, A, B\}$. Thus, $\mathcal{K}$ is a Kripke tree and it is simple to check that $\mathcal{K} \vDash X_1, \ldots, X_n, Y_1 \rightarrow D_1, \ldots, Y_k \rightarrow D_k, \alpha \rightarrow C$.
As $\mathcal{K}_A \nvDash A$ and $\mathcal{K}_B \nvDash B$, we obtain $\mathcal{K} \nvDash \alpha$ and then it remains to prove that $\mathcal{K} \vDash (A_j \rightarrow B_j) \rightarrow C_j$ hold. For any $i \in \{1, \ldots, p, A, B\} - \{j\}$, $(A_j \rightarrow B_j) \rightarrow C_j$ is in the context (Δ_i or Γ) and so $\mathcal{K}_i \vDash (A_j \rightarrow B_j) \rightarrow C_j$. Why does $\mathcal{K}_j \vDash (A_j \rightarrow B_j) \rightarrow C_j$ hold? Because we have $\mathcal{K}_j \vDash A_j$ and $\mathcal{K}_j \vDash B_j \rightarrow C_j$. Moreover $\mathcal{K}_j \nvDash B_j$ and also $\mathcal{K}_j \nvDash A_j \rightarrow B_j$ and thus $\mathcal{K} \nvDash A_j \rightarrow B_j$. Finally, if $\mathcal{K} \vDash A_j \rightarrow B_j$ then $\mathcal{K} \vDash C_j$.

[2] Remember we have supposed this but a complete proof would also consider the case $\alpha \equiv X$ and $X \notin \{X_1, \ldots, X_n, Y_1, \ldots, Y_k\}$.

This completeness proof for the boxed fragment of SLJ also describes an algorithm that produces a proof or a counter-model out of a given (boxed) sequent. It can be effectively used to build counter-models.

4 Sharing Techniques

In this section, we will present an alternative approach, w.r.t. [3], to address the problem of duplication of formulae during proof-search. It is not logical but structural: only the structure of sequents is modified, no new formula or variable is introduced. Looking at the SLJ-system, we can observe that a duplication occurs when the active formula is of type $(A \vee B) \to C$, and only in this case, as for example in the rule $[\vee\!\to]$. Hudelmaier proposed a way to bypass this problem for complexity reasons: he wanted the linear size of sequent to decrease strictly while applying rules of his system and put a restriction on C that is to be an atomic formula, a variable for example. But doing this, he had to introduce a new rule in the system so that any formula C could be replaced by a variable without modifying the meaning of the sequent:

$$\frac{\Gamma, A \to X, X \to C \vdash G}{\Gamma, A \to C \vdash G}\ [X \text{ new variable}]$$

By this trick, the system assures that any duplication is done on a formula of size 1 and then has the property that the application of rules decreases the linear size of the sequent. That is why we say that the duplication problem is addressed on the logical side. We can see that the variable X represents the sharing of C between $A \to C$ and $B \to C$ by combining the two rules:

$$\frac{\dfrac{\Gamma, A \to X, B \to X, X \to C \vdash G}{\Gamma, (A \vee B) \to X, X \to C \vdash G}\ [\vee\!\to]}{\Gamma, (A \vee B) \to C \vdash G}\ [X \text{ new variable}]$$

But this method has the drawback of introducing some new formulae ($A \to X$ for example) during the proof-search. This could be a problem when we look for a resource-conscious implementation of proof-search. It may not be easy to allocate in advance all the formulae that could appear during the proof-search process. We propose an alternative approach that consists in encoding the sharing of C between $A \to C$ and $B \to C$ inside the sequent structure.

4.1 A Structural Solution

Usually, an intuitionistic sequent is a multiset (or set or list, depending on structural rules) of formulae, as hypothesis, together with a unique formula for the conclusion. Here, we will consider a new structure encoding the sharing.

Sequent structure. The left part of sequents will have the structure of a forest (i.e. list of trees). Each tree represents a compound formula where sharing is taken into account. For this, the nodes of the trees are associated to subformulae of the initial sequent which will be called *tags*. Let l be a leaf in this tree, we denote by $F_0, \ldots, F_l$ the list of formulae encountered while following the path from the root of tag F_0 to the leaf l tagged with F_l.

Each tree $\mathcal{T}$ is associated a *meaning*, which is the logical formula $\bigwedge_l F_l \to \cdots \to F_0$ where l ranges over the leaves of $\mathcal{T}$. As usual, $\to$ is right associative and $F_l \to \cdots \to F_0$ means $F_l \to (\cdots \to F_0)$. See the example in figure 3. There is an obvious map from formulae to trees $F \longmapsto \bullet F$ that associates a formula F to one-node tree whose root is tagged with F.

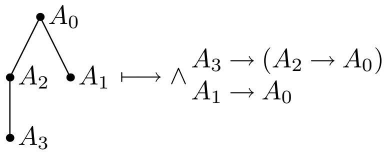


Fig. 3. Meaning function

Rules. The rules are derived from the SLJ rules and the preceding meaning maps. They act on the leaves of the trees and there are two cases: the tree is a one-node tree or else the root is not a leaf.

In the case of a one-node tree, either the formula is an implication in which case the node is rewritten as a two-nodes tree (rule $[\to_L]$ in figure 5) or we apply left conjunction or disjunction rule to obtain one or two one-node trees.

In the case the root is not a leaf, a formula $F_l \to (F_{l-1} \to \cdots \to F_0)$ corresponds to the leaf l and thus to an left implication rule of SLJ. For example, if F_l is $A \wedge B$ the leaf l is replaced by a two-nodes tree whose leaf is tagged with A and whose root is tagged with B like it appears in rule $[\wedge\to]$ of figure 5.

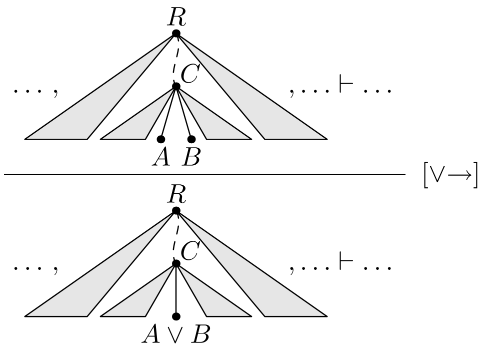


Fig. 4. Tree decomposition rule

We focus here on one of the rules that involves duplication. Considering the trees of figure 4, we see that the tree of the conclusion is associated to a formula of the shape $\cdots \wedge \big[(A \vee B) \to (C \to \cdots \to R)\big] \wedge \cdots$. Considering the rule $[\vee\to]$ of SLJ, and the fact that $\wedge$ can be exchanged with commas (,) on the left side of SLJ-sequents because of the rule $[\wedge_L]$, we can convert this formula into $\cdots \wedge \big[A \to (C \to \cdots \to R) \wedge B \to (C \to \cdots \to R)\big] \wedge \cdots$. The new formula exactly corresponds to the tree of the premise. It illustrates the method from which we derive the tree manipulation rules from SLJ-rules. We also notice how the duplication has been removed with the help of sharing. The node tagged with C is shared in the two paths leading to A and B. Finally, A and B which replace $A \vee B$ are subformulae. This fact is a general one, and all rules introduce only subformulae so that the tags of the trees are all subformulae of the formulae of the initial sequent. The it provides a subformula property (see section 4.2).

It is important to point out that the sharing version of SLJ is not a simple implementation in terms of trees. In particular, we have previously seen that proving a boxed sequent only involves boxed sequents. This is no longer the case

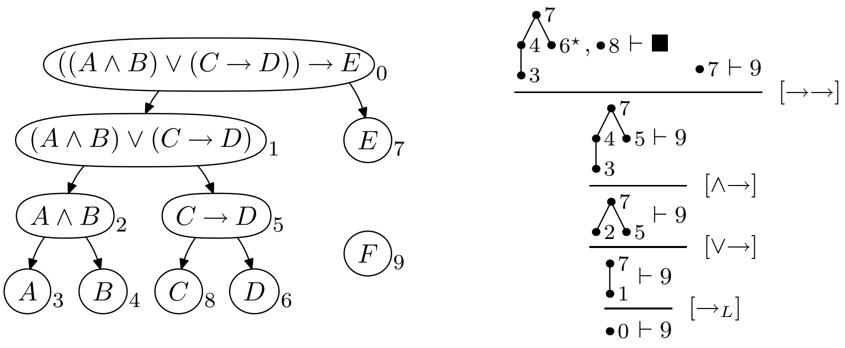


Fig. 5. An example of proof-search

in the sharing tree version. The tree version of the rule $[X\to]$ might remove a whole subtree and this one might contain the marked (or starred) formula. In this case, the premiss cannot be a boxed sequent and we must reintroduce the marked formula as the conclusion of the premiss.

An example. Considering figure 5, we aim to prove the sequent $((A \wedge B) \vee (C \to D)) \to E \vdash F$. First we build the tree of the subformulae of this sequent. The nodes are given with numbers used to label the nodes of proof trees. The proof-search tree is written bottom-up as usual. We start by translating the premiss into a forest (in this case, there is only one tree) and then we develop this forest following the rules for trees. Notice that only the leaves of the trees are concerned in those rules. Inner nodes are "asleep" until all the sons are "removed." This point has some consequences when efficient implementation is taken into account (see section 5). Moreover a marked formula $6^\star$ appears in the tree at the end of the proof-search. As for SLJ, there might be at most one marked formula in the forest, meaning that this formula is shared with the conclusion and in this case, the conclusion is boxed.

4.2 Properties of Structural Sharing

Structural sharing avoids the duplication of formulae. In fact, anytime a formula on a leaf is decomposed, it is always replaced by one or two leaves that are tagged with the subformulae of the tagging formula. Sometimes a part of the tree can be removed. Let us consider some results about complexity, subformula property and counter-models. If we measure the size of the forest by the sum of the size of each tagging formula, then the size of any premise of a rule is strictly smaller than the size of the conclusion. Moreover, the size of the initial forest is exactly the size of initial sequent in the usual sense.

Theorem 6. *The depth of the proof-search tree is bounded by the size of the sequent (or forest) to prove.*

This result is important and already obtained in [3], providing an $\mathcal{O}(n \log n)$-space decision procedure for intuitionistic logic. Here we focus on an efficient implementation of such a procedure, based on structural sharing. Another important point is that a tagging formula is always replaced by some of its direct subformulae.

Theorem 7. *The tagging formulae occurring during the proof-search, as the forest gets decomposed, are subformulae of the initial sequent (or forest).*

The consequence for implementation is significant. In fact, an occurrence of a formula A is always decomposed by the same rule, which ever the context. This only depends on the *polarity*[3] of A as a subformula. It allows to do the allocation of space for subformulae before the proof-search starts, and not dynamically.
In section 3.3, we sketched an algorithm to effectively build a counter-model when the proof-search fails, like what was already done in [10]. The proof can be very easily adapted to the sharing-trees version of the system. In fact, one can effectively build Kripke-trees that are *strong counter models* (see [10]) in case of the failure of the proof-search algorithm on sharing trees.

Theorem 8. *The algorithm can produce proofs or counter-models and their depth is bounded by the size of the sequent to prove.*

5 Implementation Issues

The sharing tree version of the SLJ system is well suited for an efficient implementation in imperative languages. A proof-search algorithm requires a bounded memory space (depending on the size of the initial sequent) and it can be implemented without using dynamic memory allocation at all. Moreover, good choices are not obvious, especially in terms of time, if we aim to have a light use of resources. To define a good proof-search strategy is not an easy task and can depend on the kind of formulae. Its implementation could be difficult because it could involve too much administrative tasks like maintaining lists of formulae or even traversing lists at each point in the proof-search. A strategy has to take into account that some rules have the invertibility property. Moreover, in case the algorithm has to return counter-models, the strategies are restricted to those respecting the order in the completeness proof of section 3.3.
For a given strategy, the program having a sequent (set of sharing trees) as input has to choose a leaf of the tree. Going through the tree to search for a particular kind of leaf is not a good solution and it is necessary to gather leaves by types. But this information has to be maintained when the proof rules are applied. To summarise, the basic resources are the sequent represented as a tagged tree of formulae and the conclusion (or starred formula if we have a boxed sequent). This information is completed with some administrative information so as to make these resources accessible in minimal time. The key-points are the following: the

[3] This is the parity of the number of times a subformula occurs on the left of $\rightarrow$.

leaves should be organised by types to have an efficient choice. It is especially important to avoid the traversal of leaves at each point of the proof-search; the application of rules should not involve too much administrative tasks otherwise the benefits of the added information would be null; the reverse application of rules is also concerned by this point because the non-invertibility of some rules implies some backtracking.

6 Conclusion

In this paper, we have presented a new system SLJ derived from ideas of LJT [1] and LG [3] which is proved sound and complete for IPL. It is an intermediate step towards an efficient implementation of IPL based on the concept of sharing trees. This system has very nice properties like the subformula property, a linear bound on the depth of proof-search and the ability to generate Kripke counter-models. In addition to the theoretical complexity results, our approach provides an effective and fine control of space and time resources, adapted to the design in imperative languages. A first implementation has been developed in C and the practical feasibility of the method has been confirmed. A possible extension of this work would be, as in [10], implementing the intuitionistic proof of the system based on sharing trees and mechanically extracting a (proved) algorithm from this proof.

References

1. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992.
2. D. Galmiche and D. Larchey-Wendling. Formulae-as-resources management for an intuitionistic theorem prover. In *5th Workshop on Logic, Language, Information and Computation, WoLLIC'98*, Sao Paulo, Brazil, July 1998.
3. J. Hudelmaier. An $\mathcal{O}(n \log n)$-space decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.
4. M. Ornaghi P. Miglioli, U. Moscato. Avoiding duplications in tableaux systems for intuitionistic and Kuroda logic. *Logic Journal of the IGPL*, 5(1):145–167, 1997.
5. L. Pinto and R. Dyckhoff. Loop-free construction of counter-models for intuitionistic propositional logic. In Behara and al., editors, *Symposia Gaussiana*, pages 225–232, 1995.
6. N. Shankar. Proof search in the intuitionistic sequent calculus. In *11th Conference on Automated DEduction, LNAI 607*, pages 522–536, Saratoga Springs, June 1992.
7. T. Tammet. A resolution theorem prover for intuitionistic logic. In *13th Conference on Automated Deduction, LNAI 1104*, pages 2–16, NJ, USA, 1996.
8. C. Urban. Implementation of proof search in the imperative programming language pizza. In *Int. Conference TABLEAUX'98, LNAI 1397*, pages 313–319, Oisterwijk, The Netherlands, May 1998.
9. A. Voronkov. Proof-search in intuitionistic logic based on constraint satisfaction. In *Int. Workshop TABLEAUX'96, LNAI 1071*, pages 312–327, Italy, 1996.
10. K. Weich. Decisions procedures for intuitionistic logic by program extraction. In *Int. Conference TABLEAUX'98, LNAI 1397*, pages 292–306, Oisterwijk, The Netherlands, May 1998.

Verifying Probabilistic Programs Using a Hoare like Logic

J.I. den Hartog

Faculty of Exact Sciences, Free Universiteit,
de Boelelaan 1081, 1081 HV Amsterdam, the Netherlands
`jerry@cs.vu.nl`

Abstract. Hoare logic can be used to verify properties of deterministic programs by deriving correctness formulae, also called Hoare triples. The goal of this paper is to extend the Hoare logic to be able to deal with probabilistic programs. To this end a generic non-uniform language $\mathcal{L}_{\mathrm{pw}}$ with a probabilistic choice operator is introduced and a denotational semantics $\mathcal{D}$ is given for the language. A notion of probabilistic predicate is defined to express claims about the state of a probabilistic program. To reason about the probabilistic predicates a derivation system *pH*, similar to that of standard Hoare logic, is given. The derivation system is shown to be correct with respect to the semantics $\mathcal{D}$. Some basic examples illustrate the use of the system.

1 Introduction

Probability is introduced into the description of computer systems to model the inherent probabilistic behaviour of processes like, for example, a faulty communication medium. Probability is also explicitly introduced to obtain randomized algorithms to solve problems which can not be solved efficiently, or can not be solved at all, by deterministic algorithms. With increasing complexity of computer programs and systems, formal verification has become an important tool in the design. The presence of probabilistic elements in a program usually makes understanding and testing of the program more difficult. A way of formally verifying programs becomes even more important.

To formally verify a probabilistic program, the semantics of the program is given. The mathematical model of the program that is obtained in this way can be used to directly check properties of the program. In the probabilistic analyses of the model, results from probability theory are used to obtain e.g. average performance or bounds on error probabilities [18, 15]. Models that are often used are Markov chains and Markov decision processes [11, 4] probabilistic input-output automata [19, 20] and probabilistic transition systems [9, 8], sometimes augmented with probabilistic bisimulation [14, 10, 2].

For some programs the construction of the mathematical model can already be difficult. A systematic approach to simplify the program, or obtain properties without having to actually calculate the semantics are useful. Approaches in this area are probabilistic process algebra [2] and stochastic process algebra [6] where equivalences of programs can be checked syntactically by equational reasoning. Another approach is to introduce a logic to reason about the probabilistic programs. Here the later approach is followed by extending Hoare logic as known for deterministic programs to probabilistic programs. Other work on probabilistic logic can be found in e.g. [13, 4, 16, 17, 6]. In [13] an algebraic version of propositional dynamic logic is introduced. In [4] probabilities for temporal logic formulae are calculated. In [16] a weakest precondition calculus based on 'expectations' is defined and in [17] a notion of probabilistic predicate transformer, also used to find weakest preconditions, is given. Model checking is used in [1] to check formulae in a probabilistic temporal logic.

Deterministic Hoare logic is a system to derive correctness formulae, also called Hoare triples. A formula $\{p\}\ s\ \{q\}$ states that the predicate p is a sufficient precondition for the program s to guarantee that predicate q is true after termination. An extensive treatment of Hoare logic can be found in [5]. What the values of the variables in a program, i.e. the (deterministic) state of the program, will be, can not be fully determined if the program is probabilistic. Only the probability of being in a certain state can be given. This gives the notion of a probabilistic state. In a probabilistic state, a deterministic predicate will no longer be true or false, it is true with a certain probability. This can be dealt with by changing the interpretation of validity of a predicate to a function to $[0, 1]$ instead of to $\{\mathit{true}, \mathit{false}\}$ as in [13, 16]. The approach chosen here instead is to extend the syntax of predicates to allow making claims about the probability that a certain deterministic predicate holds. The extended form of predicates are called probabilistic predicates. A logic for probabilistic programs should reason with these probabilistic predicates.

In section 2 some mathematical definitions are given. The syntax of the non-uniform language $\mathcal{L}_{\mathrm{pw}}$ is given in section 3 together with its semantics. In section 4 probabilistic predicates are defined and a Hoare like logic is introduced to reason about probabilistic predicates. The logic is shown to be correct with respect to the denotational semantics. Some examples of the use of the logic are given in section 5 and some concluding remarks are given in section 6.

2 Mathematical Preliminaries

A complete partially ordered set (cpo) is a set with partial order $\leq$ that has a least element and for which each ascending chain has a least upper bound within the set. An order on Y is extended point wise to functions from X to Y ($f, g : X \to Y$ then $f \leq g$ if $f(x) \leq g(x)$ for all $x \in X$).

The support of a function $f : X \to [0, 1]$ is defined as the $x \in X$ for which $f(x) \neq 0$. The set of all functions from X to $[0, 1]$ with countable support is denoted by $X \to_{\mathcal{CS}} [0, 1]$. Given a function $f : X \to_{\mathcal{CS}} [0, 1]$ and a set $Y \subseteq X$ the sum $\sum f[Y] = \sum_{y \in Y} f(y)$ is well-defined (allowing the value ∞). The set of (pseudo) probabilistic measures $\mathcal{M}(X)$ on a set X is defined as the subset of functions in $X \to_{\mathcal{CS}} [0, 1]$ with sum at most 1.

$$\mathcal{M}(X) = \{\, f \in X \to_{\mathcal{CS}} [0, 1] \mid \textstyle\sum f[X] \leq 1 \,\} .$$

For a measure $f \in \mathcal{M}(X)$, $f(x)$ (for $x \in X$) is interpreted as the probability that x occurs. The set $\mathcal{M}(X)$ is a cpo, with minimal element $\underline{0}$, the function that assigns 0 to each element of X. For each ascending sequence in $\mathcal{M}(X)$ the limit exists within $\mathcal{M}(X)$ and corresponds to the least upper bound of the sequence.

For element $x \in X$ and $y \in Y$ and a function $f : X \to Y$, $f[x/y]$, called a variant of f, is defined by

$$f[x/y](x') = \begin{cases} y & \text{if } x = x' \\ f(x') & \text{otherwise.} \end{cases}$$

3 Syntax and Semantics of $\mathcal{L}_{\text{pw}}$

The language $\mathcal{L}_{\text{pw}}$ is a basic programming language with an extra operator used to denote probabilistic choice. A typical variable is denoted by v, the set of all variables by *Var*. The types of the variables are not made explicit. Instead a set of values *Val* is fixed as the range for all variables. (The examples deviate from this assumption and use integer and boolean variables.) Types can easily be added at the cost of complicating notation with less important details.

Definition 1. *The statements in $\mathcal{L}_{pw}$, ranged over by s, are given by:*

$$s ::= \mathit{skip} \mid v := e \mid s; s \mid s \oplus_r s \mid \mathit{if}\ c\ \mathit{then}\ s\ \mathit{else}\ s\ \mathit{fi} \mid \mathit{while}\ c\ \mathit{do}\ s\ \mathit{od}\,,$$

where $c \in BC$ is a boolean condition, $e \in Exp$ is an expression over values in Var and variables in Var and r is a ratio in the open interval $(0, 1)$.

The statements are interpreted as follows. The statement $s \oplus_r s'$ makes a probabilistic choice. With probability r the statement s will be executed, and s' will be executed with probability $1 - r$. The other constructs of $\mathcal{L}_{\text{pw}}$ are well known. The *skip* statement does nothing. Assignment $v := e$ assigns the value of the expression e to the variable v. Sequential composition $s; s'$ is executed by first executing s, then executing s'. The *if* c *then* s *else* s' *fi* statement executes s if the condition c holds, and otherwise s'. Finally, *while* c *do* s *od* repeatedly executes s until condition c no longer holds.

The internal details of the boolean conditions (*BC*) and expressions (*Exp*) are abstracted away from. Instead of defining an explicit syntax for the boolean conditions and the expressions, it is assumed that given the value of the variables, they can be evaluated. This is made more precise below.

For a deterministic program, the state of the computation is given by the value of the variables. The state space $\mathcal{S}$ for a deterministic program consists of $\mathcal{S} = \mathit{Var} \to \mathit{Val}$. For a probabilistic program, the values of the variables are no longer determined. For example, after executing $x := 0 \oplus_{\frac{1}{2}} x := 1$, the value of x could be zero but it could also be one. Instead of giving the value of a variable, a distribution over possible variables should be given. A first idea may be to take as a state space $\mathit{Var} \to \mathcal{M}(\mathit{Val})$. This does give, for each variable v, the chance that v takes a certain value but it does not describe the possible dependencies between the variables.

Consider the following example. In the left situation, a fair coin is thrown and a second coin is put beside it with the same side up. In the right situation, two fair coins are thrown. The two situations are indistinguishable if the dependency between the two coins is not known; the probability of heads or tails is $\frac{1}{2}$ for both coins in both situations. The difference between the situations is important e.g. if the next step is comparing the coins. In the first situation the coins are always equal, In the second situation they are equal with probability $\frac{1}{2}$ only.

		coin 1	
		heads	tails
coin 2	heads	$\frac{1}{2}$	0
	tails	0	$\frac{1}{2}$

		coin 1	
		heads	tails
coin 2	heads	$\frac{1}{4}$	$\frac{1}{4}$
	tails	$\frac{1}{4}$	$\frac{1}{4}$

The more general state space $\Pi = \mathcal{M}(\mathit{Var} \to \mathit{Val})$ is required. In $\theta \in \Pi$, instead of giving the distributions for the variables separately, the probability of being in a certain deterministic state is given. The chance that a variable v takes value w can be found by summing the probabilities of all states which assign w to v.

Definition 2.

(a) The set of deterministic states $\mathcal{S}$, ranged over by σ, is given by $\mathcal{S} = \mathit{Var} \to \mathit{Val}$.
(b) The evaluation functions $\mathcal{V} : \mathit{Exp} \to \mathcal{S} \to \mathit{Val}$ and $\mathcal{B} : \mathit{BC} \to \mathcal{S} \to \{\mathit{true}, \mathit{false}\}$ are the functions that compute the value of expressions and boolean conditions.
(c) The set of (pseudo) probabilistic states Π, ranged over by θ, is given by $\Pi = \mathcal{M}(\mathcal{S})$.
(d) On Π the following operations are defined:

$$\begin{aligned} \theta_1 \oplus_r \theta_2 &= r \cdot \theta_1 + (1-r) \cdot \theta_2, \\ c?\theta(\sigma) &= \begin{cases} \theta(\sigma) & \text{if } c \text{ true in } \sigma \text{ i.e. } \mathcal{B}(c)(\sigma) = \mathit{true}, \\ 0 & \text{otherwise}, \end{cases} \\ \theta[v/\mathcal{V}(e)](\sigma) &= \textstyle\sum \theta[\{\, \sigma' \mid \sigma'[v/\mathcal{V}(e)(\sigma')] = \sigma \,\}] \,. \end{aligned}$$

where $+$ is standard addition of functions and $r \cdot$ is scalar multiplication.

Note that $\theta \in \Pi$ is a function from $\mathcal{S}$ to $[0,1]$. The value $\theta(\sigma)$ returned by θ is the probability of being in the deterministic state σ.

The functions $\mathcal{V}$ and $\mathcal{B}$ are assumed given. The syntactic details of expressions and conditions as well as the precise definitions of these functions are abstracted away from. In a probabilistic state the values of the variables are, in general, not known and the value of expressions and conditions can not be found. Evaluation of expressions and conditions can only be done in a deterministic state. To find the probability of being in a state σ if in θ the expression e is assigned to variable v, the probabilities of all states σ' that yield σ after changing the value for v to that of e (evaluated in σ') have to be added.

The denotational semantics $\mathcal{D}$ for $\mathcal{L}_{\mathrm{pw}}$ gives, for each statement s, and state θ, the state $\mathcal{D}(s)(\theta)$ resulting from executing s starting in state θ.

Definition 3.

(a) The higher-order operator $\Psi_{\langle c,s\rangle} : (\Pi \to \Pi) \to (\Pi \to \Pi)$ *is given by*

$$\Psi_{\langle c,s\rangle}(\psi)(\theta) = \psi(\mathcal{D}(s)(c?\theta)) + \neg c?\theta.$$

(b) The denotational semantics $\mathcal{D} : \mathcal{L}_{pw} \to (\Pi \to \Pi)$ *is given by*

$$\begin{aligned}
\mathcal{D}(\textit{skip})(\theta) &= \theta \\
\mathcal{D}(v := e)(\theta) &= \theta[v/\mathcal{V}(e)] \\
\mathcal{D}(s; s')(\theta) &= \mathcal{D}(s')(\mathcal{D}(s)(\theta)) \\
\mathcal{D}(s \oplus_r s')(\theta) &= \mathcal{D}(s)(\theta) \oplus_r \mathcal{D}(s')(\theta) \\
\mathcal{D}(\textit{if } c \textit{ then } s \textit{ else } s' \textit{ fi})(\theta) &= \mathcal{D}(s)(c?\theta) + \mathcal{D}(s')(\neg c?\theta) \\
\mathcal{D}(\textit{while } c \textit{ do } s \textit{ od}) \quad &= \textit{the least fixed point of } \Psi_{\langle c,s\rangle} \, .
\end{aligned}$$

For a while statement *while c do s od*, one would like to use the familiar unfolding to *if c then s*; *while c do s od else skip fi*. This can not be done directly, as the second statement is more complex than the first. Instead we can use the fact that $\mathcal{D}$(*while c do s od*) is a fixed point of the higher-order operator $\Psi_{\langle c,s\rangle}$ to show that

$$\begin{aligned}
\mathcal{D}(\textit{while } c \textit{ do } s \textit{ od})(\theta) &= \Psi_{\langle c,s\rangle}(\mathcal{D}(\textit{while } c \textit{ do } s \textit{ od}))(\theta) \\
&= \mathcal{D}(\textit{while } c \textit{ do } s \textit{ od})(\mathcal{D}(s)(c?\theta)) + \neg c?\theta \\
&= \mathcal{D}(\textit{if } c \textit{ then } s; \textit{while } c \textit{ do } s \textit{ od else skip fi})(\theta) \, .
\end{aligned}$$

Note that the total probability of $\mathcal{D}$(*while c do s od*)(θ) may be less than that of θ. The 'missing' probability is the probability of non-termination.

The least fixed point of $\Psi_{\langle c,s\rangle}$ can be contructed explicitly.

Definition 4. *For a statement s define* $s^0 = s$ *and* $s^{n+1} = s; s^n$*. The functions* $\textit{if}^n_{\langle c,s\rangle}$ *and* $L_{\langle c,s\rangle}$ *from probabilistic states to probabilistic states are given by*

$$\begin{aligned}
\textit{if}^n_{\langle c,s\rangle}(\theta) &= \mathcal{D}((\textit{if } c \textit{ then } s \textit{ else skip fi})^n)(\theta) \\
L_{\langle c,s\rangle}(\theta) &= \lim_{n\to\infty} \neg c?\textit{if}^n_{\langle c,s\rangle}(\theta).
\end{aligned}$$

Lemma 1. *The least fixed point of* $\Psi_{\langle c,s\rangle}$ *is given by* $L_{\langle c,s\rangle}$.

The function $\text{if}^n_{\langle c,s\rangle}$ is merely a shorthand notation. The function $L_{\langle c,s\rangle}$ characterizes the least fixed point of $\Psi_{\langle c,s\rangle}$ and is thus equal to $\mathcal{D}$(*while c do s od*).

4 Probabilistic Predicates and Hoare Logic

The deterministic predicates used with deterministic Hoare logic are first order predicate formulae. Here *dp* is used to denote a deterministic predicate. The usual notions of fulfillment, $\sigma \models dp$ i.e. *dp* holds in σ, and substitution, $dp[v/e]$, for deterministic predicates are assumed to be known. An important property of substitution is that $\sigma \models dp[v/e]$ exactly when $\sigma[v/\mathcal{V}(e)(\sigma)] \models dp$. (Replacing the variable by an expression in the predicate is the opposite of assigning the value of the expression to the variable in the state.)

A deterministic Hoare triple, or correctness formula, $\{\, dp \,\}\ s\ \{\, dp' \,\}$, describes that *dp* is a pre condition and *dp'* is a post condition of program s. The Hoare triple is said to be correct if execution s in any state that satisfies *dp* will lead to a state satisfying *dp'*. To extend the Hoare triples to probabilistic programs, a notion of probabilistic predicate has to be introduced. One option is to use the same predicates as for deterministic programs but to change the interpretation of a predicate. A deterministic predicate can be seen as a function from states to $\{\, 0, 1 \,\}$, returning 1 if the state satisfies the predicate and 0 otherwise. The predicates can be made probabilistic by making them into functions to $[0, 1]$, returning the probability that the predicate is satisfied in a probabilistic state (See e.g. [13, 17]). This approach, however, does not allow making claims about the probability within the predicate itself, only the value of the predicate gives information about the probabilities. A property like "*dp* holds with probability *ce*" can not be expressed as a predicate. Also the normal logical operators like $\wedge$ have to be extended to work on $[0, 1]$.

In this paper probabilistic predicates can only have a truth value i.e. true or false. Probabilistic predicates are predicates in the usual sense, but with an extended syntax to express claims about probabilities. The construct $\mathbb{P}(dp) \prec ce$, for $\prec \in \{\, <, \leq, =, \geq, > \,\}$, is the basis for probabilistic predicates. Here *dp* is any deterministic predicate and *ce* is an expression, not using program variables, evaluating to a number in $[0, 1]$. The predicate $\mathbb{P}(dp) = ce$ holds in a state θ if the chance in θ of being in a deterministic state that satisfies *dp* is equal to *ce*. Similar for the other choices for $\prec$. Probabilistic predicates can be combined by the logical operators from predicate logic. For example, assuming that $Val = \{\, 1, 2, \dots \,\}$, $\forall i : \mathbb{P}(v = i) = \frac{1}{2}^i$ is a valid predicate stating that v has a geometric distribution. The expression $\frac{1}{2}^i$ uses the logical variable i, but does not depend on program variables like v. Furthermore for probabilitic predicates p and p', $p + p'$, $r \cdot p$ and $c?p$ are also probabilistic predicates. Their interpretation is given below.

Definition 5.

(a) *A probabilistic predicate is a basic probabilistic predicate of the form* $\mathbb{P}(dp) \prec ce$ *(*$\prec \in \{\, <, \leq, =, \geq, > \,\}$*) or a composition of probabilistic predicates with one of the logic operators* $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\exists$, $\forall$, *or one of the operators* $+$, $r\cdot$, $c?$. *Probabilistic predicates are ranged over by* p *and* q. *The following shorthand notations are also used*

$$\begin{aligned} p \oplus_r p' &= r \cdot p + (1 - r) \cdot p', \\ [dp] &= \mathbb{P}(dp) = 1, \\ \mathit{not}\ c &= \mathbb{P}(c) = 0\,. \end{aligned}$$

(b) The probabilistic predicates are interpreted as follows.
$\theta \models \mathbb{P}(dp) \prec ce$ *when* $\sum_{\sigma \models dp} \theta(\sigma) \prec ce$,
$\theta \models p_1 + p_2$ *when there exists* θ_1, θ_2 *with* $\theta = \theta_1 + \theta_2$, $\theta_1 \models p_1$ *and* $\theta_2 \models p_2$,
$\theta \models r \cdot p$ *when there exists* θ' *such that* $\theta = r \cdot \theta'$ *and* $\theta' \models p$,
$\theta \models c?p$ *when there exists* θ' *for which* $\theta = c?\theta'$ *and* $\theta' \models p$.
For the logical connectives the interpretation is as usual.

(c) Substitution on probabilistic predicates $[v/e]$ *is passed down through all constructs until a deterministic predicate is reached.*

$$\begin{aligned}(\mathbb{P}(dp) \prec ce)[v/e] &= \mathbb{P}(dp[v/e]) \prec ce, \\ (p\ op\ p')[v/e] &= p[v/e]\ op\ p'[v/e] \quad op \in \{\wedge, \vee, \Rightarrow, +, \oplus_r\}, \\ (op\ p)[v/e] &= op\ (p[v/e]) \quad op \in \{\neg, \exists, \forall, r\cdot\} \\ (c?p)[v/e] &= c[v/e]?(p[v/e])\,.\end{aligned}$$

Note that the extension of deterministic predicates to [0,1]-valued functions is more or less incorporated within the probabilistic predicates as used in this paper. To check the probability of a certain deterministic predicate dp in state θ, look for which r the predicate $\mathbb{P}(dp) = r$ is true in θ instead of checking the value of dp in θ is r.

When reasoning about probabilistic predicates, caution is advised. Some equivalences which may seem true at first sight do not hold. The most important of these is that in general $p \oplus_r p \not\Leftrightarrow p$. Take for example $\mathbb{P}(x = 1) = 1 \vee \mathbb{P}(x = 2) = 1$ for p and a state satisfying $\mathbb{P}(x = 1) = \frac{1}{2} + \mathbb{P}(x = 2) = \frac{1}{2}$ will satisfy $p \oplus_{\frac{1}{2}} p$ but not p. Other examples are $p = \exists i : q[i]$ and $p = \forall i : (q[i] \vee q'[i])$. The equivalence does hold for the basic predicates $\mathbb{P}(dp) \prec r$ and if the equivalence holds when $p = q$ and when $p = q'$ then it also holds for $p = q \wedge q'$.

Using probabilistic predicates the Hoare-triples as introduced for deterministic programs can be extended to probabilistic programs. Hoare triple $\{p\}\ s\ \{q\}$ indicates that p is a pre condition and q is a post condition for the probabilistic program s. The Hoare triple is said to hold, denoted by $\models \{p\}\ s\ \{q\}$, if the pre condition p guarantees that post condition q holds after execution of s.

$$\models \{p\}\ s\ \{q\} \text{ if } \forall \theta \in \Pi : \theta \models p \Rightarrow \mathcal{D}(s)(\theta) \models q.$$

For example $\models \{p\}\ \mathit{skip}\ \{p\}$ and $\models \{\mathbb{P}(x = 0) = 1\}\ x := x + 1\ \{\mathbb{P}(x = 1) = 1\}$.

To prove the validity of Hoare triples, a derivation system called *pH* is introduced. The derivation system consists of the axioms and rules as given below.

$$\{p\}\ \mathit{skip}\ \{p\} \quad \text{(Skip)} \qquad \frac{\{p\}\ s\ \{q\} \quad \{p\}\ s'\ \{q'\}}{\{p\}\ s \oplus_c s'\ \{q \oplus_c q'\}} \quad \text{(Prob)}$$

$$\{p[v/e]\}\ v := e\ \{p\} \quad \text{(Assign)} \qquad \frac{\{c?p\}\ s\ \{q\} \quad \{\neg c?p\}\ s'\ \{q'\}}{\{p\}\ \mathit{if}\ c\ \mathit{then}\ s\ \mathit{else}\ s'\ \mathit{fi}\ \{q + q'\}} \quad \text{(If)}$$

$$\frac{\{p\}\ s\ \{p'\} \quad \{p'\}\ s'\ \{q\}}{\{p\}\ s; s'\ \{q\}} \quad \text{(Seq)} \qquad \frac{p \text{ invariant for } \langle c, s\rangle}{\{p\}\ \mathit{while}\ c\ \mathit{do}\ s\ \mathit{od}\ \{p \wedge \text{not } c\}} \quad \text{(While)}$$

$$\frac{\{\,p[j]\,\}\ s\ \{\,q\,\} \quad j \notin p, q}{\{\,\exists i : p[i]\,\}\ s\ \{\,q\,\}}\ \text{(Exists)} \qquad \frac{p' \Rightarrow p \quad \{\,p\,\}\ s\ \{\,q\,\} \quad q \Rightarrow q'}{\{\,p'\,\}\ s\ \{\,q'\,\}}\ \text{(Imp)}$$

$$\frac{\{\,p\,\}\ s\ \{\,q[j]\,\} \quad j \notin p, q}{\{\,p\,\}\ s\ \{\,\forall i : q[i]\,\}}\ \text{(Forall)} \qquad \frac{\{\,p\,\}\ s\ \{\,q\,\} \quad \{\,p'\,\}\ s\ \{\,q\,\}}{\{\,p \vee p'\,\}\ s\ \{\,q\,\}}\ \text{(Or)}$$

The rules (Skip), (Assign), (Seq) and (Cons) are as within standard Hoare logic but now dealing with probabilistic predicates. The rules (If) and (While) have changed and the rules (Prob), (Or), (Exists) and (Forall) are new.

For p to hold after the execution of *skip*, it should hold before the execution since *skip* does nothing. The predicate p holds after an assignment $v := e$ exactly when p with e substituted for v holds before the assignment, as the effect of the assignment is exactly replacing v with the value of e. The rule (Seq) states that p is a sufficient pre condition for q to hold after execution of $s; s'$ if there exists an intermediate predicate p' which holds after the execution of s and which implies that q holds after the execution of s'. The rule (Cons) states that the pre condition may be strengthened and the post condition may be weakened.

The rule (Prob) states that the result of executing $s \oplus_r s'$ is obtained by combining the results obtained by executing s and s' with the appropriate probabilities. The necessity for the (Or), (Exists) and (Forall) rules becomes clear when one recalls that $p \oplus_r p \not\Rightarrow p$. Proving correctness of $\{\,p \vee q\,\}$ *skip*$\oplus_r$*skip* $\{\,p \vee q\,\}$ is, in general, not possible without the (Or)-rule. Similar examples show the need for the (Exists) and (Forall) rule. Note the similarity with the natural deduction rules for $\vee$ and $\exists$ elimination and $\forall$ introduction.

The rule (If) has changed with respect to the (If) rule of standard Hoare logic. In a probabilistic state the value of the boolean condition c is not determined. Therefore the probabilistic state is split into two parts, a part in which c is true and a part in which c is false. After splitting the state, the effect of the corresponding statement, either s or s', can be found after which the parts are recombined using the $+$ operator.

To use the (While) rule, an invariant p should be found. For p to be an invariant, it should satisfy $\{\,p\,\}$ *if c then s else skip fi* $\{\,p\,\}$. This condition is sufficient to obtain partial correctness. If the program s terminates and $\{\,p\,\}\ s\ \{\,q\,\}$ can be derived from *pH*, then $\models \{\,p\,\}\ s\ \{\,q\,\}$. A probabilistic program is said to terminate, if the program is sure to terminate when all probabilistic choices are interpreted as non-deterministic choices, i.e if the program terminates for all possible outcomes of the probabilistic choices. Partial correctness, however, is not sufficient for probabilistic programs. Many probabilistic programs do not satisfy the termination condition, they may for instance only terminate with a certain probability. (Note that, even if that probability is one, the termination condition need not be satisfied.) To derive valid Hoare triples for programs that need not terminate, a form of total correctness is required. This requires somehow adding termination conditions to the rules. To obtain total correctness we strengthen the notion of invariant by imposing the extra condition of $\langle c, s\rangle$-closedness.

Definition 6.

(a) For a predicate p the n-step termination ratio, denoted by $r^n_{\langle c,s\rangle}$, is the probability that, starting from a state satisfying p, the while loop "while c do s od" terminates within n steps.

$$\begin{aligned} r\theta^n_{\langle c,s\rangle} &= \textstyle\sum \neg c?\mathit{if}^n_{\langle c,s\rangle}(\theta)[\mathcal{S}] \\ r^n_{\langle c,s\rangle} &= \inf\{\, r\theta^n_{\langle c,s\rangle} \mid \theta \models p\,\}\,. \end{aligned}$$

(b) A sequence of states $(\theta_n)_{n\in\mathbb{N}}$ is called a $\langle c,s\rangle$-sequence if $(\neg c?\theta_n)_{n\in N}$ is an ascending sequence with $\sum \neg c?\theta_n[\mathcal{S}] \geq r^n_{\langle c,s\rangle}$.

(c) A predicate p is called $\langle c,s\rangle$-closed if each $\langle c,s\rangle$-sequence within p has a limit (least upper bound) within p.

p invariant for $\langle c,s\rangle$ when
$\{\,p\,\}$ if c then s else skip fi $\{\,p\,\}$ and p is $\langle c,s\rangle$-closed.

Note that for a loop *while c do s od* that terminates every p automatically satisfies $\langle c,s\rangle$-closedness. Therefore, for a terminating program, there is no need to check any $\langle c,s\rangle$-closedness conditions.

A Hoare triple $\{\,p\,\}\ s\ \{\,q\,\}$ is said to be derivable from the system *pH*, denoted by $\vdash\ \{\,p\,\}\ s\ \{\,q\,\}$, if there exists a proof tree for $\{\,p\,\}\ s\ \{\,q\,\}$ in *pH*. The derivation system is correct, i.e. only valid Hoare triples can be derived from *pH*.

Lemma 2. *The derivation system pH is correct, i.e. for all predicates p and q and statements s, $\vdash\ \{\,p\,\}\ s\ \{\,q\,\}$ implies $\models\ \{\,p\,\}\ s\ \{\,q\,\}$.*

Proof. It is sufficient to show that if $\theta \models p$ and $\vdash\ \{\,p\,\}\ s\ \{\,q\,\}$ then $\mathcal{D}(s)(\theta) \in q$. This is shown by induction on the depth of the derivation tree for $\{\,p\,\}\ s\ \{\,q\,\}$, by looking at the last rule used. A few cases are given below.

- *If the rule (Exists) was used and $\theta \models \exists i : p[i]$ then there is an i_0 for which $\theta \models p[i_0]$. By induction $\models\ \{\,p[j]\,\}\ s\ \{\,q\,\}$ which gives, by substituting the value i_0 for the free variable j $\{\,p[i_0]\,\}\ s\ \{\,q\,\}$. But then $\mathcal{D}(s)(\theta) \models q$.*
- *Known from the non-probabilistic case is that $\sigma[v/\mathcal{V}(e)(\sigma)] \models dp$ exactly when $\sigma \models dp[v/e]$. By induction on the structure of the probabilistic predicate p this extends to $\theta[v/\mathcal{V}(e)] \models p$ exactly when $\theta \models p[v/e]$. Correctness of the (Assign) rule follows directly.*
- *If rule (Prob) is used to derive $\vdash\ \{\,p\,\}\ s \oplus_r s'\ \{\,q \oplus q'\,\}$ from $\vdash\ \{\,p\,\}\ s\ \{\,q\,\}$ and $\vdash\ \{\,p\,\}\ s'\ \{\,q'\,\}$ then by induction $\models\ \{\,p\,\}\ s\ \{\,q\,\}$ and $\models\ \{\,p\,\}\ s'\ \{\,q'\,\}$. This means that if $\theta \models p$ then $\mathcal{D}(s)(\theta) \models q$ and $\mathcal{D}(s')(\theta) \models q'$. But then $\mathcal{D}(s \oplus s')(\theta) = \mathcal{D}(s)(\theta) \oplus_r \mathcal{D}(s')(\theta) \models q \oplus_r q'$. The case for rule (If) is similar.*
- *Assume rule (While) is used with statement s, condition c and invariant p. Clearly $\mathcal{D}$(while c do s od)$(\theta) \models$ not c and $\models\ \{\,p\,\}$ if c then s else skip fi $\{\,p\,\}$ can be used repeatedly to gives that if $\theta \models p$ then $(\mathit{if}^n_{\langle c,s\rangle}(\theta))_{n\in\mathbb{N}}$ is a $\langle c,s\rangle$ sequence. By $\langle c,s\rangle$-closedness $\mathcal{D}$(while c do s od)$(\theta) = L_{\langle c,s\rangle} \models p$.*

5 Examples

The picture below gives an example of a proof tree in the system *pH*. For larger programs, instead of giving the proof tree, a proof outline is used. In a proof outline the rules (Imp) and (Seq) are implicitly used by writing predicates between the statements and some basic steps are skipped. A predicates between the statements give conditions that the intermediate states in the computation must satisfy.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\,[x+1=2]\,\}\ x:=x+1\ \{\,[x=2]\,\}}\text{(Assign)}}{\{\,[x=1]\,\}\ x:=x+1\ \{\,[x=2]\,\}}\text{(Imp)} \qquad \dfrac{\dfrac{}{\{\,[x+2=3]\,\}\ x:=x+2\ \{\,[x=3]\,\}}\text{(Assign)}}{\{\,[x=1]\,\}\ x:=x+2\ \{\,[x=3]\,\}}\text{(Imp)}}{\{\,[x=1]\,\}\ x:=x+1 \oplus_{\frac{1}{2}} x:=x+2\ \{\,[x=2] \oplus_{\frac{1}{2}} [x:=3]\,\}}\text{(Prob)}}{\{\,[x=1]\,\}\ x:=x+1 \oplus_{\frac{1}{2}} x:=x+2\ \{\,\mathbb{P}(x=2)=\frac{1}{2} \wedge \mathbb{P}(x=3)=\frac{1}{2}\,\}}\text{(Imp)}$$

The following program adds an array of numbers, but some elements may inadvertently get skipped. A lower bound on the probability that the answer will still be correct is derived. An n-ary version of $\vee$ is used as a shorthand.

```
int $ss[1\ldots N],\ r,\ k$;
$\{\,[true]\,\} \Rightarrow \{\,\mathbb{P}(0=0, 1=1)=1\,\}$
$t=0;\qquad k=1;$
$\{\,\mathbb{P}(t=0, k=1)=1\,\} \Rightarrow$
$\{\,\mathbb{P}(k=N+1, t=\sum_{i=1}^{N} ss[i]) \geq r^N\ \vee\ \vee_{n=0}^{N}\mathbb{P}(k=n, t=\sum_{i=1}^{k-1} ss[i]) \geq r^{n-1}\,\}$
while $(k \leq N)$ do
        $\{\,\vee_{n=0}^{N}\mathbb{P}(k=n, t=\sum_{i=1}^{k-1} ss[i]) \geq r^{n-1}\,\} \Rightarrow$
        $\{\,\vee_{n=0}^{N}\mathbb{P}(k=n, t+ss[k]=\sum_{i=1}^{k} ss[i]) \geq r^{n-1}\,\}$
        $t := t+ss[k] \oplus_r skip;$
        $\{\,\vee_{n=0}^{N}\mathbb{P}(k=n, t=\sum_{i=1}^{k} ss[i]) \geq r^{n-1} \oplus_r true\,\} \Rightarrow$
        $\{\,\vee_{m=1}^{N+1}\mathbb{P}(k+1=m, t=\sum_{i=1}^{k} ss[i]) \geq r^{m-1}\,\}$
        $k := k+1$
        $\{\,\vee_{m=1}^{N+1}\mathbb{P}(k=m, t=\sum_{i=1}^{k-1} ss[i]) \geq r^{m-1}\,\}$
od
$\{\,\vee_{n=0}^{N+1}\mathbb{P}(k=n, t=\sum_{i=1}^{k-1} ss[i]) \geq r^{n-1}\,\} \wedge$ not $(k \leq N)\ \Rightarrow$
$\{\,\mathbb{P}(t=\sum_{i=1}^{N} ss[i]) \geq r^N\,\}$
```

In the following example, a coin is tossed until heads is thrown. The number of required throws is shown to be geometrically distributed. For ease of notation the following shorthand notations are used.

$$p = q_\infty \vee \exists i : q[i]$$
$$q_\infty = \forall j > 0 : \mathbb{P}(x=j, done=true) = \tfrac{1}{2}^{j}$$
$$q[i] = \mathbb{P}(x=i, done=false) = \tfrac{1}{2}^{i} \wedge \forall j \in \{1,\ldots,i\} : \mathbb{P}(x=j, done=true) = \tfrac{1}{2}^{j}.$$

Then, assuming p is an invariant:

```
{ [true] }
bool done = false;    int n = 0;
{ $\mathbb{P}(n = 1, done\ = false) = 1$ }⇒ { $p$ }
while not done do $x := x + 1; done = true \oplus_{\frac{1}{2}}$ skip od
{ $p \wedge done$ }⇒ { $\forall n > 0 : \mathbb{P}(x = n) = \frac{1}{2}^n$ }
```

To show that p is an invariant proof the rule (Or) is used to split the proof into two parts, the first of which is trivial. For the second part the rule (Exists) is used to give:

```
{ $q[k]$ }
while not done do
    { $(not\ done)?q[k]$ } ⇒ { $\mathbb{P}(x = i, done\ = false) = \frac{1}{2}^i$ }
    $x := x + 1$;
    { $\mathbb{P}(x = i + 1, done\ = false) = \frac{1}{2}^i$ }
    $done\ = true \oplus_{\frac{1}{2}}$ skip
    { $\mathbb{P}(x = i + 1, done\ = false) = \frac{1}{2}^{i+1} + \mathbb{P}(x = i + 1, done\ = true) = \frac{1}{2}^{i+1}$ }
od
{ $\mathbb{P}(x = i + 1, done\ = false) = \frac{1}{2}^{i+1} + \mathbb{P}(x = i + 1, done\ = true) = \frac{1}{2}^{i+1} +$
  $\forall j \in \{1, \ldots, i\}$: $\mathbb{P}(x = j, done\ = true) = \frac{1}{2}^j$ } ⇒ { $q[j + 1]$ } ⇒ { $p$ }.
```

The requirement that p is $\langle not\ done, x := x + 1; done = true \oplus_{\frac{1}{2}} skip\rangle$-closed is easy to check but requires the presence of the q_∞ term.

6 Conclusions and Further Work

The main result of this paper is the introduction of a Hoare like logic, called *pH*, for reasoning about probabilistic programs. The programs are written in a language $\mathcal{L}_{\mathrm{pw}}$ and their meaning is given by the denotational semantics $\mathcal{D}$.

The probabilistic predicates used in the logic retain their usual truth value interpretation, i.e. they can be interpreted as true or false. Deterministic predicates can be extended to arithmetical functions yielding the probability that the predicate holds as done in e.g. [13] and [17]. This extension is incorporated by using the notation $\mathbb{P}(dp)$ to refer to exactly that, the chance that deterministic predicate *dp* holds. The chance of *dp* holding can then be exactly expressed or lower and/or upper bounds can be given within a probabilistic predicate. The main advantage of keeping the interpretation as truth values is that the logical operators do not have to be extended.

The logic *pH* is show correct with respect to the semantics $\mathcal{D}$. For an (earlier) infinite version of the logic a completeness result exists. For the current logic the question of completeness is still open. Especially the expressiveness of the probabilistic predicates has to studied further.

To be able to describe distributed randomized algorithms, it would also be interesting to extend the language and the logic with parallelism. However, verification of concurrent systems in general and extending Hoare logic to concurrent systems in specific (see e.g. [3, 7]) is already difficult in the non-probabilistic case.

To make the logic practically useful, the process of checking the derivation of a Hoare-triple should be automated. Some work has been done to embed the logic in the proof verification system PVS. (See e.g. [12] on non-probabilistic Hoare logic in PVS.) The system PVS can then be used both to check the applications of the rules and to check the derivation of the implications between predicates required for the (Imp) rule. By modeling probabilistic states, PVS could perhaps also be used to verify the correctness of the logic, however this would require a lot of work on modeling infinite sums.

References

[1] L. de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, 1997.

[2] S. Andova. Probabilistic process algebra. In *Proceedings of ARTS'99*, Bamberg, Germany, 1999.

[3] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Texts and Monographs in Computer Science. Springer-Verlag, 1991.

[4] C. Baier, M. Kwiatkowska, and G. Norman. Computing probabilistic lower and upper bounds for ltl formulae over sequential and concurrent markov chains. Technical Report CSR-98-4, University of Birmingham, June 1998.

[5] J.W. de Bakker. *Mathematical Theory of Program Correctness.* Series in Computer Science. Prentice-Hall Internationall, 1980.

[6] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. A stochastic process algebra for discrete event simulation. Technical report, University of Twente, 1998.

[7] N. Francez. *Program Verification.* International Computer Science Series. Addison-Wesley, 1992.

[8] J.I. den Hartog. Comparative semantics for a process language with probabilistic choice and non-determinism. Technical Report IR-445, Vrije Universiteit, Amsterdam, February 1998.

[9] J.I. den Hartog and E.P. de Vink. Mixing up nondeterminism and probability: A preliminary report. *ENTCS*, 1999. to appear.

[10] J.I. den Hartog and E.P. de Vink. Taking chances on merge and fail: Extending strong and probabilistic bisimulation. Technical Report IR-454, Vrije Universiteit, Amsterdam, March 1999.

[11] H. Hermanns. *Interactive Markov Chains.* PhD thesis, University of Erlangen-Nurnberg, July 1998.

[12] J. Hooman. Program design in PVS. In *Proceedings Workshop on Tool Support for System Development and Verification*, June 1996.

[13] D. Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30:162–178, 1985.

[14] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94:1–28, 1991.

[15] N. Lynch. *Distributed Algorithms.* The Morgan Kaufmann series in data management systems. Kaufmann, 1996.

[16] C. Morgan and A. McIver. pGCL: formal reasoning for random algorithms. *South African Computer Journal*, 1999. to appear.

[17] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
[18] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.
[19] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, June 1995.
[20] S.-H. Wu, S.A. Smolka, and E.W. Stark. Composition and behavior of probabilistic I/O automata. *Theoretical Computer Science*, 176:1–38, 1999.

An Expressive Extension of TLC
(Extended Abstract)*

Jesper Gulmann Henriksen

BRICS**, Department of Computer Science,
University of Aarhus, Denmark
gulmann@brics.dk

Abstract. A temporal logic of causality (TLC) was introduced by Alur, Penczek and Peled in [1]. It is basically a linear time temporal logic interpreted over Mazurkiewicz traces which allows quantification over causal chains. Through this device one can directly formulate causality properties of distributed systems. In this paper we consider an extension of TLC by strengthening the chain quantification operators. We show that our logic TLC* adds to the expressive power of TLC. We do so by defining an Ehrenfeucht-Fraïssé game to capture the expressive power of TLC. We then exhibit a property and by means of this game prove that the chosen property is not definable in TLC. We then show that the same property is definable in TLC*. We prove in fact the stronger result that TLC* is expressively stronger than TLC exactly when the dependency relation associated with the underlying trace alphabet is not transitive.

1 Introduction

One traditional approach to automatic program verification is model checking LTL [11] specifications. In this context, the model checking problem is to decide whether or not all computation sequences of the system at hand satisfy the required properties formulated as an assertion of LTL. Several software packages exploiting the rich theory of LTL are now available to carry out the automated verification task for quite large finite-state systems.

Usually computations of a distributed system will constitute interleavings of the occurrences of causally independent actions. Often, the computation sequences can be naturally grouped together into equivalence classes of sequences corresponding to different interleavings of the same partially ordered computation stretch. For a large class of interesting properties expressed by linear time temporal logics, it turns out that either all members of an equivalence class satisfy a certain property or none do. For such properties the verification task can be substantially improved by the partial-order methods for verification [6,10].

Such equivalence classes can be canonically represented by restricted labelled partial orders known as Mazurkiewicz traces [4,8]. These objects allow

* Part of this work was done at Lehrstuhl für Informatik VII, RWTH Aachen, Germany
** **B**asic **R**esearch **i**n **C**omputer **S**cience,
Centre of the Danish National Research Foundation.

direct formulations of properties expressing concurrency and causality. A number of linear time temporal logics to be interpreted directly over Mazurkiewicz traces (e.g. [1,3,9,12,13,14,15]) has been proposed in the literature starting with TrPTL [13].

Among these, we consider here a temporal logic of causality (TLC) introduced in [1] to express serializability (of partially ordered computations) in a direct fashion. The operators of TLC are essentially the branching-time operators of CTL [2] interpreted over causal chains of traces. However, the expressive power of this logic has remained an interesting open problem. Indeed, not much is known about the relative expressive powers of the various temporal logics over traces.

What is known is that a linear time temporal LTrL, patterned after LTL, was introduced [15] and proven expressively equivalent to the first-order theory of traces. LTrL has a simple and natural formulation with very restricted past operators, but was shown non-elementary in [16]. Recently, it was shown that the restricted past operators of LTrL can be replaced by certain new future operators while maintaining expressive completeness. In other work, Niebert introduced a fixed point based linear time temporal logic [9]. This logic has an elementary-time decision procesure and is equal in expressive power to the monadic second-order theory of traces.

However, the expressive powers of most other logics put forth (e.g. [1,12,13]) still have an unresolved relationship to each other and, in particular, to first-order logic. Most notably, it is still a challenging open problem whether or not TrPTL or TLC is expressively weaker than first-order logic. With virtually no other seperation result known, this paper is a contribution towards understanding the relative expressive power of such logics.

A weakness of TLC is that it doesn't facilitate direct reasoning about causal relationships between the individual events on the causal chains. In this paper we remedy this deficiency and extend TLC by strengthening quantification over causal chains. This extended logic, which we call TLC*, will enjoy a similarity to CTL* [2] that TLC has to CTL. The main result of this paper is that our extension TLC* is expressively stronger than TLC for general trace alphabets whereas they express the same class of properties over trace alphabets with a transitive dependency relation. We prove this result with the aid of an Ehrenfeucht-Fraïssé game for traces that we develop. To our knowledge this is the first instance of the use of such games to obtain seperation results for temporal logics defined over partial orders. We believe that this approach is fruitful and that similar techniques may lead to other seperation results within this area.

In the next section we briefly recall Mazurkiewicz traces and a few related notions. In Section 3 we introduce TLC and TLC*, the main objects of study in this paper. We give a very simple and natural example of a property easily captured in TLC* but not in TLC. In Section 4 we define an Ehrenfeucht-Fraïssé game and prove its correspondence to TLC. We use this correspondence in Section 5 to exhibit a property which we prove is undefinable in TLC. In

Section 6 we show that the said property can be defined within TLC*. Finally, we put all the pieces together to arrive at the main result.

2 Preliminaries

A *(Mazurkiewicz) trace alphabet* is a pair (Σ, I), where Σ, the alphabet, is a finite set and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric *independence relation.* Usually, Σ consists of the actions performed by a distributed system while I captures a static notion of causal independence between actions. For the rest of the section we fix a trace alphabet (Σ, I). We define $D = (\Sigma \times \Sigma) - I$ to be the *dependency relation* which is then reflexive and symmetric.

Let $T = (E, \leq, \lambda)$ be a Σ-labelled poset. In other words, $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labelling function. For $e \in E$ we define ${\downarrow}e = \{x \in E \mid x \leq e\}$. We also let $\lessdot$ be the *covering relation* given by $x \lessdot y$ iff $x < y$ and for all $z \in E$, $x \leq z \leq y$ implies $x = z$ or $z = y$. Moreover, we let the *concurrency relation* be defined as $x \; co \; y$ iff $x \not\leq y$ and $y \not\leq x$. A *Mazurkiewicz trace* (over (Σ, I)) is then a Σ-labelled poset $T = (E, \leq, \lambda)$ satisfying:

(T1) $\forall e \in E.$ ${\downarrow}e$ is a finite set
(T2) $\forall e, e' \in E.$ $e \lessdot e'$ implies $\lambda(e) \; D \; \lambda(e')$.
(T3) $\forall e, e' \in E.$ $\lambda(e) \; D \; \lambda(e')$ implies $e \leq e'$ or $e' \leq e$.

We shall let $TR(\Sigma, I)$ denote the class of traces over (Σ, I). As usual, a trace language L is a subset of traces, i.e. $L \subseteq TR(\Sigma, I)$. Throughout the paper we will not distinguish between isomorphic elements in $TR(\Sigma, I)$. We will refer to members of E as *events.* It will be convenient to assume the existence of a unique least event $\bot \in E$ corresponding to a system initialization event carrying no label, i.e. $\lambda(\bot)$ is undefined and $\bot < e$ for every $e \in E - \{\bot\}$. We will sometimes abuse notation and let a string in Σ^* denote its corresponding trace in (Σ, I) whenever no confusion arises. This is enforced by using conventional parentheses for string languages and square brackets for trace languages.

In setting the scene for defining the semantics of formulas of TLC* we first introduce some notation for sequences. The length of a finite sequence ρ will be denoted by $|\rho|$. In case ρ is infinite we set $|\rho| = \omega$. Let $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ and $0 \leq k < |\rho|$. We set $\rho_k = (e_k, e_{k+1}, \ldots, e_n, \ldots)$.

Let $T = (E, \leq, \lambda)$ be a trace over (Σ, I). A *future causal chain rooted at* $e \in E$ is a (finite or infinite) sequence $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ with $e = e_0$, $e_i \in E$ such that $e_{i-1} \lessdot e_i$ for every $i \geq 1$. The labelling function $\lambda : E \to \Sigma$ is extended to causal chains in the obvious way by: $\lambda(\rho) = (\lambda(e_0)\lambda(e_1) \cdots \lambda(e_n) \cdots)$. We say that a future causal chain ρ is *maximal* in case ρ is either infinite or it is finite and there exists no $e' \in E$ such that $e_{|\rho|} \lessdot e'$. A *past causal chain rooted at* $e \in E$ is defined in the obvious manner.

3 Syntax and Semantics

In this section we will define the syntax and semantics of the temporal logics over traces to be considered in this paper. We start by introducing TLC* and continue by giving an explicit definition of the sublogic TLC. We will not define first-order logic over traces (FO), but we refer the reader to e.g. [14,15].

TLC* consists of three different syntactic entities; event formulas (Φ_{ev}), future chain formulas (Φ^+_{ch}) and past chain formulas (Φ^-_{ch}) defined by mutual induction as described below:

$$\begin{aligned}
\Phi_{ev} &::= p_a \mid {\sim}\alpha \mid \alpha_1 \vee \alpha_2 \mid \mathrm{co}(\alpha) \mid E(\phi) \mid E^-(\psi), \text{ with } a \in \Sigma.\\
\Phi^+_{ch} &::= \alpha \mid {\sim}\phi \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 U \phi_2.\\
\Phi^-_{ch} &::= \alpha \mid {\sim}\psi \mid \psi_1 \vee \psi_2 \mid X^-\psi \mid \psi_1 U^- \psi_2 \ ,
\end{aligned}$$

where α, ϕ and ψ (with or without subscripts) are formulas of Φ_{ev}, Φ^+_{ch} and Φ^-_{ch}, respectively. The formulas of TLC$^*(\Sigma, I)$ are the set of *event* formulas Φ_{ev} as defined above[1].

The semantics of formulas of TLC* is divided into two parts; event formulas and chain formulas. Let $T \in TR(\Sigma, I)$ and $e \in E$. The notion of an event formula α being satified at an event e of T is defined inductively in the following manner.

- $T, e \models p_a$ iff $\lambda(e) = a$.
- $T, e \models {\sim}\alpha$ iff $T, e \not\models \alpha$.
- $T, e \models \alpha_1 \vee \alpha_2$ iff $T, e \models \alpha_1$ or $T, e \models \alpha_2$.
- $T, e \models \mathrm{co}(\alpha)$ iff there exists an $e' \in E$ with $e \ co \ e'$ and $T, e' \models \alpha$.
- $T, e \models E(\phi)$ iff there exists a future causal chain ρ rooted at e with $T, \rho \models \phi$.
- $T, e \models E^-(\psi)$ iff there exists a past causal chain ρ rooted at e with $T, \rho \models \psi$.

As usual, $\mathrm{tt} = p_a \vee {\sim}p_a$ and $\mathrm{ff} = {\sim}\mathrm{tt}$. Suppose $\rho = (e_0, e_1, \ldots, e_n, \ldots)$ is a future causal chain. The notion of $T, \rho \models \phi$ for a future chain formula ϕ is defined inductively below.

- $T, \rho \models \alpha$ iff $T, e_0 \models \alpha$.
- $T, \rho \models {\sim}\phi$ iff $T, \rho \not\models \phi$.
- $T, \rho \models \phi_1 \vee \phi_2$ iff $T, \rho \models \phi_1$ or $T, \rho \models \phi_2$.
- $T, \rho \models X\phi$ iff $T, \rho_1 \models \phi$.
- $T, \rho \models \phi_1 U \phi_2$ iff there exists a $0 \leq k < |\rho|$ such that $T, \rho_k \models \phi_2$. Moreover, $T, \rho_m \models \phi_1$ for each $0 \leq m < k$.

The notion of $T, \rho \models \psi$ for a past causal chain ρ and past chain formula ψ is defined in the straightforward manner. The well-known future chain operators are derived as $F\phi = \mathrm{tt}U\phi$ and $G\phi = {\sim}F{\sim}\phi$.

Suppose $T \in TR(\Sigma, I)$ and $\alpha \in$ TLC$^*(\Sigma, I)$. Then T *satisfies* α iff $T, \bot \models \alpha$, denoted $T \models \alpha$. The *language defined by* α is: $\mathcal{L}(\alpha) = \{T \in TR(\Sigma, I) \mid T \models \alpha\}$. We say that $L \subseteq TR(\Sigma, I)$ is *definable* in TLC* if there exists some $\alpha \in$ TLC$^*(\Sigma, I)$ such that $\mathcal{L}(\alpha) = L$. By slight abuse of notation, the class of trace languages over (Σ, I) definable in TLC* will also be denoted by TLC$^*(\Sigma, I)$.

[1] Another logic was in [1] termed "TLC*", but as that logic denoted TLC interpreted over linearizations it is unrelated to our logic which seems naturally to earn the name "TLC*".

The formulas of TLC(Σ, I) — introduced in [1] with a slightly different syntax — is then the set of formulas of TLC*(Σ, I) where each of the chain operators X, U, G, X^-, U^- is immediately preceded by a chain quantifier E. As TLC will play a prominent role in this paper we will bring out its definition in more detail. More precisely, the set of formulas is given as:

$$\begin{aligned}\mathrm{TLC}(\Sigma, I) ::= p_a \mid {\sim}\alpha \mid \alpha \vee \beta \mid EX(\alpha) \mid EU(\alpha, \beta) \mid \\ EG(\alpha) \mid EX^-(\alpha) \mid EU^-(\alpha, \beta) \mid \mathrm{co}(\alpha),\end{aligned}$$

where $a \in \Sigma$. The semantics is inherited directly from TLC* in the obvious manner, so notions of definability etc. are carried over directly. It can be shown that our extension TLC* remains decidable. In work to appear we construct an elementary-time decision procedure for TLC* by means of Büchi automata.

Hence while the formulas of TLC are basically the well-known operators of the branching-time logic CTL [2] augmented with symmetrical past operators and concurrency information, the operators of TLC* are basically the well-known operators of CTL*[2] similarly extended with past quantifiers in a restricted fashion as well as concurrency information. The crucial difference is that while CTL and CTL* are branching-time logics interpreted over Kripke structures, TLC and TLC* are linear time temporal logics on traces interpreted over the underlying Hasse diagrams of the partial orders.

One of the weaknesses of TLC is that it doesn't directly facilitate reasoning about causal relationships of the individual events of the causal chains at hand. As a consequence, a number of interesting properties are not (either easily or at all) expressible within TLC. Section 5 provides a formal proof of this claim, but we will in the following bring out another such property which is very natural.

Suppose that a and b are actions representing the acquiring and releasing, respectively, of some resource. A relevant property of this system is then whether or not there exists some causal chain in the execution of the system — presumably containing other system actions than $\{a, b\}$ — such that the a's and b's alternate strictly until the task is perhaps eventually completed. Via the future chain formula $\phi_{xy} = p_x \rightarrow X({\sim}(p_x \vee p_y)U(p_y))$ we can easily express this property in TLC* by $E(G(\phi_{ab} \wedge \phi_{ba}))$. The point is here that TLC* allows us to investigate each causal chain in mention by a causal chain formula, which is then confined to this very chain. This is not possible in TLC, as the existential quantifications interpreted at some fixed event of the chain would potentially consider *all* causal chains originating at this event — not just the one presently being investigated.

We conclude with two important notions relating to TLC. Firstly, let α be a formula of TLC(Σ, I). The *operator depth* of α is defined inductively as follows: $od(p_a) = 0$, $od({\sim}\alpha) = od(\alpha)$, $od(\alpha \vee \beta) = \max(od(\alpha), od(\beta))$, $od(EX(\alpha)) = od(EG(\alpha)) = od(EX^-(\alpha)) = od(\mathrm{co}(\alpha)) = 1 + od(\alpha)$ and $od(EU(\alpha, \beta)) = od(EU^-(\alpha, \beta)) = 1 + \max(od(\alpha), od(\beta))$. The set of formulas of operator depth k is denoted by $OD(k)$.

Given $T_0, T_1 \in TR(\Sigma, I)$ and e_i events of T_i we define that $(T_0, e_0) \equiv_n (T_1, e_1)$ if for any formula $\alpha \in \mathrm{TLC}(\Sigma, I)$ with $od(\alpha) \leq n$, $T_0, e_0 \models \alpha$ iff $T_1, e_1 \models \alpha$, i.e. both structures agree on all subformulas of operator depth at most $n \geq 0$. It is then not hard to see that $(T_0, e_0) \equiv_0 (T_1, e_1)$ iff e_0 and e_1 are identically labelled, i.e. either $\lambda(e_0) = \lambda(e_1)$ or $e_0 = e_1 = \bot$.

4 An Ehrenfeucht-Fraïssé Game for TLC

In this section we will present an Ehrenfeucht-Fraïssé game to capture the expressive power of TLC. The game is played directly on the poset representation of (finite or infinite) Mazurkiewicz traces and it is similar in spirit to the Ehrenfeucht-Fraïssé game for LTL introduced by Etessami and Wilke [5]. We extend their approach to the richer setting of traces by highlighting current causal chains in the until-based moves and adding past- and co-moves.

The EF-TLC game is a game played between two persons, Spoiler and Preserver, on a pair of traces (T_0, T_1). The game is played over k rounds starting from an initial game state (e_0, e_1) and after each round the current game state is a pair of events (e'_0, e'_1) with $e'_i \in E_i$. Each round starts with the game in some specific initial game state (e_0, e_1) and Spoiler chooses one of the moves defined below and the game proceeds accordingly:

EX**-Move:** This move can only be played by Spoiler if there exists an $e'_0 \in E_0$ such that $e_0 \lessdot e'_0$ or there exists an $e'_1 \in E_1$ such that $e_1 \lessdot e'_1$. Spoiler then wins the game in case there either exists no $e'_0 \in E_0$ such that $e_0 \lessdot e'_0$ or no $e'_1 \in E_1$ such that $e_1 \lessdot e'_1$. Otherwise (in which case both e_0 and e_1 has $\lessdot$-successors) the game proceeds as follows: *(1)* Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e_i \lessdot e'_i$. *(2)* Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ such that $e_{1-i} \lessdot e'_{1-i}$. *(3)* The new game state is now (e'_0, e'_1).

EU**-Move:** *(1)* Spoiler chooses $i \in \{0, 1\}$, and an event $e'_i \in E_i$ such that $e_i \leq e'_i$ and he highlights a future causal chain $(e_i = f_i^0, f_i^1, \ldots, f_i^n = e'_i)$ with $n \geq 0$. *(2)* Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ with $e_{1-i} \leq e'_{1-i}$ such that if $e_i = e'_i$ then $e_{1-i} = e'_{1-i}$. Furthermore she highlights a future causal chain $(e_{1-i} = f_{1-i}^0, f_{1-i}^1, \cdots, f_{1-i}^m = e'_{1-i})$ with $m \geq 0$. *(3)* Spoiler now chooses *one* of the following two steps: *(3a)* Spoiler sets the game state to (e'_0, e'_1). *(3b)* Spoiler chooses an event $f_{1-i} \in \{f_{1-i}^0, f_{1-i}^1 \cdots f_{1-i}^m\}$. Preserver responds with an event $f_i \in \{f_i^0, f_i^1 \ldots f_i^n\}$ and the game continues in the state (f_0, f_1).

EG**-Move:** *(1)* Spoiler chooses $i \in \{0, 1\}$, and highlights a maximal future causal chain $(e_i = f_i^0, f_i^1, \ldots, f_i^n, \ldots)$ with $f_i^j \in E_i$ and $n \geq 0$. *(2)* Preserver responds by highlighting a maximal future causal chain $(e_{1-i} = f_{1-i}^0, f_{1-i}^1, \ldots, f_{1-i}^m, \ldots)$ with $f_i^j \in E_{1-i}$ and $m \geq 0$. *(3)* Spoiler chooses an event $f_{1-i} \in \{f_{1-i}^0, f_{1-i}^1 \cdots f_{1-i}^m\}$. Preserver responds with an event $f_i \in \{f_i^0, f_i^1 \ldots f_i^n\}$ and the game continues in the state (f_0, f_1).

co**-Move:** This move can only be played by Spoiler if there exists an $e'_0 \in E_0$ such that e_0 *co* e'_0 or there exists an $e'_1 \in E_1$ such that e_1 *co* e'_1. Spoiler then wins the game in case there either exists no $e'_0 \in E_0$ such that e_0 *co* e'_0

or no $e'_1 \in E_1$ such that e_1 *co* e'_1. Otherwise (in which case both e_0 and e_1 have concurrent events) the game proceeds as follows: *(1)* Spoiler chooses $i \in \{0,1\}$, and an event $e'_i \in E_i$ such that e_i *co* e'_i in T_i. *(2)* Preserver responds by choosing an event $e'_{1-i} \in E_{1-i}$ such that e_{1-i} *co* e'_{1-i} in T_{1-i}. *(3)* The new game state is now (e'_0, e'_1).

There are analogous EX^-- and EU^--moves. Here and throughout we refer to the full version for more details [7].

In the 0-round game Spoiler wins if $(T_0, e_0) \not\equiv_0 (T_1, e_1)$ and otherwise Preserver wins. In the $(k+1)$-round game Spoiler wins if $(T_0, e_0) \not\equiv_0 (T_1, e_1)$. If it is the case that $(T_0, e_0) \equiv_0 (T_1, e_1)$, a round is played according to the above moves. This round either results in a win for Spoiler (e.g. by the EX-move) or a new game state (e'_0, e'_1). In the latter case, a k-round game is then played starting from the initial game state (e'_0, e'_1).

We say that *Preserver has a winning strategy* in the k-round game on (T_0, e_0) and (T_1, e_1), denoted $(T_0, e_0) \sim_k (T_1, e_1)$, if she can win the k-round game on the structures T_0 and T_1 starting in the initial game state (e_0, e_1) no matter which moves are performed by Spoiler. If not, we say that *Spoiler has a winning strategy.* We refer to [5] for basic intuitions about the game.

Our interest in the game lies in the following fact.

Proposition 1. *For every $k \geq 0$, $(T_0, e_0) \sim_k (T_1, e_1)$ iff $(T_0, e_0) \equiv_k (T_1, e_1)$.*

Proof. We prove that $(T_0, e_0) \sim_k (T_1, e_1)$ iff $(T_0, e_0) \equiv_k (T_1, e_1)$ by induction on k. The base case where $k = 0$ follows trivially from the definition.

For the inductive step suppose that the claim is true for k. We first prove the direction from left to right. Suppose that $(T_0, e_0) \sim_{k+1} (T_1, e_1)$. Let $\alpha \in \mathrm{TLC}(\Sigma, I)$ with $od(\alpha) = k+1$. We must show that $T_0, e_0 \models \alpha$ iff $T_1, e_1 \models \alpha$. It suffices to prove the statement when the top-level connective of α is a chain-operator because by boolean combinations (T_0, e_0) and (T_1, e_1) would then agree on all formulas of operator depth $k+1$. We will only consider the case where the top-level chain-operator is EU. The other cases follow similarly.

Suppose now $\alpha = EU(\beta, \beta')$. Assume without loss of generality that $T_0, e_0 \models \alpha$, i.e. there exists a future causal chain $\rho^0 = (f_0^0, f_0^1, \ldots, f_0^n)$ with $e_0 = f_0^0$ and $f_0^n = e'_0$ such that $T_0, f_0^j \models \beta$ for each $0 \leq j < n$ and $T_0, e'_0 \models \beta'$. Hence we let Spoiler play the EU-move on T_0 and make him highlight ρ^0 on T_0. Preserver now uses her winning strategy and highlights $\rho^1 = (f_1^0, f_1^1, \ldots, f_1^m)$ with $e_1 = f_1^0$ and $f_1^m = e'_1$. Two subcases now arise.

Assume first that Spoiler sets the new game state to (e'_0, e'_1). As e'_1 was chosen from Preserver's winning strategy we have that $(T_0, e'_0) \sim_k (T_1, e'_1)$ which by induction hypothesis implies that $(T_0, e'_0) \equiv_k (T_1, e'_1)$. Thus $T_1, e'_1 \models \beta'$. Now, assume that Spoiler instead picked an event f_1 on ρ^1. By Preserver's winning strategy she could pick an event f_0 on ρ^0 (This is possible due to the requirement that if $e_0 = e'_0$ then $e_1 = e'_1$). Again by the winning strategy we have that $(T_0, f_0) \sim_k (T_1, f_1)$ and by induction hypothesis that $T_1, f_1 \models \beta$. Hence $T_1, f_1 \models EU(\beta, \beta')$, which concludes this direction of the proof.

We prove the direction from right to left by contraposition, so suppose that $(T_0, e_0) \not\sim_{k+1} (T_1, e_1)$. We will then exhibit a formula $\alpha \in \mathrm{TLC}(\Sigma, I)$ with $od(\alpha) = k+1$ such that $T_0, e_0 \models \alpha$ but $T_1, e_1 \not\models \alpha$. Again, we will only prove the case where Spoiler's first move of his winning strategy is either the *EU*-move. The other cases either follows in analogous or easier manners.

Suppose Spoiler plays the *EU*-move on T_0 (without loss of generality), i.e. he chooses a future causal chain $\rho^0 = (f_0^0, f_0^1, \ldots, f_0^n)$ with $e_0 = f_0^0$ and $f_0^n = e_0'$. It is not hard to show by induction that there are only a finite number of semantically inequivalent formulas α with $od(\alpha) \leq k$ and $T_0, e \models \alpha$ for any $e \in E_0$. Hence, each formula $\beta_0^j = \bigwedge\{\alpha \in OD(k) \mid T_0, f_0^j \models \alpha\} \wedge \bigwedge\{\sim\alpha \in OD(k) \mid T_0, f_0^j \not\models \alpha\}$ is well-defined and equivalent to a formula of operator depth k for each $0 \leq j < n$, so letting $\beta_{e_0'} = \beta_0^n$ we have that $\alpha = EU(\bigvee_{0 \leq j < n} \beta_0^j, \beta_{e_0'})$ is a TLC-formula with $od(\alpha) = k+1$ and by definition $T_0, e_0 \models \alpha$. We will argue that $T_1, e_1 \not\models \alpha$.

Suppose that $T_1, e_1 \models \alpha$. Then there exists a future causal chain $\rho^1 = (f_1^0, f_1^1, \ldots, f_1^m)$ with $e_1 = f_1^0$ and $f_1^m = e_1'$ such that $T_1, f_1^l \models \bigvee_{0 \leq j < n} \beta_0^j$ for each $0 \leq l < m$ and $T_1, e_1' \models \beta_{e_0'}$.

Assume first that Spoiler chooses to set the new game state to (e_0', e_1') by following his winning strategy. As $T_1, e_1' \models \beta_{e_0'}$ it must be the case that for each $\gamma \in OD(k)$, $T_0, e_0' \models \gamma$ iff $T_1, e_1' \models \gamma$. By induction hypothesis $(T_0, e_0') {\sim_k} (T_1, e_1')$ which contradicts that Spoiler has a winning strategy because Preserver could initially have played ρ^1 as above and continued according to $(T_0, e_0') {\sim_k} (T_1, e_1')$.

Now assume that Spoiler instead by his winning strategy picks an event f_1 on ρ^1. Then $T_1, f_1 \models \beta_0^j$ for some $0 \leq j < n$ as $T_1, f_1 \models \bigvee_{0 \leq j < n} \beta_0^j$. Again by induction hypothesis we know that $(T_0, f_0^j) {\sim_k} (T_1, f_1)$ which again contradicts that Spoiler has a winning strategy because Preserver could respond by picking $f_0^j \in E_0$ and continue from the game state (f_0^j, f_1) according to $(T_0, f_0^j) {\sim_k} (T_1, f_1)$.

Hence $T_1, e_1 \not\models \alpha$ as required. □

5 An Undefinability Result

In this section we will give an example of a natural property which we, by means of the game characterization of the previous section, will show is not definable in TLC. Let (Σ, I) be a trace alphabet with $\{a, b, c\} \subseteq \Sigma$ such that $a\ D\ c$ and $c\ D\ b$ but $a\ I\ b$. Consider $L = [abcabc]^* \subseteq TR(\Sigma, I)$.

Lemma 2. *L is not definable in* $\mathrm{TLC}(\Sigma, I)$.

Proof. Let $k \geq 0$ be given and consider $T_0^k = [abc]^{4k}$ and $T_1^k = [abc]^{4k+1}$. It suffices to show that $(T_0^k, \bot) {\sim_k} (T_1^k, \bot)$. By Proposition 1 it then follows that $(T_0^k, \bot) {\equiv_k} (T_1^k, \bot)$. Suppose L would be definable by a TLC-formula α of operator depth n. In particular then $(T_0^n, \bot) {\equiv_n} (T_1^n, \bot)$. However, by definition it must be the case that $T_0^n \in L$ and $T_1^n \notin L$, contradicting that T_0^n and T_1^n satisfy the same set of formulas of operator depth at most n. Hence, L cannot be expressed by any formula of TLC assuming $(T_0^k, \bot) {\sim_k} (T_1^k, \bot)$ holds for any $k \geq 0$. The remainder

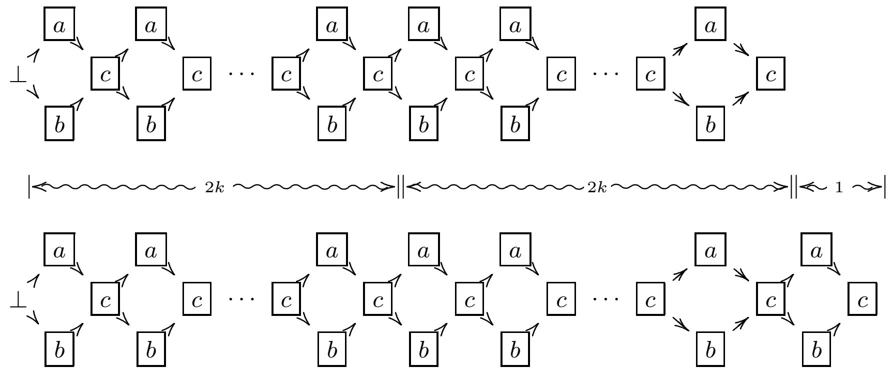


Fig. 1. T_0^k (top) and T_1^k (bottom) on which the game is played.

of the proof will be devoted to showing that it is the case that $(T_0^k, \bot) {\sim_k} (T_1^k, \bot)$. To bring this out we need a few definitions. As depicted in Figure 1 the game is played on T_0^k and T_1^k consisting of $4k$ and $4k+1$ copies of the trace factor $[abc]$, respectively. The section of e_i for $i \in \{0,1\}$ is then defined to be the number of the enclosing $[abc]$-factor in T_k^i counting from left and starting with 1. We denote this number by $sect(e_i)$. In case $e_i = \bot$ we set $sect(e_i) = 0$. Furthermore, we say that e_0 and e_1 are *position equivalent*, in case either $(e_0, e_1) = (\bot, \bot)$ or $\lambda(e_0) = \lambda(e_1)$. From the definition of T_0 and T_1 it follows that e_0 and e_1 are position equivalent in case e_0 and e_1 denote the same local positions in two (possibly distinct) sections of T_0 and T_1, respectively. The unique event of section $s \geq 1$ labelled with letter $x \in \{a, b, c\}$ in T_i^k will be denoted $e_i^{x,s}$. For example, the fourth b-labelled event of T_0^k is denoted $e_0^{b,4}$.

We will then show that Preserver has a strategy such that after $k' \leq k$ rounds played on (T_0^k, T_1^k) with current game state (e_0, e_1), the following invariant holds:

(i) e_0 and e_1 are position equivalent.
(ii) $sect(e_0) = sect(e_1)$ or $sect(e_0) = sect(e_1) - 1$.
(iii) $sect(e_0) = sect(e_1)$ implies $sect(e_0) \leq 2(k + k')$.
(iv) $sect(e_0) = sect(e_1) - 1$ implies $sect(e_0) \geq 2(k - k') + 1$.

We prove that the invariant holds by induction on k'. It is trivial to observe, that in the base case we have that $(e_0, e_1) = (\bot, \bot)$, $sect(e_0) = sect(e_1) = 0$ and $k' = 0$ thus satisfying (i),(ii), (iii) and (iv) above.

For the inductive step, assume that the statement holds for $k' < k$. From (i) it follows that $(T_0, e_0) \equiv_0 (T_1, e_1)$, so a next round is played. We then show that the Preserver can move so as to maintain the invariant for the next game state (e_0', e_1') by case analysis on the next move chosen by Spoiler. We only consider the case for the *EU*-move. The other moves follow analogously. From (ii) we know that $sect(e_0) = sect(e_1)$ or $sect(e_0) = sect(e_1) - 1$, so two subcases arise. Subcase I: $sect(e_0) = sect(e_1)$. Suppose Spoiler chooses to play the EU-move on T_0^k and highlights a future causal chain $\rho^0 = (e_0 = e_0^{x_0,s_0}, e_0^{x_1,s_1}, \dots, e_0^{x_n,s_n} = e_0')$. By assumption $sect(e_0) \leq 2(k + k')$.

Suppose first that $s_n \leq 2(k+k'+1)$. Then Preserver can just copy the move and respond with $\rho^1 = (e_1 = e_1^{x_0,s_0}, e_1^{x_1,s_1}, \dots, e_1^{x_n,s_n} = e_1')$. If Spoiler chooses to set the new game state to (e_0', e_1'), $sect(e_0') = sect(e_1') \leq 2(k + k' + 1)$ and the invariant is maintained. If Spoiler instead chooses to pick an event $e_1^{x_i,s_i}$, Preserver would respond by picking $e_0^{x_i,s_i}$ and the invariant is maintained in a similar manner.

Suppose then that $s_n > 2(k + k' + 1)$. Preserver must then "insert" an additional occurrence of a section into ρ^0 at section $2(k + k' + 1)$. To bring this out, let l be the least index such that $s_l = 2(k + k' + 1)$, which exists by assumption. Preserver then responds with

$$\rho^1 = (e_1^{x_0,s_0}, \dots, e_1^{x_l,s_l}, e_1^{x_{l+1},s_{l+1}}, e_1^{x_l,s_l+1}, e_1^{x_{l+1},s_{l+1}+1}, e_1^{x_{l+2},s_{l+2}+1}, \dots, e_1^{x_n,s_n+1})$$

with $e_1' = e_1^{x_n,s_n+1}$. If Spoiler chooses to set the new game state to (e_0', e_1'), $sect(e_0') = s_n = sect(e_1') - 1$. However, the invariant is maintained as $sect(e_0') \geq 2(k + k' + 1) \geq 2(k - (k' + 1)) + 1$. If Spoiler instead chooses to pick an event on ρ^1, Preserver responds dependent upon its index. If Spoiler picks one of the first $l + 2$ events $e_1^{x_i,s_i}$, Preserver responds with $e_0^{x_i,s_i}$. As $sect(e_0^{x_i,s_i}) = s_i = sect(e_1^{x_i,s_i}) \leq 2(k + k' + 1)$ the invariant is maintained. If Spoiler picks one of the remaining events $e_1^{x_i,s_i+1}$, Preserver responds with $e_0^{x_i,s_i}$ in which case $sect(e_0^{x_i,s_i}) = s_i = sect(e_1^{x_i,s_i+1}) - 1$ and the invariant is maintained as $sect(e_0^{x_i,s_i}) \geq 2(k + k' + 1) > 2(k - (k' + 1)) + 1$.

Suppose spoiler chooses to play the EU-move on T_1^k and highlights a future causal chain $\rho^1 = (e_1 = e_1^{x_0,s_0}, e_1^{x_1,s_1}, \dots, e_1^{x_n,s_n} = e_1')$. By assumption $sect(e_0) \leq 2(k + k')$. If $s_n \leq 2(k + k' + 1)$ then Preserver can, as above, just copy the move and maintain the invariant, so suppose that $s_n > 2(k + k' + 1)$. Preserver must then "chop" a duplicate occurrence off ρ^1 around the sections $2(k+k')+1, 2(k+k')+2 = 2(k+k'+1), 2(k+k')+3$ which exist by construction. Any causal chain passing through these three sections must pass (at least) two identical ac-labelled or bc-labelled stretches. Now, let l be the least index such that $s_l = 2(k+k')+1$ and consider the sequence $\sigma = (x_l, x_{l+2}, x_{l+4})$ with $\lambda(\sigma) \in \{a, b\}^3$. Remove from σ the first occurrence x_i where there exists an $j > i$ with x_j in σ and $x_i = x_j$. Let $\sigma' = (x_p, x_q)$ denote the resulting sequence where $p, q \in \{l, l + 2, l + 4\}$. Preserver then plays the chain ρ^0:

$$(e_0^{x_0,s_0}, \dots, e_0^{x_{l-1},s_{l-1}}, e_0^{x_p,s_l}, e_0^{c,s_{l+1}}, e_0^{x_q,s_{l+2}}, e_0^{c,s_{l+3}}, e_0^{x_{l+5},s_{l+5}-1}, \dots, e_0^{x_n,s_n-1})$$

with $e_0' = e_0^{x_n,s_n-1}$. If Spoiler chooses to set the new game state to (e_0', e_1') then $sect(e_0') = s_n = sect(e_1') - 1$ so the invariant is maintained because $s_n > 2(k + k' + 1) > 2(k - (k' + 1)) + 1$. If Spoiler chooses to pick an event on ρ^0, Preserver responds according to one of several cases. If Spoiler picks one of the first l events $e_0^{x_i,s_i}$ then Preserver picks $e_1^{x_i,s_i}$ and the invariant is maintained as usual. If Spoiler picks either $e_0^{c,s_{l+1}}$ or $e_0^{c,s_{l+3}}$ then Preserver picks either $e_1^{c,s_{l+1}}$

or $e_1^{c,s_{l+3}}$, respectively. As the sections are both s_{l+1} or both s_{l+3} and $s_{l+1} < s_{l+3} = s_l + 1 = 2(k+k'+1)$ the invariant follows. If Spoiler picks an event, $e_0^{x_m,s}$ say, in $\{e_0^{x_p,s_l}, e_0^{x_q,s_{l+2}}\}$ *before* the removed occurrence in σ then $m \in \{l, l+2\}$ and Preserver responds by $e_1^{x_m,s}$. Then $sect(e_0^{x_m,s}) = s = sect(e_1^{x_m,s}) \leq s_{l+2} = 2(k+k'+1)$. Similarly, if $e_0^{x_m,s}$ occurs *after* the removed occurrence then $m \in \{l+2, l+4\}$ and Preserver picks $e_1^{x_m,s+1}$. Then $sect(e_0^{x_m,s}) = s = sect(e_1^{x^m,s+1}) - 1 \geq 2(k+k') > 2(k-(k'+1))+1$ and in both cases the invariant is maintained. Finally, if Spoiler picks one of the remaining events $e_0^{x_i,s_i-1}$ with $i \geq l+5$ then Preserver responds with $e_1^{x_i,s_i}$. As $sect(e_0^{x_i,s_i-1}) = sect(e_1^{x_i,s_i}) - 1 \geq 2(k-(k'+1)) + 1$ the invariant is also maintained in this case.

Subcase II: $sect(e_0) = sect(e_1) - 1$. Here the futures of e_0 in T_0^k and e_1 in T_1^k both consist of $4k - sect(e_0)$ factors of $[abc]$ and are identical with respect to future moves. Hence Preserver can just "copy" the move made by Spoiler. □

6 The Expressiveness of TLC*

Let (Σ, I) be any trace alphabet with $\{a, b, c\} \subseteq \Sigma$ such that $a\ D\ c$ and $c\ D\ b$ but $a\ I\ b$. Consider $L = [abcabc]^* \subseteq TR(\Sigma, I)$ from the previous section.

Lemma 3. *L is definable in* TLC*(Σ, I).

Proof. Our proof will in fact show that the future fragment of TLC with only *one* future chain quantifier of TLC* suffices to express L. First define

$$\alpha_{[abc]^*} = AG(\bigwedge_{d \in \Sigma - \{a,b,c\}} \sim p_d) \wedge EX(p_a \wedge EX(p_c)) \wedge EX(p_b \wedge EX(p_c)) \wedge$$
$$AG(p_c \wedge EX(\mathrm{tt}) \rightarrow EX(p_a \wedge EX(p_c)) \wedge EX(p_b \wedge EX(p_c)).$$

It is easy to see that $T \models \alpha_{[abc]^*}$ iff $T \in [abc]^*$. We will then use existence of "zig-zagging" future causal chains to restrict to $[abcabc]^* \subset [abc]^*$ below. Define the future chain formula $\phi_{(acbc)^*}$ as follows.

$$\phi_{(acbc)^*} = p_a \wedge G(p_a \rightarrow X(p_c \wedge X(p_b \wedge X(p_c \wedge (\sim X\mathrm{tt} \vee Xp_a))))).$$

It's easy to see that $T, e \models E(\phi_{(acbc)^*})$ iff there exists a future causal chain ρ rooted at e such that $\lambda(\rho) \in (acbc)^* \subseteq \Sigma^*$. The statement of the lemma now follows by taking $\alpha_L = \alpha_{[abc]^*} \wedge (\sim EX\mathrm{tt} \vee EX(E(\phi_{(acbc)^*})))$. □

Putting all the pieces together, we can now state and prove the main result of the paper.

Theorem 4. *Let (Σ, I) be any trace alphabet. Then*

1. TLC(Σ, I) = TLC*(Σ, I) *if D is transitive.*
2. TLC$(\Sigma, I) \subset$ TLC*(Σ, I) *if D is not transitive.*

Proof. Obviously $\mathrm{TLC}(\Sigma, I) \subseteq \mathrm{TLC}^*(\Sigma, I)$, so (2) follows easily from Lemma 2 and Lemma 3 as $(a, c), (c, b) \in D$ but $(a, b) \notin D$ witness that D is not transitive. Hence it suffices to prove (1).

Let (Σ, I) be a trace alphabet with D transitive, i.e. the graph (Σ, D) is a disjoint union of cliques $\{C_i\}_{i=1}^n$. Thus any trace $T \in TR(\Sigma, I)$ consists of disjoint C_i-labelled causal chains only initially connected by $\perp$. We can then define three mutually inductive translations $||\cdot||_{ev}$, $||\cdot||_{ch}^+$ and $||\cdot||_{ch}^-$ converting event formulas, future chain formulas and past chain formulas, respectively, of $\mathrm{TLC}^*(\Sigma, I)$ to formulas of $\mathrm{TLC}(\Sigma, I)$ as follows.

- $||p_a||_{ev} = p_a$ and the boolean connectives are as expected.
- $||\mathrm{co}(\alpha)||_{ev} = \mathrm{co}(||\alpha||_{ev})$.
- $||E(\phi)||_{ev} = ||\phi||_{ch}^+$ and $||E^-(\phi)||_{ev} = ||\phi||_{ch}^-$.
- $||\alpha||_{ch}^+ = ||\alpha||_{ch}^- = ||\alpha||_{ev}$ and the boolean connectives are as expected.
- $||X\phi||_{ch}^+ = EX(||\phi||_{ch}^+)$ and $||X^-\phi||_{ch}^- = EX^-(||\phi||_{ch}^-)$.
- $||\phi U \psi||_{ch}^+ = EU(||\phi||_{ch}^+, ||\psi||_{ch}^+)$ and $||\phi U^- \psi||_{ch}^- = EU(||\phi||_{ch}^-, ||\psi||_{ch}^-)$.

By nested inductions one can show that for each $\alpha \in \mathrm{TLC}^*(\Sigma, I)$, $T, e \models \alpha$ iff $T, e \models ||\alpha||_{ev}$. As $||\alpha||_{ev} \in \mathrm{TLC}(\Sigma, I)$ the required conclusion follows. □

It's easy to show that $L \notin \mathrm{FO}(\Sigma, I)$ so as a simple corollary we can conclude that $\mathrm{TLC}^*(\Sigma, I)$ is not in general included in $\mathrm{FO}(\Sigma, I)$ even though they're expressively equivalent in the sequential case where $I = \emptyset$.

References

1. Alur, R., Peled, D., Penczek, W.: Model checking of causality properties. Proceedings of LICS'95, IEEE Computer Society Press (1995) 90–100
2. Clarke, E. M., Emerson, E. A., Sistla, A. P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems **8**(2) (1986) 244–263
3. Diekert, V., Gastin, P.: An expressively complete temporal logic without past tense operators for Mazurkiewicz traces. Proceedings of CSL'99, LNCS **1683**, Springer-Verlag (1999)
4. Diekert, V., Rozenberg, G. (eds.): The book of traces. World Scientific (1995)
5. Etessami, K., Wilke, Th.: An until hierarchy for temporal logic. Proceedings of LICS'96, IEEE Computer Society Press (1996) 108–117
6. Godefroid, P.: Partial-order methods for the verification of concurrent systems. LNCS **1032**, Springer-Verlag (1996)
7. Henriksen, J. G.: An Expressive Extension of TLC. Technical report RS-99-26, BRICS, Department of Computer Science, University of Aarhus (1999)
8. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Report PB-78, Department of Computer Science, University of Aarhus, Denmark (1977)
9. Niebert, P.: A temporal logic for the specification and validation of distributed behaviour. Ph.D. thesis, University of Hildesheim (1997)
10. Peled, D.: Partial order reduction: model checking using representatives. Proceedings of MFCS'96, LNCS **1113**, Springer-Verlag (1996) 93–112
11. Pnueli, A.: The temporal logic of programs. Proceedings of FOCS'77, IEEE Computer Society Press (1977) 46–57

12. Ramanujam, R.: Locally linear time temporal logic. Proceedings of LICS'96, IEEE Computer Society Press (1996) 118–127
13. Thiagarajan, P. S.: A trace based extension of linear time temporal logic. Proceedings of LICS'94, IEEE Computer Society Press (1994) 438–447
14. Thiagarajan, P. S., Henriksen, J. G.: Distributed versions of linear time temporal logic: A trace perspective. In Reisig and Rozenberg (Eds.), Lectures on Petri Nets I: Basic Models, LNCS **1491**, Springer-Verlag (1998) 643–681
15. Thiagarajan, P. S., Walukiewicz, I.: An expressively complete linear time temporal logic for Mazurkiewicz traces. Proceedings of LICS'97, IEEE Computer Society Press (1997) 183–194
16. Walukiewicz, I.: Difficult configurations — on the complexity of LTrL (extended abstract). Proceedings of ICALP'98, LNCS **1443**, Springer-Verlag (1998) 140–151

Completeness and Decidability of a Fragment of Duration Calculus with Iteration

Dang Van Hung* and Dimitar P. Guelev**

International Institute for Software Technology
The United Nations University, P.O.Box 3058, Macau
{dvh, dg}@iist.unu.edu

Abstract. Duration Calculus with Iteration (DC*) has been used as an interface between original Duration Calculus and Timed Automata, but has not been studied rigorously. In this paper, we study a subset of DC* formulas consisting of so-called simple ones which corresponds precisely with the class of Timed Automata. We give a complete proof system and the decidability results for the subset.

1 Introduction

Duration Calculus (DC) was introduced by Zhou, Hoare and Ravn in 1991 as a logic to specify the requirements for real-time systems. DC has been used successfully in many case studies. In [4], we have developed a method for designing a real-time hybrid system from its specification in DC. In that paper, we introduced a class of so-called simple Duration Calculus formulas with iterations which corresponds precisely with the class of real-time automata to express the design of real-time hybrid systems, and show how to derive a design in this language from a specification in the original Duration Calculus. We use the definition of semantic of our design language to reason about the correctness of our design. However, it would be more practical and interesting if the correctness of a design can be proved syntactically with a tool. Therefore, developing a proof system to assist the formal verification of the design plays an important role in making the use of formal methods for the designing process of real-time systems. This is our aim in this paper.

We achieve our aim in the following way. First we extend DC with the iteration operator (*) to obtain a logic called DC*, and define a subclass of DC* formulas called simple DC* formulas to express the designs. Secondly we develop a complete proof system for the proof of the fact that a simple DC* formula D implies a DC formula S, meaning that any implication of this form can be proved in our proof system.

To illustrate our idea, let us consider a classical simple example Gas Burner taken from [14]. The time critical requirements of a gas burner is specified by a

* On leave from the Institute of Information Technology, Hanoi, Vietnam.
** On leave from the Department of Mathematical Logic and Its Applications, Faculty of Mathematics and Informatics, Sofia University "St. Kliment Ochridski"

DC formula denoted by S, defined as $\Box(\ell > 60s \Rightarrow (20 * \int leak \leq \ell))$ which says that during the operation of the system, if the interval over which the system is observed is at least 1 min, the proportion of time spent in the leak state is not more than one-twentieth of the elapsed time.

One can design the Gas Burner as a real-time automaton depicted in Fig. 1 which expresses that any leak must be detected and stopped within one second, and that leak must be separated by at least $30s$. A natural way to express the behaviour of the automaton is to use a classical regular expression like notation

$$D \ \widehat{=}\ ((\lceil leak \rceil \wedge \ell \leq 1)^\frown(\lceil nonleak \rceil \wedge \ell \geq 30))^* \,.$$

Here we assume that the gas burner starts from the leak state. We will see later that D is a DC formula with iteration. It expresses not only the temporal order of states but also the time constraints on the state periods.

By using our complete proof system we can show formally the implication $D \Rightarrow S$ which expresses naturally the correctness of the design.

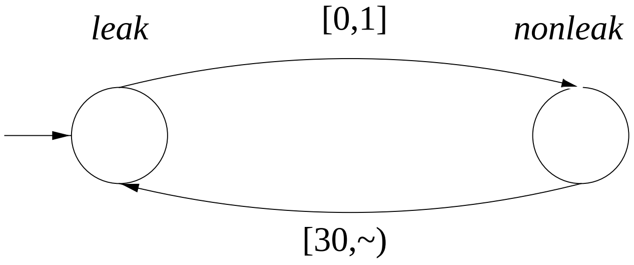


Fig. 1. Simple design of Gas Burner

The class of simple DC* formulas has an interesting property that it is decidable, which means that we can decide if a design is implementable. Furthermore, for some class of DC formulas such as linear duration invariants (see [15,11,5]), the implication from a simple DC* formula to a formula in the class can be checked by a simple algorithm.

The paper is organised as follows. In the next section, we give the syntax and semantics of our Duration Calculus with Iteration. In the third section we will give a proof system for the calculus. We prove the completeness of our proof system for the class of simple DC* formulas in Section 4. The decidability of the class will be discussed in the last section.

2 Duration Calculus with Iteration

This section presents the formal definition of Duration Calculus with iteration, which is a conservative extension of Duration Calculus [14].

A language for DC* is built starting from the following sets of *symbols*: a set of *constant symbols* $\{a, b, c, \ldots\}$, a set of *individual variables* $\{x, y, z, \ldots\}$, a set of *state variables* $\{P, Q, \ldots\}$, a set of *temporal variables* $\{u, v, \ldots\}$, a set of *function symbols* $\{f, g, \ldots\}$, a set of *relation symbols* $\{R, U, \ldots\}$, and a set of *propositional temporal letters* $\{A, B, \ldots\}$. These sets are required to be pairwise disjoint and disjoint with the set $\{\mathbf{0}, \bot, \neg, \vee, ^\frown, ^*, \exists, \int, (,)\}$. Besides, 0 should be one of the constant symbols; $+$ should be a binary function symbol; $=$ and $\leq$ should be binary relation symbols.

Given the sets of symbols, a DC* language definition is essentially that of the sets of *state expressions* S, *terms* t and *formulas* φ of the language. These sets can be defined by the following BNFs:

$$\begin{array}{l} S \mathrel{\hat{=}} \mathbf{0} \mid P \mid \neg S \mid S \vee S \\ t \mathrel{\hat{=}} c \mid x \mid u \mid \int S \mid f(t, \ldots, t) \\ \varphi \mathrel{\hat{=}} A \mid R(t, \ldots, t) \mid \neg\varphi \mid (\varphi \vee \varphi) \mid (\varphi ^\frown \varphi) \mid (\varphi^*) \mid \exists x \varphi \end{array}$$

Terms and formulas that have no occurrences of $^\frown$ (*chop*), nor of temporal variables, or $\int$, are called *rigid.*

The linearly ordered field of the real numbers,

$$\langle \mathbf{R}, =_{\mathbf{R}}, 0_{\mathbf{R}}, 1_{\mathbf{R}}, +_{\mathbf{R}}, -_{\mathbf{R}}, \times_{\mathbf{R}}, /_{\mathbf{R}}, \leq_{\mathbf{R}} \rangle ,$$

is the most important component of DC semantics. We denote by $\mathbf{I}$ the set of the bounded intervals over $\mathbf{R}$, $\{[\tau_1, \tau_2] \mid \tau_1, \tau_2 \in \mathbf{R}, \tau_1 \leq_{\mathbf{R}} \tau_2\}$. For a set $A \subseteq \mathbf{R}$, we denote by $\mathbf{I}(A)$ the set $\{[\tau_1, \tau_2] \in \mathbf{I} \mid \tau_1, \tau_2 \in A\}$ of intervals with end-points in A.

Given a DC* language $\mathcal{L}$, a model for $\mathcal{L}$ is an *interpretation* $\mathcal{I}$ of the symbols of $\mathcal{L}$ that satisfies the following conditions: $\mathcal{I}(c), \mathcal{I}(x) \in \mathbf{R}$ for constant symbols c and individual variables x; $\mathcal{I}(f) : \mathbf{R}^n \to \mathbf{R}$ for n-place function symbols f; $\mathcal{I}(v) : \mathbf{I} \to \mathbf{R}$ for temporal variables v; $\mathcal{I}(R) : \mathbf{R}^n \to \{0, 1\}$ for n-place relation symbols R; $\mathcal{I}(P) : \mathbf{R} \to \{0, 1\}$ for state variable P, and $\mathcal{I}(A) : \mathbf{I} \to \{0, 1\}$ for temporal propositional letters A. Besides, $\mathcal{I}(0) = 0_{\mathbf{R}}$, $\mathcal{I}(+) = +_{\mathbf{R}}$, $\mathcal{I}(=)$ is $=_{\mathbf{R}}$, and $\mathcal{I}(\leq)$ is $\leq_{\mathbf{R}}$. The following condition, known as *finite variability of state*, is imposed on interpretations: *For every* $[\tau_1, \tau_2] \in \mathbf{I}$ *such that* $\tau_1 < \tau_2$, *and every state variable* S *there exist* $\tau'_1, \ldots, \tau'_n \in \mathbf{R}$ *such that* $\tau_1 = \tau'_1 < \ldots < \tau'_n = \tau_2$ *and* $\mathcal{I}(S)$ *is constant on the intervals* (τ'_i, τ'_{i+1}), $i = 1, \ldots, n-1$.

For the rest of the paper we omit the index $._{\mathbf{R}}$, that distinguishes operations on reals from the corresponding symbols.

Definition 1. *Given a DC interpretation* $\mathcal{I}$ *for the DC* language* $\mathcal{L}$, *the meaning of state expressions* S *in* $\mathcal{L}$ *under* $\mathcal{I}$, $S_{\mathcal{I}} : \boldsymbol{R} \to \{0, 1\}$, *is defined inductively as follows: for all* $\tau \in \boldsymbol{R}$

$$\begin{array}{ll} \mathbf{0}_{\mathcal{I}}(\tau) \mathrel{\hat{=}} 0 & \\ P_{\mathcal{I}}(\tau) \mathrel{\hat{=}} \mathcal{I}(P)(\tau) & \textit{for state variables } P \\ (\neg S)_{\mathcal{I}}(\tau) \mathrel{\hat{=}} 1 - S_{\mathcal{I}}(\tau) & \\ (S_1 \vee S_2)_{\mathcal{I}}(\tau) \mathrel{\hat{=}} \max((S_1)_{\mathcal{I}}(\tau), (S_2)_{\mathcal{I}}(\tau)) & \end{array}$$

Given an interval $[\tau_1,\tau_2] \in \mathbf{I}$, the meaning of a term t in $\mathcal{L}$ under $\mathcal{I}$ is a number $\mathcal{I}_{\tau_1}^{\tau_2}(t) \in \mathbf{R}$ defined inductively as follows:

$\mathcal{I}_{\tau_1}^{\tau_2}(c) \mathrel{\widehat{=}} \mathcal{I}(c)$	*for constant symbols c,*
$\mathcal{I}_{\tau_1}^{\tau_2}(x) \mathrel{\widehat{=}} \mathcal{I}(x)$	*for individual variables x,*
$\mathcal{I}_{\tau_1}^{\tau_2}(v) \mathrel{\widehat{=}} \mathcal{I}(v)([\tau_1,\tau_2])$	*for temporal variables v,*
$\mathcal{I}_{\tau_1}^{\tau_2}(\int S) \mathrel{\widehat{=}} \int\limits_{\tau_1}^{\tau_2} S_{\mathcal{I}}(\tau)d\tau$	*for state expressions S,*
$\mathcal{I}_{\tau_1}^{\tau_2}(f(t_1,\ldots,t_n)) \mathrel{\widehat{=}} \mathcal{I}(f)(\mathcal{I}_{\tau_1}^{\tau_2}(t_1),\ldots,\mathcal{I}_{\tau_1}^{\tau_2}(t_n))$	*for n-place function symbols f.*

The definitions given so far are relevant to the semantics of DC in general. The extension to the semantics that comes with DC* appears in the definition of the $\models$ relation below. Let us recall the traditional relation on interpretations: For interpretations $\mathcal{I}$, and $\mathcal{J}$ of the symbols of the same DC* language $\mathcal{L}$ and for a symbol x in $\mathcal{L}$, we say $\mathcal{I}$ *x-agrees* with $\mathcal{J}$ iff $\mathcal{I}(s) = \mathcal{J}(s)$ for all symbols s in $\mathcal{L}$, but possibly x.

Definition 2. *Given a DC* language $\mathcal{L}$, and an interpretation $\mathcal{I}$ of the symbols of $\mathcal{L}$. The relation $\mathcal{I},[\tau_1,\tau_2] \models \varphi$ for $[\tau_1,\tau_2] \in \mathbf{I}$ and formulas φ in $\mathcal{L}$ is defined by induction on the construction of φ as follows:*

$\mathcal{I},[\tau_1,\tau_2] \not\models \bot$	
$\mathcal{I},[\tau_1,\tau_2] \models A$	*iff $\mathcal{I}(A)([\tau_1,\tau_2]) = 1$ for temporal propositional letters A*
$\mathcal{I},[\tau_1,\tau_2] \models R(\tau_1,\ldots,\tau_n)$	*iff $\mathcal{I}(R)(\mathcal{I}_{\tau_1}^{\tau_2}(t_1),\ldots,\mathcal{I}_{\tau_1}^{\tau_2}(t_n)) = 1$*
$\mathcal{I},[\tau_1,\tau_2] \models \neg\varphi$	*iff $\mathcal{I},[\tau_1,\tau_2] \not\models \varphi$*
$\mathcal{I},[\tau_1,\tau_2] \models (\varphi \vee \psi)$	*iff either $\mathcal{I},[\tau_1,\tau_2] \models \varphi$ or $\mathcal{I},[\tau_1,\tau_2] \models \psi$*
$\mathcal{I},[\tau_1,\tau_2] \models (\varphi^\frown\psi)$	*iff $\mathcal{I},[\tau_1,\tau] \models \varphi$ and $\mathcal{I},[\tau,\tau_2] \models \psi$ for some $\tau \in [\tau_1,\tau_2]$*
$\mathcal{I},[\tau_1,\tau_2] \models (\varphi^*)$	*iff either $\tau_1 = \tau_2$, or there exist $\tau_1',\ldots,\tau_n' \in \mathbf{R}$ such that $\tau_1 = \tau_1' < \ldots < \tau_n' = \tau_2$ and $\mathcal{I},[\tau_i',\tau_{i+1}'] \models \varphi$ for $i = 1,\ldots,n-1$*
$\mathcal{I},[\tau_1,\tau_2] \models \exists x\varphi$	*iff $\mathcal{J},[\tau_1,\tau_2] \models \varphi$ for some $\mathcal{J}$ that x-agrees with $\mathcal{I}$*

Note that the clauses that define the interpretation of constructs other than * in DC* are the same as in DC. This entails that DC* is a conservative extension of DC.

Let $\mathcal{I}$ be a DC interpretation, φ be a DC* formula, and $\mathbf{J}_1, \mathbf{J}_2, \mathbf{J} \subseteq \mathbf{I}$ be sets of intervals. Let $k < \omega$. We introduce the following notations for our convenience.

$$\tilde{\mathcal{I}}(\varphi) \mathrel{\widehat{=}} \{[\tau_1,\tau_2] \in \mathbf{I} \mid \mathcal{I},[\tau_1,\tau_2] \models \varphi\}$$
$$\mathbf{J}_1{}^\frown\mathbf{J}_2 \mathrel{\widehat{=}} \{[\tau_1,\tau_2] \in \mathbf{I} \mid (\exists\tau \in \mathbf{R})([\tau_1,\tau] \in \mathbf{J}_1 \wedge [\tau,\tau_2] \in \mathbf{J}_2)\}$$
$$\mathbf{J}^0 \mathrel{\widehat{=}} \{[\tau,\tau] \mid \tau \in \mathbf{R}\}$$
$$\mathbf{J}^k \mathrel{\widehat{=}} \underbrace{\mathbf{J}^\frown\ldots^\frown\mathbf{J}}_{k \text{ times}} \qquad \text{for } k > 0$$
$$\mathbf{J}^* \mathrel{\widehat{=}} \bigcup_{k<\omega} \mathbf{J}^k$$

In words, $\tilde{\mathcal{I}}(\varphi)$ is the set of intervals that satisfy φ under $\mathcal{I}$, $\mathbf{J}_1{}^\frown\mathbf{J}_2$ is the set of intervals that are the concatenation of an interval in $\mathbf{J}_1$ and an interval in $\mathbf{J}_2$, and $\mathbf{J}^*$ is the iteration of $\mathbf{J}$ corresponding to the operation $^\frown$.

In the rest of the paper for the convenience of reading, we use the following conventions. We use the customary *infix* notation for terms with $+$, and formulas with $\leq$ and $=$ occurring in them. We introduce the constant $\top$, the boolean connectives $\wedge$, $\Rightarrow$ and $\Leftrightarrow$, the relation symbols $\neq$, $\geq$, $<$ and $>$, and the $\forall$ quantifier as abbreviations in the usual way. We assume that boolean connectives bind more tightly than $^\frown$. Since $^\frown$ is associative, we omit parentheses in formulas that contain consecutive occurrences of $^\frown$. Besides, we use the following abbreviations, that are generally accepted in Duration Calculus:

$$\begin{array}{ll}
\mathbf{1} \mathrel{\widehat{=}} \neg\mathbf{0} & \ell \mathrel{\widehat{=}} \int \mathbf{1} \\
\lceil S \rceil \mathrel{\widehat{=}} \int S = \ell \wedge \ell \neq 0 & \Diamond\varphi \mathrel{\widehat{=}} \top^\frown\varphi^\frown\top \\
\Box\varphi \mathrel{\widehat{=}} \neg\Diamond\neg\varphi & (\varphi^+) \mathrel{\widehat{=}} \varphi^\frown(\varphi^*) \\
\varphi^0 \mathrel{\widehat{=}} \ell = 0 & \varphi^k \mathrel{\widehat{=}} \underbrace{\varphi^\frown\ldots^\frown\varphi}_{k \text{ times}} \text{ for } k > 0
\end{array}$$

3 A Proof System for DC*

In this section, we propose a proof system for DC* which consists of a complete Hilbert-style proof system for first order logic (cf. e.g. [13]), axioms and rules for interval logic (cf. e.g. [7]), Duration Calculus axioms and rules ([9]) and axioms about iteration ([6]). We assume that the readers are familiar with Hilbert-style proof systems for first order logic and do not give one here. Here follow the interval logic and DC-specific axioms and rules.

Axioms and Rules for Interval Logic

$$\begin{array}{ll}
(A1_l) & (\varphi^\frown\psi) \wedge \neg(\chi^\frown\psi) \Rightarrow (\varphi \wedge \neg\chi^\frown\psi) \\
(A1_r) & (\varphi^\frown\psi) \wedge \neg(\varphi^\frown\chi) \Rightarrow (\varphi^\frown\psi \wedge \neg\chi) \\
(A2) & ((\varphi^\frown\psi)^\frown\chi) \Leftrightarrow (\varphi^\frown(\psi^\frown\chi)) \\
(R_l) & (\varphi^\frown\psi) \Rightarrow \varphi \text{ if } \varphi \text{ is rigid} \\
(R_r) & (\varphi^\frown\psi) \Rightarrow \psi \text{ if } \psi \text{ is rigid} \\
(B_l) & (\exists x\varphi^\frown\psi) \Rightarrow \exists x(\varphi^\frown\psi) \text{ if } x \text{ is not free in } \psi \\
(B_r) & (\varphi^\frown\exists x\psi) \Rightarrow \exists x(\varphi^\frown\psi) \text{ if } x \text{ is not free in } \varphi \\
(L1_l) & (\ell = x^\frown\varphi) \Rightarrow \neg(\ell = x^\frown\neg\varphi) \\
(L1_r) & (\varphi^\frown\ell = x) \Rightarrow \neg(\neg\varphi^\frown\ell = x) \\
(L2) & \ell = x + y \Leftrightarrow (\ell = x^\frown\ell = y) \\
(L3_l) & \varphi \Rightarrow (\ell = 0^\frown\varphi) \\
(L3_r) & \varphi \Rightarrow (\varphi^\frown\ell = 0)
\end{array}$$

$$(N_l)\ \frac{\varphi}{\neg(\neg\varphi^\frown\psi)} \qquad (N_r)\ \frac{\varphi}{\neg(\psi^\frown\neg\varphi)}$$

$$(Mono_l)\ \frac{\varphi \Rightarrow \psi}{(\varphi^\frown\chi) \Rightarrow (\psi^\frown\chi)} \qquad (Mono_r)\ \frac{\varphi \Rightarrow \psi}{(\chi^\frown\varphi) \Rightarrow (\chi^\frown\psi)}$$

Duration Calculus Axioms and Rules

$(DC1)$ $\int \mathbf{0} = 0$
$(DC2)$ $\int \mathbf{1} = \ell$
$(DC3)$ $\int S \geq 0$
$(DC4)$ $\int S_1 + \int S_2 = \int(S_1 \vee S_2) + \int(S_1 \wedge S_2)$
$(DC5)$ $(\int S = x^\frown \int S = y) \Rightarrow \int S = x + y$
$(DC6)$ $\int S_1 = \int S_2$ if $S_1 \Leftrightarrow S_2$ in propositional calculus.

$$(IR_1)\quad \frac{[\ell = 0/A]\varphi \quad \varphi \Rightarrow [A^\frown \lceil S \rceil /A]\varphi \quad \varphi \Rightarrow [A^\frown \lceil \neg S \rceil /A]\varphi}{[\top/A]\varphi}$$

$$(IR_2)\quad \frac{[\ell = 0/A]\varphi \quad \varphi \Rightarrow [\lceil S \rceil^\frown A/A]\varphi \quad \varphi \Rightarrow [\lceil \neg S \rceil^\frown A/A]\varphi}{[\top/A]\varphi}$$

$$(\omega)\quad \frac{\forall k < \omega \ [(\lceil S \rceil \vee \lceil \neg S \rceil)^k/A]\varphi}{[\top/A]\varphi}$$

Axioms about Iteration

(DC_1^*) $\ell = 0 \Rightarrow \varphi^*$
(DC_2^*) $(\varphi^{*\frown}\varphi) \Rightarrow \varphi^*$
(DC_3^*) $(\varphi^* \wedge \psi^\frown \top) \Rightarrow (\psi \wedge \ell = 0^\frown \top) \vee (((\varphi^* \wedge \neg\psi^\frown\varphi) \wedge \psi)^\frown \top)$.

The intuition behind DC_1^* and DC_2^* is quite straightforward. To see DC_3^*, assume that some initial subinterval of a given interval satisfies ψ, and can be chopped into finitely many parts, each satisfying φ. Then the smallest among the initial subintervals of the given one formed by these parts makes ψ hold exists which is either the 0-length initial subinterval, or otherwise consists one that does not satisfy ψ.

A restriction is made on the application of first order logic rules and axioms that involve substitution: $[t/x]\varphi$ is defined if no variable in t becomes bound due to the substitution, and either t is rigid or $^\frown$ does not occur in φ.

It is known that the above proof system for interval logic is complete with respect to an abstract class of time domains in place of $\mathbf{R}$ [7]. The proof system for interval logic, extended with the axioms DC_1-DC_6 and the rules IR_1, IR_2 is complete relative to the class of interval logic sentences that are valid on its real time frame [9]. Taking the infinitary rule ω instead of IR_1 and IR_2 yields an ω-complete system for DC with respect to an abstract class of time domains, like that of interval logic [8]. Adding appropriate axioms about reals, and a rule like, e.g.,

$$\frac{\forall k < \omega \ \overline{k}x \leq 1}{x \leq 0}$$

where $\overline{k}$ stands for $1 + \ldots + 1$ (k times), extends this system to one that is ω-complete with respect to the real time based semantics of DC given above.

In the rest of this section we show that adding DC_1^*-DC_3^* to the proof system of DC makes it complete for sentences where iteration is allowed only for a restricted class of formulas that we call *simple*. The following theorem gives the soundness of these axioms.

Theorem 3. *Let $\mathcal{I}$ be a Duration Calculus interpretation. Then $\mathcal{I}$ validates $DC_1^* - DC_3^*$.*

Proof. The proof about DC_1^* and DC_2^* is trivial and we omit it here. Now consider DC_3^*. Let $[\tau_1, \tau_2] \in \mathbf{I}$ be such that $\mathcal{I}, [\tau_1, \tau_2] \models \varphi^* \wedge \psi^\frown \top$, and $\mathcal{I}, [\tau_1, \tau_2] \models \neg(\psi \wedge \ell = 0^\frown \top)$. We shall prove that $\mathcal{I}, [\tau_1, \tau_2] \models ((\varphi^* \wedge \neg\psi^\frown\varphi) \wedge \psi)^\frown \top$. We have that $[\tau_1, \tau_1] \not\in \tilde{\mathcal{I}}(\psi)$, and $[\tau_1, \tau] \in \left(\tilde{\mathcal{I}}(\varphi)\right)^k \cap \tilde{\mathcal{I}}(\psi)$ for some $k < \omega$, and some $\tau \in [\tau_1, \tau_2]$. Then there exist $\tau_1', \ldots, \tau_{k+1}'$ such that $\tau_1 = \tau_1' < \ldots < \tau_{k+1}' = \tau$ and $\mathcal{I}, [\tau_i', \tau_{i+1}'] \models \varphi$ for $i = 1, \ldots, k$. Since $[\tau_1, \tau_{k+1}'] \models \psi$ and $[\tau_1, \tau_1'] \not\models \psi$ there must be $i \leq k$ for which $[\tau_1, \tau_i'] \not\models \psi$ and $[\tau_1, \tau_{i+1}'] \models \psi$. Therefore $\mathcal{I}, [\tau_1, \tau_{i+1}'] \models (\varphi^* \wedge \neg\psi^\frown\varphi) \wedge \psi$, which implies that $\mathcal{I}, [\tau_1, \tau_2] \models ((\varphi^* \wedge \neg\psi^\frown\varphi) \wedge \psi)^\frown \top$.

Let us prove the monotonicity of * from these axioms which says that if $\phi \Rightarrow \gamma$ then $\phi^* \Rightarrow \gamma^*$.

$$\begin{array}{lll}
\phi^* \wedge \neg\gamma^* & \Rightarrow (\neg\gamma^* \wedge \ell = 0^\frown\top) \vee (((\phi^* \wedge \gamma^{*\frown}\phi) \wedge \neg\gamma^*)^\frown\top) & \text{by } DC_3^* \\
& \Rightarrow (((\phi^* \wedge \gamma^{*\frown}\phi) \wedge \neg\gamma^*)^\frown\top) & \text{by } DC_1^* \\
& \Rightarrow (\neg\gamma^* \wedge \gamma^*)^\frown\top & \text{by } DC_2^* \\
& & \text{and } \phi \Rightarrow \gamma \\
& \Rightarrow \bot &
\end{array}$$

The following theorem is useful in practice. The readers are referred to [6] for a formal proof of its.

Theorem 4.

$$\vdash_{DC^*} \Box(\varphi \Rightarrow \neg(\top^\frown\neg\alpha) \wedge \neg(\neg\beta^\frown\top)) \wedge \Box(\ell = 0 \Rightarrow \alpha \wedge \beta) \Rightarrow \varphi^* \Rightarrow \Box(\alpha^\frown\varphi^{*\frown}\beta)\,.$$

Let us now use the proof system of DC* to prove the implication for the correctness of the simple Gas-Burner mentioned in the introduction of the paper. We have to prove that

$$((\lceil\!\lceil leak \rceil\!\rceil \wedge \ell \leq 1)^\frown(\lceil\!\lceil nonleak \rceil\!\rceil \wedge \ell \geq 30))^* \Rightarrow \Box(\ell \geq 60 \Rightarrow \textstyle\int leak \leq (1/20)\ell)\,.$$

Let us denote

$$\begin{array}{lcl}
\varphi & \widehat{=} & \lceil\!\lceil leak \rceil\!\rceil \wedge \ell \leq 1^\frown\lceil\!\lceil \neg leak \rceil\!\rceil \wedge \ell \geq 30\,, \\
\alpha & \widehat{=} & \ell = 0 \vee \lceil\!\lceil \neg leak \rceil\!\rceil \vee (\lceil\!\lceil leak \rceil\!\rceil \wedge \ell \leq 1^\frown\lceil\!\lceil \neg leak \rceil\!\rceil \wedge \ell \geq 30)\,, \\
\beta & \widehat{=} & \ell = 0 \vee (\ell \leq 1 \wedge \lceil\!\lceil leak \rceil\!\rceil^\frown\ell = 0 \vee \lceil\!\lceil \neg leak \rceil\!\rceil)\,.
\end{array}$$

From DC axioms it can be proved easily that $\vdash_{DC} \Box(\varphi \Rightarrow \neg(\top^\frown\neg\alpha) \wedge \neg(\neg\beta^\frown\top))$ and $\vdash_{DC} \Box(\ell = 0 \Rightarrow \alpha \wedge \beta)$. Therefore, from Theorem 4 we can complete the proof of the above if we can derive that $((\alpha^\frown\varphi^*)^\frown\beta) \Rightarrow 20\int leak \leq \ell$. This is done as follows.

1	$\alpha \Rightarrow 31 \int leak \leq \ell$	DC
2	$\varphi^* \wedge 31 \int leak > \ell \Rightarrow (\varphi^* \wedge 31 \int leak > \ell \frown \top)$	DC
3	$\varphi \Rightarrow 31 \int leak \leq \ell$	DC
4	$(\varphi^* \wedge 31 \int leak > \ell \frown \top) \Rightarrow$ $(\ell = 0 \wedge 31 \int leak > \ell \frown \top) \vee$ $(((\varphi^* \wedge 31 \int leak \leq \ell \frown \varphi) \wedge 31 \int leak > \ell) \frown \top)$	by DC_3^*
5	$\ell = 0 \Rightarrow 31 \int leak \leq \ell$	DC
6	$(\varphi^* \wedge 31 \int leak > \ell \frown \top) \Rightarrow$ $(((\varphi^* \wedge 31 \int leak \leq \ell \frown \varphi) \wedge 31 \int leak > \ell) \frown \top)$	by 4, 5, $Mono_r$
7	$(\varphi^* \wedge 31 \int leak \leq \ell \frown \varphi) \Rightarrow 31 \int leak \leq \ell$	by 2, 3, DC
8	$\varphi^* \Rightarrow 31 \int leak \leq \ell$	by 6, 7, $Mono_r$
9	$(\alpha \frown \varphi^*) \Rightarrow 31 \int leak \leq \ell$	by 1, 8, DC
10	$\beta \Rightarrow \int leak \leq 1$	DC
11	$((\alpha \frown \varphi^*) \frown \beta) \wedge \ell \geq 60 \Rightarrow 20 \int leak \leq \ell$	by 9, 10, DC, arithmetic

4 Completeness of DC* Proof System for Simple Formulas

As said in the introduction to the paper, our purpose is to give a rigorous study of a class of DC* formulas that play an important roles in practice. The formulas in the class are called simple formulas and will be considered to be executable. In this section we extend the class of simple formulas, originally introduced in [4] by allowing the conjunction in simple formulas. We give a proof of the completeness of the axiom system from the previous section for this class of formulas.

Definition 5. Simple *DC* formulas are defined by the following BNF:*

$$\varphi \ \widehat{=}\ \lceil S \rceil \mid a \leq \ell \mid \ell \leq a \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \frown \varphi) \mid \varphi^*$$

Before giving our main result on the completeness, we should first mention that we obtained our axioms $DC_1^* - DC_3^*$ from propositional dynamic logic [1]. We found that there is a certain degree of semantical compatibility between interval logic frames and propositional dynamic logic frames, and then built a truth-preserving translation of PDL formulas into interval logic ones based on this semantic correspondence. We applied this translation to obtain our axioms for iteration from the corresponding axioms in PDL. The readers are referred to our full report [6] for the details of the translation.

We are going to show that given simple formula φ and DC* formula γ, DC interpretations $\mathcal{I}$ that validate

$(DC_{1,\varphi,\gamma}^*)$ $\ell = 0 \Rightarrow \gamma$
$(DC_{2,\varphi,\gamma}^*)$ $(\gamma \frown \varphi) \Rightarrow \gamma$
$(DC_{3,\varphi,\gamma}^*)$ $(\gamma \wedge \psi \frown \top) \Rightarrow (\psi \wedge \ell = 0 \frown \top) \vee (((\gamma \wedge \neg\psi \frown \varphi) \wedge \psi) \frown \top)$

for all DC* formulas ψ should satisfy the equality $\left(\tilde{\mathcal{I}}(\varphi)\right)^* = \tilde{\mathcal{I}}(\gamma)$. This means that the axioms $DC_1^* - DC_3^*$ enforce the clause about iteration in the DC*

definition of $\models$ (Definition 2) for simple formulas φ. We do this in the following way: Given the assumption that $\left(\tilde{\mathcal{I}}(\varphi)\right)^* \neq \tilde{\mathcal{I}}(\gamma)$, we find an interval $[\tau_1, \tau_2]$ and a formula ψ that refute some of $DC^*_{1,\varphi,\gamma} - DC^*_{3,\varphi,\gamma}$ under $\mathcal{I}$. Having found an appropriate interval $[\tau_1, \tau_2]$, the formula ψ we need is a *-free one that satisfies $\tilde{\mathcal{I}}(\neg\psi) \cap \mathbf{I}([\tau_1, \tau_2]) = \left(\tilde{\mathcal{I}}(\varphi)\right)^* \cap \mathbf{I}([\tau_1, \tau_2])$.

4.1 Local Elimination of Iteration from Simple DC* Formulas

Elimination of iteration from *timed regular expressions*, that are closely related to DC* simple formulas, has been employed earlier under various other conditions as part of model-checking algorithms by Dang and Pham[5], and Li, Dang [11]. Lemma 7, Lemma 8, and Proposition 10 given below are a slightly stronger form of Lemma 3.6 from [11]. Because of the space limit, the proof of these lemmas which can be found in [6], is omitted here.

Iteration can be locally eliminated from a formula φ, if, for every DC interpretation $\mathcal{I}$ and every interval $[\tau_1, \tau_2] \in \mathbf{I}$, there exists a *-free formula φ' such that $\mathcal{I}, [\tau_1, \tau_2] \models \Box(\varphi \Leftrightarrow \varphi')$.

Due to the distributivity of conjunction and *chop* ($\frown$) over disjunction, simple formulas that have no occurrences of * are equivalent to disjunctions of *very simple* formulas, that are defined as follows:

Definition 6. Very simple *formulas are defined by the following BNF:*

$$\varphi \mathrel{\widehat{=}} \ell = 0 \mid \llbracket S \rrbracket \mid a \leq \ell \mid \ell \leq a \mid (\varphi \wedge \varphi) \mid (\varphi \frown \varphi)$$

Lemma 7. *Let $\mathcal{I}$ be a DC interpretation. Let $[\tau_1, \tau_2] \in \mathbf{I}$. Let φ be a disjunction of very simple formulas that contain no subformulas of the kind $a \leq \ell$ with $a \neq 0$. Then there exists a $k < \omega$ such that $\mathcal{I}, [\tau_1, \tau_2] \models \Box(\varphi^* \Leftrightarrow \bigvee\limits_{j=0}^{k} \varphi^j)$.*

Lemma 8. *Let $\mathcal{I}$ be a DC interpretation. Let $[\tau_1, \tau_2] \in \mathbf{I}$. Let φ be a disjunction of very simple formulas. Then there exists a *-free simple formula φ' such that $\mathcal{I}, [\tau_1, \tau_2] \models \Box(\varphi^* \Leftrightarrow \varphi')$.*

Lemma 9. *Let φ be a *-free simple formula. Then there exists formula φ' which is a disjunction of very simple formulas, such that $\vdash_{DC} \varphi \Leftrightarrow \varphi'$.*

Proposition 10. *Let $\mathcal{I}$ be a DC interpretation. Let $[\tau_1, \tau_2] \in \mathbf{I}$. Then for every simple formula φ there exists a *-free simple formula φ' such that $\mathcal{I}, [\tau_1, \tau_2] \models \Box(\varphi \Leftrightarrow \varphi')$.*

Proof. Proof is by induction on the number of occurrences of * in φ. Let ψ^* be a subformula of φ and let ψ be *-free. By Lemma 9, $\vdash_{DC} \psi \Leftrightarrow \psi'$ for some disjunction of very simple formulas ψ'. Now $\mathcal{I}, [\tau_1, \tau_2] \models \Box((\psi')^* \Leftrightarrow \psi'')$ for some *-free simple formula ψ'' by Lemma 8. Hence $\mathcal{I}, [\tau_1, \tau_2] \models \Box(\varphi \Leftrightarrow \varphi')$, where φ' is obtained by replacing the occurrence of ψ^* in φ by ψ''. Thus the number of the occurrences of * in φ is reduced by at least one.

4.2 Completeness of DC_1^*-DC_3^* for Simple DC* Formulas

In this section, we prove that a formula γ is the iteration of a simple formula φ if and only if it satisfies the axioms DC$_1^*$, DC$_2^*$ and DC$_3^*$ for all DC* formulas. The following proposition has a key role in our proof.

Proposition 11. *Let $\mathcal{I}$ be a DC model that validates $DC^*_{1,\varphi,\gamma}$, $DC^*_{2,\varphi,\gamma}$ and $DC^*_{3,\varphi,\gamma}$ for some simple DC* formula φ, some arbitrary DC* formula γ, and all DC* formulas ψ. Then $\tilde{\mathcal{I}}(\gamma) = \left(\tilde{\mathcal{I}}(\varphi)\right)^*$.*

Proof. The validity of DC_1^* and DC_2^* entails that $\tilde{\mathcal{I}}(\gamma) \supseteq \left(\tilde{\mathcal{I}}(\varphi)\right)^*$. The proof of this is trivial and we omit it. For the sake of contradiction, assume that $[\tau_1, \tau_2] \in \tilde{\mathcal{I}}(\gamma) \setminus \left(\tilde{I}(\varphi)\right)^*$. By Proposition 10 there exists a simple formula φ' such that $\tilde{\mathcal{I}}(\varphi') \cap \mathbf{I}([\tau_1, \tau_2]) = \left(\tilde{\mathcal{I}}(\varphi)\right)^* \cap \mathbf{I}([\tau_1, \tau_2])$. Let $\psi \mathrel{\widehat{=}} \neg\varphi'$. Since $\mathcal{I}, [\tau_1, \tau_2] \models \gamma$ and $[\tau_1, \tau_2] \not\in \left(\tilde{\mathcal{I}}(\varphi)\right)^*$, we have $\mathcal{I}, [\tau_1, \tau_2] \models \gamma \wedge \psi$, and hence $\mathcal{I}, [\tau_1, \tau_2] \models \gamma \wedge \psi^\frown\top$. Since $[\tau_1, \tau_1] \in \left(\tilde{\mathcal{I}}(\varphi)\right)^*$, $\mathcal{I}, [\tau_1, \tau_2] \not\models \psi \wedge \ell = 0^\frown\top$. Now assume that $\mathcal{I}, [\tau_1, \tau_2] \models (\gamma \wedge \neg\psi^\frown\varphi) \wedge \psi^\frown\top$. This entails that for some $\tau', \tau'' \in [\tau_1, \tau_2]$ $\mathcal{I}, [\tau_1, \tau'] \models \neg\psi$, and $\mathcal{I}, [\tau', \tau''] \models \varphi$. Then for some $k < \omega$ there exist $\tau_1', \ldots, \tau_{k+1}'$ such that $\tau_1 = \tau_1' < \ldots < \tau_{k+1}' = \tau''$ and $\mathcal{I}, [\tau_i', \tau_{i+1}'] \models \varphi$ for $i = 1, \ldots, k$, and besides, $\mathcal{I}, [\tau_1', \tau_{k+1}'] \models \psi$. This implies that $[\tau_1, \tau_{k+1}'] \in \left(\tilde{\mathcal{I}}(\varphi)\right)^k$ and $[\tau_1, \tau_{k+1}'] \in \tilde{\mathcal{I}}(\psi) \subseteq \mathcal{I} \setminus \left(\tilde{\mathcal{I}}(\varphi)\right)^*$, which is a contradiction.

Now let us state the completeness theorem for DC* with iteration of simple formulas.

Theorem 12. *Let φ be a DC* formula. Let that all of its *-subformulas be simple. Then either φ is satisfiable by some DC interpretation, or $\neg\varphi$ is derivable in our proof system.*

Proof. Assume that $\neg\varphi$ is not derivable. Let Γ be the set of all the instances of DC_1^*-DC_3^*. Then $\Gamma \cup \{\varphi\}$ is consistent, and by considering occurrences of a formula of the form ψ^* as a temporal variable, we have that $\Gamma \cup \{\varphi\}$ is a consistent set of DC formulas with temporal variables. By the ω-completeness of DC, there exists an interpretation I, and an interval $[\tau_1, \tau_2]$ such that $I, [\tau_1, \tau_2] \models \Gamma, \varphi$. Now Proposition 11 entails that $\tilde{I}(\psi^*) = \left(\tilde{I}(\psi)\right)^*$ for all ψ such that ψ^* occurs in φ, whence the modelling relation $I \models \varphi$ is as required for a DC* interpretation.

5 Decidability Results for Simple DC* and Discussion

In this section, we will discuss about the decidability of the satisfiability of simple DC* formulas and the related work.

One of the notions in the literatures that are closed to our notion of simple DC* is the notion of *Timed Regular Expressions* introduced by Asarin et al in [3], a subset of which has been introduced by us earlier in [11]. Each simple DC* formula syntactically corresponds exactly to a timed regular expression, and their semantics coincide. Therefore, a simple DC* formula can be viewed as a timed regular expression. In [3], it has been proved that from a timed regular expression E one can build a timed automaton A to recognise exactly the models of E in which the constants occurring in the constraints for the clock variables (guards, tests and invariants) are from the expression E (see [3]). It is well known ([2]) that the emptiness of the timed automata is decidable for the case that the constants occurring in the guards and tests are integers [2], we can conclude that if only integer constants are allowed in the inequalities in the definition of simple DC* formulas, then the satisfiability of a simple DC* formulas is decidable.

Theorem 13. *Given a simple DC* formula φ in which all the constants occurring in the inequalities are integers. The satisfiability of φ is decidable.*

The complexity of the decidability procedure, however, is exponential in the size of the constants occurring in the clock constraints (see, e.g. [2]). Note that the decidability of DC* could be derived from the results in [12] also.

In [3] it is also shown that from a timed automaton, one can build a timed regular expression and a renaming of the automaton states such that each model of the timed regular expression is the renaming of a behaviour of the automaton. In this sense, we can say that the expressive power of the simple DC* formulas is the same as the expressive power of the timed automata.

If we restrict ourselves to the class of *sequential* simple DC* formulas then we can have a very simple decidability procedure for the satisfiability, and some interesting results. The sequential simple DC* formulas are defined by the following BNF:

$$\varphi \mathrel{\widehat{=}} \ell = 0 \mid \llbracket S \rrbracket \mid \varphi \vee \varphi \mid (\varphi ^\frown \varphi) \mid \varphi^* \mid \varphi \wedge a \leq \ell \mid \varphi \wedge \ell \leq a$$

Because the operators $^\frown$ and $\wedge$ are distributed over $\vee$, and because of the equivalence $(\varphi \vee \psi)^* \Leftrightarrow (\varphi^{*\frown}\psi^*)^*$, each sequential simple DC* formula φ is equivalent to a disjunction of simple formulas having no occurrences of $\vee$. Therefore φ is satisfiable iff at least one of the components of the disjunction is satisfiable. The satisfiability of sequential simple DC* formulas having no occurrence of $\vee$ is easy to decide.

In [11], we have developed some simple algorithms for checking a real-time system whose behaviour is described by a 'sequential' timed regular expression for a linear duration invariants of the form $\Box(a \leq \ell \leq b \Rightarrow \sum_{s \in S} c_s \int s \leq M)$. Because of the obvious correspondence between sequential simple DC* formulas and sequential timed regular expressions, these algorithms can be used for proving automatically the implication from a sequential simple DC* formula to a linear duration invariant. An advantage of the method is that it reduces the problem to several number of linear programming problems, which have been well understood. Because of this advantage, in [5], we tried to generalise the method for the general simple DC* formulas, and showed that in most cases, the method can still be used for checking the implication from a simple DC* formula to a linear duration invariant.

Together with the proof system presented in the previous sections, these decidability procedures will help to develop a tool to assist the designing and verification of real-time systems.

It seems that the with the extension of DC with the operator $*$, we can only capture the “regular” behaviour of real-time systems. In order to capture their full behaviour, we have to use the extension of DC with recursions ([12]). However, we believe that in this case the proof system would be more complicated, and would be far from to be complete.

References

1. Abramsky, S., Gabbay, D., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science. Clarendon Press, Oxford (1992).
2. Alur, R., Dill, D. L.: A Theory of Timed Automata. Theoretical Computer Science. **126** (1994) 183-235.
3. Asarin, E., Caspi, P., Maler, O.: A Kleene Theorem for Timed Automata. In: Winskel, G. (Ed.): Proceedings of IEEE International Symposium on Logics in Computer Science LICS’97. IEEE Computer Society Press (1997) 160-171.
4. Dang Van, H., Wang Ji: On The Design of Hybrid Control Systems Using Automata Models. In Chandru, V, Vinay, V. (eds.): Theoretical Computer Science. LNCS 1180. Springer-Verlag (1996) 415–438.
5. Dang Van, H., Pham, H.T.: On Checking Parallel Real-Time Systems for Linear Duration Invariants. In Kramer, B., Uchihita, N., Croll, P., Russo, S. (eds.): Proceedings of PDSE’98. IEEE Computer Society Press (1998) 61-71.
6. Dang Van, H., Guelev, Dimitar P.: Completeness and Decidability of a Fragment of Duration Calculus with Iterations. Technical Report 163, UNU/IIST, P.O.Box 3058, Macau, 1998.
7. Dutertre, B.: On First Order Interval Temporal Logic Report No. CSD-TR-94-3. Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, England, 1995.
8. Guelev, D.P.: A Calculus of Durations on Abstract Domains: Completeness and Extensions. Technical Report 139, UNU/IIST, P.O.Box 3058, Macau, 1998.
9. Hansen, M.R., Zhou, C.C.: Semantics and Completeness of Duration Calculus. In: Real-Time: Theory and Practice. LNCS 600. Springer-Verlag (1992) 209-225.
10. Hansen, M.R., Zhou, C.C.: Duration Calculus: Logical Foundations. Formal Aspects of Computing **9** (1997) 283-330.
11. Li, X.D., Dang Van, H.: Checking Linear Duration invariants by Linear Programming. In: Jaffar, J., Yap, R.H.C. (eds.): Concurrency and Paralellism, Programming, Networking, and Security. LNCS 1179. Springer-Verlag (1996) 321-332.
12. Pandya, P.K.: Some Extensions to Mean-Value Calculus: Expressiveness and Decidability. In: Buning, H.K.(ed), Proc. CSL’95. LNCS 1092. Springer-Verlag (1995).
13. Shoenfield, J.: Mathematical Logic. Addison-Wesley, Massachusetts (1967).
14. Zhou, C.C., Hoare, C. A. R., Ravn, A. P.: A Calculus of Durations. Information Processing Letters, **40(5)** (1991) 269-276.
15. Zhou, C.C., Zhang, J.Z., Yang, L., and Li, X.S.: Linear Duration Invariants. In: Formal Techniques in Real-Time and Fault-Tolerant systems. LNCS 863. (1994).

Logical Relations in Circuit Verification

Mia Indrika

Department of Computing Science
Chalmers University of Technology and Göteborg University
Göteborg, Sweden
indrika@cs.chalmers.se

Abstract. We present a verification methodology for combinational arithmetic circuits which allows us to reason about circuits at a high level of abstraction and to have better-structured and compositional proofs. This is obtained using a categorical characterisation of the notion of data refinement. Within this categorical framework we introduce a notion of logical relation to deal with a language for hardware description.

1 Introduction

This paper presents a methodology for circuit verification, particularly for combinational arithmetic circuits. Our concern here is how to reason in a high-level language about low-level objects. Concrete arithmetic circuits are low-level objects typically described with operations on bits. But they are meant to implement arithmetic operations such as addition, multiplication, etc. that are meaningful at a higher level such as at the level of natural numbers.

We view these two levels of abstraction as a step of data refinement. The low level is seen as the level of the implementation (level of the actual circuits) and the high level as the level of the specification. The methodology that we present is based on data refinement originated in [7] and its categorical approach (e.g. [8], [10]). It uses logical relations as abstraction relations between the two levels. The use of logical relations allows us to deal with higher-order structure of circuits (*circuit combinators*), that is, an operation that constructs a circuit from other circuits. Defining circuits by means of circuit combinators will allow better-structured and compositional proofs.

2 Circuits, Specification, and Verification

We begin by introducing a signature of a hardware description language. The language is a simply-typed λ-calculus with sorts

$$T \ ::= \ \mathsf{B}_n \mid \mathbf{1} \mid T \to T \mid T \times T$$

where B_n ($n \geq 0$) is a type of n-bit wire. Its operation symbols are primitive circuits, for example, **not** : $\mathsf{B}_1 \longrightarrow \mathsf{B}_1$, **and**, **or**, **xor** : $\mathsf{B}_1 \times \mathsf{B}_1 \longrightarrow \mathsf{B}_1$, and converters between n 1-bit wires and one n-bit wire $\lhd_n : \mathsf{B}_n \longrightarrow \underbrace{\mathsf{B}_1 \times \mathsf{B}_1 \times \ldots \times \mathsf{B}_1}_{n}$,

$\triangleright_n : \underbrace{\mathsf{B}_1 \times \mathsf{B}_1 \times ... \times \mathsf{B}_1}_{n} \longrightarrow \mathsf{B}_n$. We may further include primitive circuit combinators, for example, **map** : $(T_1 \to T_2) \times {T_1}^n \longrightarrow {T_2}^n$ using higher-order sorts. So, circuit descriptions are programs in the language. Further, we use Haskell-like notation whose meaning should be clear, to write circuit descriptions.

Example 1. A half-adder circuit **hadd** is described as

$$\begin{aligned} \mathbf{hadd} &: \mathsf{B}_1 \times \mathsf{B}_1 \longrightarrow \mathsf{B}_1 \times \mathsf{B}_1 \\ \mathbf{hadd} &= \lambda x.\ (\mathbf{and}\ x,\ \mathbf{xor}\ x) \end{aligned}$$

A 1-bit full-adder **fadd** is made out of two half-adders **hadd** and an **or**

$$\begin{aligned} \mathbf{fadd} &: \mathsf{B}_1 \times \mathsf{B}_1 \times \mathsf{B}_1 \longrightarrow \mathsf{B}_1 \times \mathsf{B}_1 \\ \mathbf{fadd} &= \lambda(a,b,c).\ \mathtt{let}\ (u,\ v)\ =\ \mathbf{hadd}\ (a,\ b) \\ &\qquad\qquad\qquad\quad (w,\ x)\ =\ \mathbf{hadd}\ (v,\ c) \\ &\qquad\qquad\quad \mathtt{in}\ (\mathbf{or}\ (u,\ w),\ x) \end{aligned}$$

Abstracting out **hadd**, we obtain the circuit combinator

$$\begin{aligned} \varphi_{\mathbf{fadd}} &: (\mathsf{B}_1 \times \mathsf{B}_1 \to \mathsf{B}_1 \times \mathsf{B}_1)\ \times\ (\mathsf{B}_1 \times \mathsf{B}_1 \times \mathsf{B}_1) \longrightarrow \mathsf{B}_1 \times \mathsf{B}_1 \\ \varphi_{\mathbf{fadd}} &= \lambda h.\ \lambda(a,\ b,\ c).\ \mathtt{let}\ (u,\ v)\ =\ h\ (a,\ b) \\ &\qquad\qquad\qquad\qquad\quad (w,\ x)\ =\ h\ (v,\ c) \\ &\qquad\qquad\qquad\quad \mathtt{in}\ \ (\mathbf{or}\ (u,\ w),\ x) \end{aligned}$$

which represents the connections to build **fadd** from the components **hadd**,

$$\mathbf{fadd}\ =\ \varphi_{\mathbf{fadd}}(\mathbf{hadd})$$

Actual circuits which work at the bit level are meant to implement arithmetic functions which are meaningful at a more abstract level of, e.g. natural numbers. For example, an *adder* circuit implements the addition function. The specification of a circuit is at the same level of abstraction as the arithmetic function rather than at the bit level. This idea raises the question of how to develop a verification methodology that allows implementation at a low level and reasoning at a higher level of abstraction.

The verification process begins with interpreting circuit descriptions in two levels of abstraction: concrete level (bit level) and abstract level (natural numbers level). A concrete-level circuit (*concrete circuit*) represents the actual circuit (the implementation). An abstract-level circuit (*abstract circuit*) has exactly the same topology as concrete circuit but uses arithmetic functions in place of bit-level operations. So, abstract circuits can be verified against the specifications at the same high level. Our proposed methodology will make sure that proving the correctness of the abstract circuit guarantees the correctness of the concrete one.

3 Preliminaries

We assume familiarity with the standard notions of category, functor, Cartesian closed category (CCC) and its correspondence with simply-typed λ-calculus [12], product of two categories, and the category **Set** of sets and total functions, which can be found for example in [1, 13].

In this section, we give an account of the background that is necessary for our verification methodology like the notion of sketch, the category **Rel** of binary relations and the functor $\langle dom, cod \rangle$ from **Rel** to **Set** $\times$ **Set**.

3.1 Sketches

In developing our methodology, we regard the terms of hardware description language to be denoting arrows of a certain CCC **HW**. The purpose is to use the characterization of **HW** to obtain an appropriate (logical) relation connecting the concrete and abstract levels.

We can represent a signature of the hardware description language as a directed graph whose nodes are primitive sorts and whose arrows are primitive operations on values of the sorts. An operation with several arguments can be represented by an arrow from a node representing the Cartesian product of all the sorts of the arguments. Similarly, a higher-order operation can be represented by an arrow from a node representing a function space built from other sorts. In this case, we need more than just a graph, we need some equations stating that a certain node must be constructed from other sorts.

A sketch is a directed graph with some equations; it is a way of expressing a structure. We refer the reader to [1] for an introduction to the concept of sketch. There are many kinds of sketch; the one used in this paper is Cartesian closed (CC) sketch, which is an instance of the general sketch presented in [11]

Definition 1. *A* CC sketch $S = \langle G,\ E \rangle$ *consists of a graph G and a set E of equations of the form $c = \alpha(c_1, \cdots, c_n)$ where c, c_i are nodes or edges of G and α is an expression built from operations of CCC structure (compositions $\circ$, identity arrows* $\mathrm{id}_{\_}$, *products $\times$, pairing $\langle -, - \rangle$, projections $\pi_{\_\_}$, $\pi'_{\_\_}$, the terminal object* $\mathbf{1}$, *the unique arrows to* $\mathbf{1}$ $!_{\_}$, *exponentials $\rightarrow$, currying $(-)^*$, evaluation maps* $\mathrm{ev}_{\_\_}$*). A* model M *of S in a Cartesian closed category $\mathcal{C}$ is a graph homomorphism from G to the underlying graph of $\mathcal{C}$ such that $Mc = \alpha(Mc_1, \cdots, Mc_n)$ holds in $\mathcal{C}$.*

Hence, a model of the sketch $S = \langle G,\ E \rangle$ in the category **Set** interprets sorts and operations as sets and functions, respectively. Further, the interpretation satisfies the equations stated in E. For instance, when $X = A \times B$ is given in E, the interpretation of X is the product of the interpretations of A and B.

For any CC sketch $S = \langle G,\ E \rangle$ there is the CC category $\mathcal{F}_{CC}(S)$ which is *freely generated by S subject to the equations in E* [11]. The CC category $\mathcal{F}_{CC}(S)$ comes with a model $\iota : S \longrightarrow \mathcal{F}_{CC}(S)$ and has the following characterization: for every Cartesian closed category $\mathcal{C}$ and every model $F_0 : S \longrightarrow \mathcal{C}$ there is a unique strict CC functor $F : \mathcal{F}_{CC}(S) \longrightarrow \mathcal{C}$ for which $F \circ \iota = F_0$.

Intuitively this property tells that the objects and the arrows of $\mathcal{F}_{CC}(S)$ are inductively generated by Cartesian closed operations from the nodes and the edges of G, respectively, subject to the equations in E and the axioms of CCC.

The model ι is then given by inclusion of S in $\mathcal{F}_{CC}(S)$ modulo the equations. It also tells that F is given by F_0 for primitive sorts and operation symbols, and by structural recursion for constructed sorts and terms.

3.2 Category of Relations

We regard a binary relation R as a triple (A, A', R) where A and A' are sets and $R \subseteq A \times A'$. We use the notation $A \overset{R}{\leftarrowtail\!\rightharpoonup} A'$ to mean R is a binary relation over A and A', and $a\ R\ a'$ to mean $(a, a') \in R$.

Definition 2. *The* category **Rel** *has as objects binary relations* (A, A', R) *and an arrow from* (A, A', R) *to* (B, B', S) *is a pair of functions* $(f : A \to B,\ f' : A' \to B')$ *such that* $\forall a \in A.\ \forall a' \in A'.$ *if* $a\ R\ a'$ *then* $fa\ S\ f'a'$*. We use the following diagram for describing that condition.*

$$\begin{tikzcd}
A \arrow[r, tail, harpoon, "R"] \arrow[d, "f"'] & A' \arrow[d, "f'"] \\
B \arrow[r, tail, harpoon, "S"'] & B'
\end{tikzcd}$$

Composition and identities are given component-wise.

It is routine to verify that the category **Rel** is Cartesian closed with

- the terminal object is a triple whose components are singletons: $\mathbf{1}_{Rel} = (\mathbf{1}, \mathbf{1}, \mathbf{1} \times \mathbf{1})$ where $\mathbf{1}$ is the singleton $\{*\}$,
- product is defined component-wise: $(A, A', R) \times (B, B', S) = (A \times B, A' \times B', R \times S)$ where $(a, b)\ R \times S\ (a', b')$ if and only if $a\ R\ a'$ and $b\ S\ b'$, and
- exponentiation: $[(A, A', R) \to (B, B', S)] = ([A \to B], [A' \to B'], [R \to S])$ where $f\ [R \to S]\ f'$ if and only if $\forall a \in A.\ \forall a' \in A'.\quad a\ R\ a' \Rightarrow fa\ S\ f'a'$

We denote a forgetful functor $\langle dom, cod\rangle$ from **Rel** to **Set** $\times$ **Set** sending (A, A', R) to (A, A') and (f, g) to (f, g). Note that **Set** $\times$ **Set** is a CCC with the structure given component-wise. It is straightforward to verify that $\langle dom, cod\rangle$ is a strict Cartesian closed functor.

4 Circuit Descriptions

Both the abstract-level and the concrete-level versions of a circuit have the same topology given by the circuit description. We now introduce the category **HW**whose objects are sorts of the language and whose arrows are circuit description (or simply *circuits*). We start with the CC sketch H of primitive circuits, then we define the category **HW** to be the free CCC on it.

The carrier graph H_0 of H has as nodes B_n ($n \geq 0$) of n-bit wire types, together with $\mathsf{N}_{B_1 \times B_1}$, $\mathsf{N}_{[B_1 \to B_1]}$, etc. which represent product $\mathsf{B}_1 \times \mathsf{B}_1$ and exponential $[\mathsf{B}_1 \to \mathsf{B}_1]$, respectively. The edges of H_0 are the names of primitive circuits, including among others are **not**, **and**, **or**, and **xor**. The primitive circuit **not** is an edge from B_1 to B_1, whereas primitive circuits **and**, **or**, and **xor** are edges from $\mathsf{N}_{B_1 \times B_1}$ to B_1. We also consider the conversions between n 1-bit wires and one n-bit wire:

– $\lhd_n : \mathsf{B}_n \longrightarrow \mathsf{N}_{\underbrace{(((\mathsf{B}_1 \times \mathsf{B}_1) \times ...) \times \mathsf{B}_1)}_{n}}$

– $\rhd_n : \mathsf{N}_{\underbrace{(((\mathsf{B}_1 \times \mathsf{B}_1) \times ...) \times \mathsf{B}_1)}_{n}} \longrightarrow \mathsf{B}_n$

The equations of the sketch H simply specify $\mathsf{N}_{\mathsf{B}_1 \times \mathsf{B}_1} = \mathsf{B}_1 \times \mathsf{B}_1$, $\mathsf{N}_{[\mathsf{B}_1 \to \mathsf{B}_1]} = [\mathsf{B}_1 \to \mathsf{B}_1]$, etc.

For the rest of the paper, we use convention that the product associates to the left, so $\mathsf{B}_1 \times \mathsf{B}_1 \times \mathsf{B}_1$ means $(\mathsf{B}_1 \times \mathsf{B}_1) \times \mathsf{B}_1$.

The reason we consider both B_n and $\mathsf{N}_{\underbrace{\mathsf{B}_1 \times \mathsf{B}_1 \times ... \times \mathsf{B}_1}_{n}}$ is that the abstract interpretation of B_n is not necessarily isomorphic to the n-fold product of that of B_1. Thus, we do not demand the equations $\lhd_n \circ \rhd_n = id_{\mathsf{B}_n}$ and $\rhd_n \circ \lhd_n = id_{\mathsf{N}_{\underbrace{\mathsf{B}_1 \times ... \times \mathsf{B}_1}_{n}}}$.

Definition 3. *A* category **HW** *is the free Cartesian closed category $\mathcal{F}_{CC}(H)$ on the sketch H.*

So, **HW** has a terminal object **1** and for each pair of objects A and B, a product $A \times B$ and an exponential object $[A \to B]$. The products allow us to express multiple input/output and the exponentials enable us to have circuit combinators which can be used, for instance, to represent the connections of a circuit. The arrows in **HW** are given by canonical arrows of CCC (id_-, $!_-$, π_{--}, π'_{--}, ev_{--}), compositions ($f \circ g$), pairing ($\langle -, - \rangle$), currying ($(-)^*$), and the primitive circuits in H. We omit the canonical isomorphisms of CCC, e.g. $(A \times B) \times C \;\cong\; A \times (B \times C)$.

With the well-known correspondence between CCC and λ-calculus [12], we identify arrows of **HW** and circuit description.

Example 2. A 2-bit full-adder **fadd2** is made out of two full-adders **fadd**,

$$\begin{aligned}\mathbf{fadd2} &: \mathsf{B}_2 \times \mathsf{B}_2 \times \mathsf{B}_1 \longrightarrow \mathsf{B}_2 \times \mathsf{B}_1 \\ \mathbf{fadd2} &= \mathbf{apply}_{(\varphi_{\mathbf{fadd2}},\ \mathbf{fadd})}\end{aligned}$$

where

$$\begin{aligned}
\varphi_{\mathbf{fadd2}} &: [\mathsf{B}_1 \times \mathsf{B}_1 \times \mathsf{B}_1 \to \mathsf{B}_1 \times \mathsf{B}_1] \times (\mathsf{B}_2 \times \mathsf{B}_2 \times \mathsf{B}_1) \longrightarrow \mathsf{B}_2 \times \mathsf{B}_1 \\
\varphi_{\mathbf{fadd2}} &= \lambda f.\ \lambda(a,b,c).\ \texttt{let}\ (a_1,\ a_0)\ =\ \rhd_2\ a \\
&\qquad\qquad\qquad\quad (b_1,\ b_0)\ =\ \rhd_2\ b \\
&\qquad\qquad\qquad\quad (u,\ v)\ =\ f\ (a_0,\ b_0,\ c) \\
&\qquad\qquad\qquad\quad (w,\ x)\ =\ f\ (a_1, b_1,\ u) \\
&\qquad\qquad\qquad\ \texttt{in}\ (w,\ \lhd_2\ (x,\ v))
\end{aligned}$$

and $\mathbf{apply}_{(\varphi, c_1, \ldots, c_n)}$ is an abbreviation for application of φ that involves conversions of $C_i : A \longrightarrow B$ to global elements $(C_i \circ \pi'_{1A})^* : \mathbf{1} \longrightarrow [A \to B]$.

Without having higher-order structure (circuit combinators), proving the correctness of **fadd2** involves proving the correctness of **fadd** for specific arguments (**fadd** $(a_0,\ b_0,\ c)$ and **fadd** $(a_1,\ a_2,\ v)$). This is not the case when we have circuit combinators. Instead, we prove the correctness of **fadd** and then instantiate the proof for specific arguments.

5 Circuit Interpretations

Definition 4. *An interpretation of the circuit descriptions is a Cartesian closed functor from* **HW** *to* **Set**.

So, the interpretation of a circuit description (an arrow in **HW**) is a function. By the freeness of **HW**, an interpretation F is uniquely determined by a model of F_0 of the CC sketch H in **Set**.

If we interpret a circuit description at the level of bits, we get the corresponding *concrete circuit* representing an actual circuit. Similarly, by interpreting a circuit description at the level of the specification of the circuit, for instance natural numbers level, we get the corresponding *abstract circuit.*

We define the concrete model C_0 of H in **Set** by $C_0(\mathsf{B}_1) = \texttt{Bit}$ $(= \{1,\ 0\})$ and $C_0(\mathsf{B}_n) = \texttt{Bit}^n$ for nodes, and for edges:

$$\begin{aligned}
\mathbf{not}_C &: \texttt{Bit} \longrightarrow \texttt{Bit} \\
\mathbf{not}_C &= \lambda x.\ \texttt{if}\ (x\ ==\ 1)\ \texttt{then}\ 0\ \texttt{else}\ 1
\end{aligned}$$

$$\begin{aligned}
\mathbf{and}_C,\ \mathbf{or}_C,\ \mathbf{xor}_C &: \texttt{Bit} \times \texttt{Bit} \longrightarrow \texttt{Bit} \\
\mathbf{and}_C &= \lambda(x,\ y).\ \texttt{if}\ (x\ ==\ 1\ \wedge\ y\ ==\ 1)\ \texttt{then}\ 1\ \texttt{else}\ 0 \\
\mathbf{or}_C &= \lambda(x,\ y).\ \texttt{if}\ (x\ ==\ 1\ \vee\ y\ ==\ 1)\ \texttt{then}\ 1\ \texttt{else}\ 0 \\
\mathbf{xor}_C &= \lambda(x,\ y).\ \texttt{if}\ (x\ \neq\ y)\ \texttt{then}\ 1\ \texttt{else}\ 0
\end{aligned}$$

$$\begin{aligned}
\lhd_{nC},\ \rhd_n &: \texttt{Bit}^n \longrightarrow \texttt{Bit}^n \\
\lhd_{nC} &= \texttt{id} \\
\rhd_{nC} &= \texttt{id}
\end{aligned}$$

The concrete interpretation C is the unique strict CC functor with $C \circ \iota = C_0$. In other words, C is defined by structural recursion with base cases given by C_0. So, the concrete interpretation C of the primitive circuits are given by the concrete model C_0. The concrete interpretation of 1-bit full-adder **fadd** after unfolding **hadd** is

$$\begin{aligned}
C(\mathbf{fadd}) &: \texttt{Bit} \times \texttt{Bit} \times \texttt{Bit} \longrightarrow \texttt{Bit} \times \texttt{Bit} \\
C(\mathbf{fadd}) &= \lambda(a,\ b,\ c).\ \texttt{let}\ (u,\ v) = (\mathbf{and}_C\ (a,\ b),\ \mathbf{xor}_C\ (a,\ b)) \\
&\qquad\qquad\qquad\ \ (w,\ x) = (\mathbf{and}_C\ (v,\ c),\ \mathbf{xor}_C\ (v,\ c)) \\
&\qquad\qquad \texttt{in}\ \ (\mathbf{or}_C\ (u,\ w),\ x)
\end{aligned}$$

Similarly, we define the abstract model A_0 of H in **Set** by $A(\texttt{Bit}_n) = \texttt{Nat}$ (the set of natural numbers) for nodes, and for edges:

$$\begin{aligned}
\mathbf{not}_A &: \texttt{Nat} \longrightarrow \texttt{Nat} \\
\mathbf{not}_A &= \lambda x.\ (\texttt{succ}\ x)\ \ \texttt{mod}\ 2
\end{aligned}$$

$$\begin{aligned}
\mathbf{and}_A,\ \mathbf{or}_A,\ \mathbf{xor}_A &: \texttt{Nat} \times \texttt{Nat} \longrightarrow \texttt{Nat} \\
\mathbf{and}_A &= \lambda(x,\ y).\ (x\ +\ y)\ \ \texttt{div}\ 2 \\
\mathbf{or}_A &= \lambda(x,\ y).\ (\texttt{succ}\ (x\ +\ y))\ \ \texttt{div}\ 2 \\
\mathbf{xor}_A &= \lambda(x,\ y).\ (x\ +\ y)\ \ \texttt{mod}\ 2.
\end{aligned}$$

$$\begin{aligned}
\lhd_{nA} &: \texttt{Nat} \longrightarrow \texttt{Nat} \times \texttt{Nat} \times \ldots \times \texttt{Nat} \\
\lhd_{nA} &= \lambda x.\ ((x\ \texttt{div}\ 2^{n-1})\ \texttt{mod}\ 2,\ \ldots,\ (x\ \texttt{div}\ 2^0)\ \texttt{mod}\ 2)
\end{aligned}$$

$$\begin{aligned}
\rhd_{nA} &: \texttt{Nat} \times \texttt{Nat} \times \ldots \times \texttt{Nat} \longrightarrow \texttt{Nat} \\
\rhd_{nA} &= \lambda(x_1,\ \ldots,\ x_n).\ \ x_1 \times 2^{n-1}\ +\ \ldots\ +\ x_n \times 2^0
\end{aligned}$$

Similarly, the abstract interpretation A is the unique strict CC functor with $A \circ \iota = A_0$. So, the abstract interpretation A of the primitive circuits are given by A_0. The abstract interpretation of 1-bit full-adder **fadd** after unfolding **hadd** is

$$\begin{aligned}
A(\mathbf{fadd}) &: \texttt{Nat} \times \texttt{Nat} \times \texttt{Nat} \longrightarrow \texttt{Nat} \times \texttt{Nat} \\
A(\mathbf{fadd}) &= \lambda(a,\ b,\ c).\ \texttt{let}\ (u,\ v) = (\mathbf{and}_A\ (a,\ b),\ \mathbf{xor}_A\ (a,\ b)) \\
&\qquad\qquad\qquad\ \ (w,\ x) = (\mathbf{and}_A\ (v,\ c),\ \mathbf{xor}_A\ (v,\ c)) \\
&\qquad\qquad \texttt{in}\ \ (\mathbf{or}_A\ (u,\ w),\ x)
\end{aligned}$$

We use the subscripts $_C$ and $_A$ to denote concrete and abstract circuits, respectively.

6 Verification Methodology

Given the description of a circuit and its specification, the verification process starts with interpreting the circuit description at the level of implementation (bit level) and at the level of specification (natural numbers level). We then get the corresponding *concrete circuit* and *abstract circuit*. Our verification consists only of proving, at the same high level of abstraction, that the abstract circuit satisfies the specification. The correctness of the concrete circuit follows automatically, as we explain now using a correctness criteria from data refinement.

The correctness criteria for a circuit description says that if the concrete circuit and the corresponding abstract one have appropriately related inputs then the two circuits yield appropriately related output. We use binary logical relations to relate concrete values and abstract values. The reason for using logical relations is to be able to cope with higher-order structure of circuits (*circuit combinators*). The approach that we will use is based on a categorical approach to data refinement [10].

6.1 Logical Relations

We present here the notions of logical relation between two interpretations of a circuit. Discussion on logical relations can be found in [14].

Definition 5. *Let I, J be two interpretations of circuit descriptions. Let (R_α) be a family of binary relations $(I(\alpha), J(\alpha), R_\alpha)$ indexed by object $\alpha \in \mathbf{HW}$. We say that (R_α) is a* logical relation *between the two interpretations I and J if*

- *$a\ R_{\mathbf{1}}\ b$ holds for (the unique) $a \in I(\mathbf{1})$ and $b \in J(\mathbf{1})$*
- *$(a, b)\ R_{\alpha\times\beta}\ (a', b')$ if and only if $a\ R_\alpha a'$ and $b\ R_\beta\ b'$, for all $a \in I(\alpha)$, $a' \in J(\alpha)$, $b \in I(\beta)$ and $b' \in J(\beta)$*
- *$f\ R_{[\alpha\to\beta]}\ f'$ if and only if $\forall a \in I(\alpha).\ \forall a' \in J(\alpha).\ a\ R_\alpha\ a' \Rightarrow fa\ R_\beta\ f'a'$, for all $f \in I([\alpha \to \beta])$ and $f' \in J([\alpha \to \beta])$.*

Observe that from the definition of the Cartesian closed category **Rel**, these three conditions are equivalent to $R_{\mathbf{1}} = \mathbf{1}_{Rel}$, $R_{\alpha\times\beta} = R_\alpha \times R_\beta$ and $R_{[\alpha\to\beta]} = [R_\alpha \to R_\beta]$.

6.2 Correctness Criteria

Showing that the correctness of concrete circuits follows from the correctness of the abstract ones amounts to finding an appropriate logical relation (R_α) between the two interpretations C and A such that for every circuit description *circuit* in **HW** the diagram

$$\begin{tikzcd}
C(inputs) \arrow[r, leftrightarrow, "R_{inputs}"] \arrow[d, "C(circuit)"'] & A(inputs) \arrow[d, "A(circuit)"] \\
C(outputs) \arrow[r, leftrightarrow, "R_{outputs}"'] & A(outputs)
\end{tikzcd}$$

commutes in the sense described in the category **Rel**, that is, that

$$C(circuit)\ R_{[inputs\to outputs]}\ A(circuit)$$

Thus, a family (R_α) gives, for every object α in **HW**, an object $(C(\alpha), A(\alpha), R_\alpha)$ in **Rel** such that for every arrow $c : \alpha \longrightarrow \beta$ in **HW** $(C(c), A(c))$ is an arrow in **Rel** from $(C(\alpha), A(\alpha), R_\alpha)$ to $(C(\beta), A(\beta), R_\beta)$. In other words, a logical relation R between the two interpretations C and A is equivalent to a Cartesian closed functor R from **HW** to **Rel** such that $\langle dom, cod\rangle \circ R = \langle C, A\rangle$, i.e. the following commutes.

$$\begin{tikzcd}
 & \mathbf{Rel} \arrow[d, "\langle dom, cod\rangle"] \\
\mathbf{HW} \arrow[ur, "R"] \arrow[r, "\langle C, A\rangle"'] & \mathbf{Set}\times\mathbf{Set}
\end{tikzcd}$$

Recall that C and A are the interpretations uniquely corresponding to the models C_0 and A_0, respectively. The basic lemma of logical relations is as follows.

Lemma 1 (Basic lemma of logical relations). *Given any model R_0 of the CC sketch H in* **Rel** *with $\langle dom, cod \circ R_0 = \langle C_0, A_0\rangle$, the unique CC functor (interpretation) R with $R \circ \iota = R_0$ satisfies $\langle dom, cod\rangle \circ R = \langle C, A\rangle$, i.e., R is a logical relation between the two interpretations C and A from* **HW** *to* **Set** × **Set**.

Proof. It follows from the fact that **HW** is the free CCC on the sketch H and that $\langle dom, cod\rangle$ is a strict CC functor.

Intuitively, this lemma says that the commutativity of the diagrams for compound circuits follows from the structural similarity between the abstract and concrete circuit and from the commutativity of the diagram for their primitive circuits.

To formulate the correctness criteria, what remains is to define appropriate R_{B_n} for each n and to prove the commutativity of the diagram for the primitive circuits. For example, the diagram for **and**:

$$\begin{tikzcd}
C(\mathsf{B}_1 \times \mathsf{B}_1) = \mathtt{Bit} \times \mathtt{Bit} \arrow[d, "\mathbf{and}_C"'] & \mathtt{Nat} \times \mathtt{Nat} = A(\mathsf{B}_1 \times \mathsf{B}_1) \arrow[l, leftrightarrow, "R_{\mathsf{B}_1} \times R_{\mathsf{B}_1}"'] \arrow[d, "\mathbf{and}_A"] \\
C(\mathsf{B}_1) = \mathtt{Bit} & \mathtt{Nat} = A(\mathsf{B}_1) \arrow[l, leftrightarrow, "R_{\mathsf{B}_1}"]
\end{tikzcd}$$

Note that outputs of $\mathbf{and}_A$ on inputs which are not in the domain of R is irrelevant.

Definition 6. *R_{B_1}, a relation over* `Bit` *and* `Nat`*, relates* 1 *with 1 and* 0 *with 0. R_{B_n}, a relation over* $\mathtt{Bit}^n$ *and* natt*, relates $x \in \mathtt{Bit}^n$ and $x' \in \mathtt{Nat}$ iff x is the binary representation of x'.*

The relation R does not need to be a relation over `Bit` and `Nat`. It can be defined as a relation over other two representations of numbers as long as it appropriately relates the two representations.

Example 3. Consider 1-bit full-adder **fadd**. By Lemma 1, commutativity of the diagrams for the primitive circuits guarantees that of

$$\begin{tikzcd}
\mathtt{Bit} \times \mathtt{Bit} \times \mathtt{Bit} \arrow[d, "\mathbf{fadd}_C"'] & \mathtt{Nat} \times \mathtt{Nat} \times \mathtt{Nat} \arrow[l, leftrightarrow, "R_{\mathsf{B}_1} \times R_{\mathsf{B}_1} \times R_{\mathsf{B}_1}"'] \arrow[d, "\mathbf{fadd}_A"] \\
\mathtt{Bit} \times \mathtt{Bit} & \mathtt{Nat} \times \mathtt{Nat} \arrow[l, leftrightarrow, "R_{\mathsf{B}_1} \times R_{\mathsf{B}_1}"]
\end{tikzcd}$$

Our verification task is then to prove $\mathbf{fadd}_A$ agrees with the specification

$$\mathbf{fadd}_{\text{spec}} \;=\; \lambda(a,\; b,\; c).\; ((a+b+c) \;\mathtt{div}\; 2,\; (a+b+c) \;\mathtt{mod}\; 2)$$

for $0 \le a, b, c, \le 1$. We write $\mathbf{fadd}_A \leftrightarrow \mathbf{fadd}_{\text{spec}}$ to mean $\mathbf{fadd}_A$ agrees with $\mathbf{fadd}_{\text{spec}}$ on relevant inputs in the domain of the logical relation.

6.3 Compositional Verification

Even at the high level of abstraction, formally proving that an abstract circuit meets its specification is not easy to do directly, particularly when the circuit is large. We propose here to organise a large verification as the composition of smaller verifications of component circuits. This also allows reuse of component verification and high-level verification of circuits with many different component implementations.

For the purpose of explanation, we take the simple example of a 2-bit full-adder **fadd2** having two 1-bit full-adders **fadd** as components. That $C(\mathbf{fadd2})$ is appropriately related to $A(\mathbf{fadd2})$ is automatic, but we must prove that $A(\mathbf{fadd2})$ agrees with

$$\mathbf{fadd2}_{\mathrm{spec}} \;=\; \lambda(a,\ b,\ c).\ a\ +\ b\ +\ c$$

The problem is that $\mathbf{fadd2}_{\mathrm{spec}}$ does not have the same structure as **fadd2** but we wish to use a lemma about **fadd**.

This can be naturally resolved by dividing the verification in two steps. Firstly, we verify **fadd**, i.e., prove that $A(\mathbf{fadd}) \leftrightarrow \mathbf{fadd}_{\mathrm{spec}}$. Then, we consider **fadd** to be a primitive operation and extend our sketch H to H' including **fadd**. The concrete interpretation $C' : \mathbf{HW}' \longrightarrow \mathbf{Set}$ of now extended category HW' is given by setting $C'(\mathbf{fadd}) = \mathbf{fadd}_C$. For the abstract interpretation, we set $A'(\mathbf{fadd}) = \mathbf{fadd}_{\mathrm{spec}}$, which is a simpler and higher-level formula than $\mathbf{fadd}_A$. By the first step, $C'(\mathbf{fadd})\ R_{[\mathsf{B}_1^3 \to \mathsf{B}_1^2]}\ A'(\mathbf{fadd})$, hence so is $C'(\mathbf{fadd2}) = C(\mathbf{fadd2})\ R_{[\mathsf{B}_2^2 \times \mathsf{B}_1 \to \mathsf{B}_2 \times \mathsf{B}_1]}\ A'(\mathbf{fadd})$ by Lemma 1. Thus, our verification is simplified to proving that $A'(\mathbf{fadd2}) \leftrightarrow \mathbf{fadd2}_{\mathrm{spec}}$ where $A'(\mathbf{fadd2})$ is simpler formula than $A(\mathbf{fadd2})$.

The same organisation of proofs can be obtained by considering higher-order circuit combinators. We consider $\varphi_{\mathbf{fadd2}}$ that combines **fadd** taken as an argument. The above amounts to organising the proof of $A(\mathbf{fadd2}) \leftrightarrow A(\mathbf{fadd2}_{\mathrm{spec}})$ as

$$\begin{aligned} A(\mathbf{fadd2}) &= A(\varphi_{\mathbf{fadd2}})(A(\mathbf{fadd})) \\ &\leftrightarrow A(\varphi_{\mathbf{fadd2}})(\mathbf{fadd}_{\mathrm{spec}}) \\ &\leftrightarrow \mathbf{fadd2}_{\mathrm{spec}} \end{aligned}$$

Note that, without higher-order types in the language, the two occurrences of **fadd** iin **fadd2** must be treated separately.

7 Concluding Remarks

We have presented a methodology for describing and verifying combinational arithmetic circuits. It is based on a category theoretic approach to data refinement which uses logical relation as abstraction relation. Our methodology allows us to reason about circuits at a high level of abstraction. Additionally, having higher-order structure allows us to have compositional proofs.

The idea of using data or program refinement in circuit verification has been explored before. Burch and Dill [3] use specification state and implementation state spaces connected by abstraction mapping. Their correctness criteria diagram is similar to ours. There, implementations is verified directly against specifications. Cyrluk [4] uses abstraction mapping between specification state and implementation state spaces to guide in structuring proof by transforming terms involving implementation state variables into equivalent specification state variables. Velev and Bryant [18] combine the approach of Burch and Dill with abstraction methods for functional units in microprocessors. Jones and Sheeran [9] view circuit design as program refinement. Hanna et al [6] use the techniques of data abstraction to build up specifications in a structured way.

Various uses of category theory as a guide in circuit verification have been explored. Brown and Hutton [2] used category theory to formalise the use of picture in circuit design. Fourman [5] and Sabadini et al. [15] have developed categorical models of sequential circuits. Sheeran [16] has shown that category theory can be used to derive useful theorems about hardware verification.

Further works are to employ the methodology in practice to verify non-trivial arithmetic circuits and to extend the methodology for verifying sequential circuits. For these, we enrich the hardware description languages in terms of both practicality and expressiveness, e.g. polymorphism and dependent types.

Acknowledgement

I am grateful to Mary Sheeran and Makoto Takeyama for their guidance in this work. I would like also to thank Daniel Fridlender and anonymous referees for their comments on the earlier version of this paper. Diagrams in this paper are drawn using the diagram environment by Paul Taylor [17].

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice Hall International, London, 2nd edition, 1990.

[2] C. Brown and G. Hutton. Categories, Allegories, and Circuit Design. In *Proc. of 10th Annual IEEE symposium on Logic in Computer Science.* IEEE Computer Society Press, 1994.

[3] J. Burch and D. Dill. Automated Verification of Pipelined Microprocessors Control. In *Proc. of 6th International Conference Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*. Springer-Verlag, 1994.

[4] D. Cyrluk. Inverting the Abstraction Mapping: A Methodology for Hardware Verification. In M. Srivas and A. Camilleri, editors, *Proc. of the First International Conference Formal Methods in Computer-Aided Design '96*, volume 1166 of *LNCS*, Berlin, 1996. Springer-Verlag.

[5] M. P. Fourman. Proof and design: Machine-assisted formal proof as a basis for foundation of computer science. LFCS report ECS-LFCS-95-319, Laboratory for Foundation of Computer Science, The University of Edinburgh, Edinburgh, 1995.

[6] K. Hanna, N. Daeche, and M. Longley. Veritas$^+$: A Specification Language Based on Type Theory. In M. E. Lesser and G. M. Brown, editors, *Proc. of International Workshop on Hardware Specification, and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, New York, 1990. Springer-Verlag.

[7] C. Hoare. Proof of Correctness of Data Representation. *Acta Informatica*, 1, 1972.

[8] H. Jifeng and C. Hoare. Data Refinement in a Categorical Setting. Technical Monograph PRG-90, Oxford University Computing Laboratory, Programming Research Group, Oxford, 1990.

[9] G. Jones and M. Sheeran. Circuit Design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, IFIP WG 10.5 Lecture Notes, chapter 1. North-Holland, Amsterdam, 1990.

[10] Y. Kinoshita, P. O'Hearn, A. Power, M. Takeyama, and R. Tennent. An Axiomatic Approach to Binary Logical Relation with Applications to Data Refinement. In M. Abadi and T. Ito, editors, *Proc. of Third International Symposium on Theoretical Aspects of Computer Software '97*, volume 1281 of *LNCS*, Berlin, 1997. Springer-Verlag.

[11] Y. Kinoshita, A. Power, and M. Takeyama. Sketches. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[12] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic.* Number 7 in Cambridge studies in advanced mathematics. Cambridge University Press, 1986.

[13] S. Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, New York, 2nd edition, 1998.

[14] J. Mitchell. *Foundations for Programming Language.* Foundations of Computing Series. MIT Press, 1996.

[15] N. Sabadini, R. Walters, and H. Weld. On categories of asynchronous circuits. `http://cat.maths.usyd.edu.au/~bob/papers/asyncirci.ps`, 1994.

[16] M. Sheeran. Categories for the Working Hardware Designer. In M. E. Lesser and G. M. Brown, editors, *Proc. of International Workshop on Hardware Specification, and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, New York, 1990. Springer-Verlag.

[17] P. Taylor. *Commutative Diagrams in TEX(version 4).* Department of Computer Science, Queen Mary and Westfield College, London, 1997.

[18] M. N. Velev and R. E. Bryant. Bit-Level Abstractiion in the Verification of Pipelined Microprocessors by Correspondence Checking. In G. Gopalakrishnan and P. Windley, editors, *Proc. of the Second International Conference Formal Methods in Computer-Aided Design '98*, volume 1522 of *LNCS*, Berlin, 1998. Springer-Verlag.

Lemma Generalization and Non-unit Lemma Matching for Model Elimination[⋆]

Koji Iwanuma and Kenichi Kishino

Department of Computer Science and Media Engineering
Yamanashi University
Kofu, Yamanashi, 400-8511, Japan.
iwanuma@esi.yamanashi.ac.jp

Abstract. In this paper, we study two lemma methods for accelerating Loveland's model elimination calculus: One is lemma generalization and another is non-unit lemma matching. The derivation of lemmas in this paper is a dynamic one, i.e., lemma generation is repeatedly performed during an entire refutation search process. A derived lemma is immediately generalized by investigating the obtained subproof of the lemma. The lemma generalization increases the possibility of successful applications of the lemma matching rule. The non-unit lemma matching is an extension of the previously proposed unit lemma matching, and has the ability for stably speeding up model elimination calculus by monotonically reducing the refutation search space. We have implemented a PTTP-based theorem prover, named I-THOP, which performs unit lemma generalization and 2-literal non-unit lemma matching. We report good experimental results obtained with I-THOP.

1 Introduction

The *lemma* facility in top-down theorem proving has been pointed out to be quite important and beneficial in order to avoid a redundant computation. Lemmas are extra clauses which are derivable during a proof search, and can be identified with initial axiom clauses once they are produced. The use of lemmas makes it possible for a top-down theorem prover both to find a shorter proof and to cut off duplicate computations.

Many first-order theorem provers utilize various sorts of lemma facility [1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 14, 16, 18, 20, 21, 22]. However, the naive use of lemmas often increases the breadth of then search space. It consequently causes an explosion, and results in loss of efficiency of provers in many cases [7, 23, 24]. Automated theorem provers had been struggled for extreme deterioration and instability of the proving performance caused by lemmata for a long time. However several technologies for overcoming such difficulty have been proposed

⋆ This research was supported partly by Telecommunications Advancement Organization of Japan (TAO), and also partly by Grant-in-Aid from The Ministry of Education, Science and Culture of Japan.

in this decade. Caching, proposed by Astrachan and Stickel [3], is a modified method of lemmas. It succeeds in drastically reducing the search space of Horn problems. However caching has no effect for non-Horn problems. We proposed various unit lemma matching rules for model elimination calculus [11, 12, 13], which are applicable non-Horn problems, and can stably accelerate a refutation search process by monotonically reducing the search space. In this paper, we shall extend this unit lemma matching method in two manners: One is lemma generalization, and another is an extended matching which is applicable to non-unit lemmas.

The lemma generation process of this paper is a dynamic one, i.e., lemmas are repeatedly derived during an entire proof search. Generated lemmas are generalized, and are stored into a lemma database. The lemma generalization is a subproof-based one, i.e., it can be performed by investigating the structure of the obtained subproof of a lemma. The computational cost of this subproof-based generalization is quite low. This real-time generalization of a lemma obtained at a search stage clearly increases the possibility of controlled use of the lemma at the later stages. Such a dynamic lemma generalization technique has never been studied, up to our knowledge, in the literature of fully-automated theorem proving technology.[1] First we describe the lemma generalization method in a general form. We have implemented a PTTP-based theorem prover, named I-THOP, which can perform unit lemma generalization. Second we give an experimental evaluation of the good performance of this restricted form, i.e., unit lemma generalization with I-THOP.

Next, we give the non-unit lemma matching, which is an extension of the unit lemma matching [11]. The non-unit lemma matching never broadens the breadth of the search space, and makes it possible for a top-down theorem prover both to find a shorter proof and to cut off duplicate computations.

Roughly speaking, non-unit lemma technology is classified into two subcategories: One is the *local* use of non-unit lemmas such as C-reduction, folding-down operation [14, 22] and anti-lemma [10, 14], and another is the *global* one. Although little research has been achieved on the global use of non-unit lemmas until now, Fuchs [8, 9] investigated a controlled global use of non-unit lemmas. However Fuchs' work is an extension of Schumann [21], and its lemma generation method is static one, i.e., lemma production is performed only once at the preprocessing stage. The non-unit lemma matching studied in this paper is a sort of the global use of non-unit lemmas with the dynamic lemma generation. Compared with the static one, the dynamic generation has the great advantage that several difficult lemmas can be derived by using other lemmas obtained at the previous stages. Such a nested lemma generation quite often occurs in real

[1] Interactive theorem provers, especially the ones using the induction principle, sometimes perform the generalization of intermediately obtained lemmas [5]. Also some similar techniques such as explanation-based learning have been studied in the field of machine learning [19].

theorem proving processes.[2] Up to our knowledge, the global use of non-unit lemmas together with dynamic generation has also never been studied up to this point. We implemented 2-literal non-unit lemma matching rule in I-THOP, and show some good experimental results for this restricted version of non-unit lemma matching.

This paper is organized as follows: Section 2 is preliminaries. In Sect.3, firstly we give a dynamic lemma generation method; secondly we show a lemma generalization technology which is based on the investigation of a subproof; finally we give the non-unit lemma matching rule. Section 4 shows experimental results of the performance of the dynamic lemma generation, the unit lemma generalization and 2-literal non-unit lemma matching. Section 5 is the conclusion.

2 Preliminaries

We briefly introduce *Tableau Model Elimination* (TME) [8, 14], (or sometimes called *connection tableau calculus*), which is a generalization of the chain-based model elimination calculus originally proposed by Loveland [15].

A *clause* is a multi-set of literals, usually written as a disjunction $L_1 \vee \ldots \vee L_n$.

Definition 1. A *tableau* T for a set S of clauses is a tree whose non-root nodes are labeled with literals and these labels satisfy the condition: if the immediate successor nodes $N_1, \ldots, N_n$ of a node of T are labeled with literals $L_1, \ldots, L_n$, then the clause $L_1 \vee \ldots \vee L_n$ (*tableau clause*) is an instance of a clause in S. A tableau is called a *Model Elimination (ME) tableau*[3] if, for every inner node N (except the root) labeled with a literal L, there is an immediate successor node N' of N such that N' is labeled with a literal L' complementary to L.

The *trivial* tableau is a one-node tableau with a root only. If no confusion arises, we shall identify a node N of a tableau with the literal associated with N in an appropriate manner.

Let T be a tableau. A node N *dominates* a node N' in T if $\langle N', N \rangle$ is in the transitive closure of the immediate predecessor relation. An *open* branch in T is a branch of T not containing two complementary literals. A *closed* tableau is a tableau which does not have any open branches. A *subgoal* of T is a literal at the leaf node N of an open branch in T. We also sometimes call N itself a subgoal.

Given a tableau T, a subgoal L and a clause C, then the *tableau expansion* of L in T with C is the following operation: firstly obtain a new variant $L_1 \vee \ldots \vee L_n$ of C; secondly attach new leaf nodes $N_1, \ldots, N_n$ as immediate successors of L in T; finally label these new leaves with the literals $L_1, \ldots, L_n$, respectively.

Definition 2. Given a set S of clauses, the inference rules of TME for S consists of the followings:

[2] We indeed confirmed this phenomenon through many experiments using TPTP library, by examining the tableau proofs produced by I-THOP.

[3] An ME tableau is sometimes called a *connection tableau.*

Start. Given a trivial tableau, select a clause C from S, and apply the tableau expansion rule with C to the root of the tableau.

Reduction. Given a tableau T, select a subgoal L in T and a node L' dominating L such that there is a most general unifier θ of L and $\neg L'$, then apply θ to the whole T. We say, *L is reduced by the ancestor L'*.

Extension. Given a tableau T, select a subgoal L in T and a clause C from S, apply a tableau expansion step with C to L and immediately perform a reduction step with L and one of its newly created successors.

Given a ME tableau, the TME inference rules clearly generates only ME tableaux. Throughout the rest of this paper, we assume any tableaux are ME tableaux, thus we simply use the word *tableau* for ME tableau.

Definition 3. Let S be a set of clauses, and T and T' be two tableaux. A sequence $T_1, \ldots, T_n$ is called a *TME derivation* in S from T to T' if

1. $T_1 = T$ and $T_n = T'$ hold, and
2. T_{i+1} $(i = 1, \ldots, n-1)$ is obtained from T_i by means of one of TME inference rules of S, i.e., a start step, an extension step or a reduction step.

A *TME refutation from T* is a TME derivation from T to a closed tableau. Finally, a *TME refutation of S* is a TME derivation in S from a trivial tableau to a closed tableau.

In the following, an inference I performed in a TME derivation is written as a tuple $\langle s, \langle r, a\rangle\rangle$ where s is the subgoal, r specifies the inference rule applied to s, and a indicates the input clause applied in the extension or start rules or the ancestor literal used in the reduction. Furthermore, we sometimes identify a TME derivation $T_1, \ldots, T_n$ with a sequence $I_1, \ldots, I_{n-1}$ of inferences such that I_j has been applied to T_j $(j = 1, \ldots, n-1)$.

Theorem 1 (Completeness [14, 15]). *Let S be a set of clauses. S is unsatisfiable iff there is a TME refutation of S.*

Example 1. Fig.1 depicts an example of a closed tableau for the set of clauses.

$$\{\neg p(X),\quad p(X) \vee q(X),\quad p(X) \vee \neg q(X) \vee \neg r(X),\quad r(a)\}$$

The dotted line in Fig.1 denotes the reduction of a subgoal with its ancestor which is not an immediate predecessor.

3 Lemmas in Model Elimination

Lemmas are extra clauses which are derivable during a proof search, and can be identified with initial axiom clauses once they are produced. The use of lemmas makes it possible for a tableau-based theorem prover both to find a shorter proof and to cut off duplicate computations.

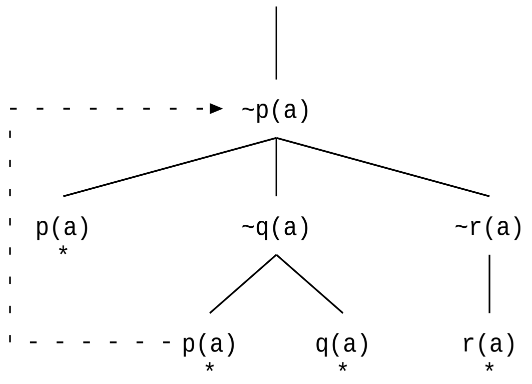


Fig. 1. A closed ME tableau

3.1 Dynamic Lemma Generation

Clausal non-unit lemmas can be created from a closed subtableau as follows:

Definition 4. Let T be a non-trivial tableau and L be a literal. We say T is an *ME tableau with the head* L, denoted as T^L, if the root node of T is labeled by L and there is an immediate successor labeled with a literal L' which is complementary to L.

If T_s is a subtree of a tableau T, and contains only a node N labeled by L and all nodes dominated by N, then T_s is called a *subtableau* of T with the head L. A lemma can be produced from a subtableau T_s of T if T_s is closed in the context of T.

Definition 5 (Lemma). Let T be a tableau, and T_s^L be a subtableau of T with the head L. Suppose T_s^L contains no open branches of T, and $M_1, \ldots, M_n$ are all leaf literals in T_s^L such that M_i $(i = 1, \ldots, n)$ is reduced by an ancestor literal which dominates L, i.e., an ancestor literal occurring outside T_s^L. Then the clause $\neg L \vee M_1 \vee \ldots \vee M_n$ is called a *lemma produced by the subtableau* T_s^L.

Notice if there are no leaf literals in T_s^L reduced by ancestor literals outside T_s^L, then the produced lemma becomes a unit clause.

Proposition 1. *Let S be a set of clauses, T be a tableau of S, and C be a lemma produced by a subtableau T_s of T. Then C is a logical consequence of S.*

We dynamically and repeatedly perform the above lemma generation procedure during the search of a TME refutation. The dynamic generation enables us to derive a difficult lemma, which ordinarily needs a huge subtableau, by using other lemmas obtained at the previous stages.

3.2 Lemma Generalization

Let us consider the generalization of lemmas. Suppose L_1 and L_2 are literals. We say L_1 is *more general* than (or simply, *subsumes*) L_2 if there is a substitution θ such that $L_1\theta = L_2$.

Definition 6. We say a tableau T_1 *is more general* than a tableau T_2, denoted by $T_1 \leq T_2$, if the tree structures of T_1 and T_2 are isomorphic and there is a substitution θ such that $L_1\theta = L_2$ for any pair of corresponding nodes L_1 and L_2 in T_1 and T_2, respectively.

Definition 7. Let S be a set of clauses. Suppose C_1 and C_2 are lemmas produced by tableaux T_1 and T_2 of S, respectively. We say C_1 is *more proof-based general* than C_2, denoted by $C_1 \leq_p^S C_2$, if $T_1 \leq T_2$. We denote $C_1 <_p^S C_2$ if $C_1 \leq_p^S C_2$ but not $C_2 \leq_p^S C_1$.

Given a set S of clauses, the relation $<_p^S$ obviously constitutes a well-founded order. Thus there is a *most proof-based general* lemma for each lemma C, denoted as mpg(C). That is, mpg(C) is a clause satisfying the condition:

1. mpg(C) $\leq_p^S C$, and;
2. there is no clause C' such that $C' <_p^S$ mpg(C).

The clause mpg(C) is unique modulo renaming variables.

Given a lemma C produced by a tableau T_s^L in a TME derivation, the clause mpg(C) can easily be obtained by investigating the inference steps needed for constructing T_s^L.

Definition 8. Let D be a TME derivation for a tableau T, and T_s^L be a subtableau of T with the head L. The *inference sequence for* T_s^L is the order-preserved sequence $I_1, \ldots, I_n$ of all inferences of D such that I_j $(j = 1, \ldots, n)$ has been performed to either the node L or a node dominated by L in T_s^L during D.

Obviously, the first inference step in any inference sequence must be performed to the head node of the subtableau.

Definition 9. Let L be a literal, p be the predicate symbol of L and n be the arity of p. If L is positive, then the *skeleton* of L is the literal $p(X_1, \ldots, X_n)$; otherwise $\neg p(X_1, \ldots, X_n)$.

Clearly the skeleton of L is the most general form of L.

Definition 10 (Generalization Algorithm). Let T be a tableau, T_s^L be a subtableau with the head L, and $I = I_1, \ldots, I_n$ be an inference sequence for T_s^L. Obtain a *generalized subtableau*, gene(T_s^L), from T_s^L and I by performing the following procedure:

1. First, get a tableau T_1 by applying the first inference I_1 to the one-node tableau labeled by the skeleton of L.
2. Obtain the tableau T_{i+1}, for each $i = 1, \cdots, n-1$, as follows:
   (a) If I_{i+1} is a reduction step with an ancestor literal outside T^L, i.e., an ancestor literal dominating L, then skip the inference step I_{i+1} and return the current T_i as T_{i+1};
   (b) Otherwise, perform simply the inference step I_{i+1} to the corresponding subgoal in the tableau T_i, and return the resulting tableau as T_{i+1}.
3. Finally, return the resulting tableau T_n as the answer gene(T^L_s).

Let $C = \neg L \vee M_1 \vee \ldots \vee M_k$ be a lemma produced by a subtableau T^L_s. Clearly, the head literal L' of gene(T^L_s) is more general than L. Moreover there are some literals $M'_1, \ldots, M'_k$ in gene(T^L_s) corresponding $M_1, \ldots, M_k$, respectively, and each M'_i $(i = 1, \cdots, k)$ is more general than the corresponding M_i. The clause $C' = \neg L' \vee M'_1 \vee \ldots \vee M'_k$ is called a *generalized lemma produced by* gene(T^L_s). We say C' is *properly generalized* if $C' <^S_p C$.

Example 2. Consider the set S of clauses containing the three clauses:

$$p(X, Y) \vee \neg q(X, Y) \vee \neg r(Y) \tag{1}$$
$$p(X, Y) \vee q(X, Y) \tag{2}$$
$$r(a) \tag{3}$$

Fig.2 depicts two subtableau for S. Assume that the tree traverse for tableau construction is performed in the left-to-right and depth-first order. We can obtain the unit lemma $p(a, a)$ from the left subtableau. However we can not apply this lemma to the subgoal $\neg p(b, a)$ which appears in the right subtableau at the later stage.

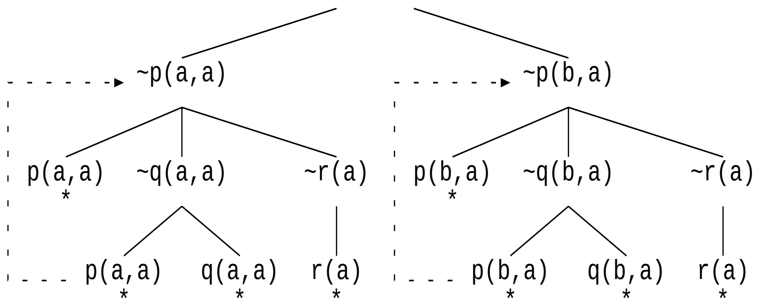


Fig. 2. A tableau without lemma generalization

Notice the lemma $p(a, a)$ can immediately be generalized by investigating the left subtableau. The skeleton of $\neg p(a, a)$ is the literal $\neg p(X, Y)$, and the inference sequence for the left subtableau for $\neg p(a, a)$ is

$$\begin{aligned}
I_1 &: \langle \neg p(a,a),\ \langle \text{extension},\ \text{Clause (1)}\rangle\rangle \\
I_2 &: \langle \neg q(a,a),\ \langle \text{extension},\ \text{Clause (2)}\rangle\rangle \\
I_3 &: \langle p(a,a),\ \langle \text{reduction},\ \text{Ancestor } \neg p(a,a)\rangle\rangle \\
I_4 &: \langle \neg r(a),\ \langle \text{extension},\ \text{Clause (3)}\rangle\rangle
\end{aligned}$$

Therefore we can construct a generalized tableau by repeating these inference steps, i.e., first we apply the extension to the skeleton $\neg p(X,Y)$ with the clause (1); second the extension to the immediate successor $\neg q(X,Y)$ with the clause (2); third the reduction with the head literal $\neg p(X,Y)$; finally the extension with the clause (3). The newly obtained subtableau has the head literal $\neg p(X,a)$, where the first argument is not instantiated. $\neg p(X,a)$ is properly generalized from $\neg p(a,a)$. If we store the unit clause $p(X,a)$ as a lemma, instead of $p(a,a)$, then we can reduce the right subtableau in Fig.2, because the head literal $\neg p(b,a)$ can immediately be resolved with the lemma $p(X,a)$, as depicted in Fig.3.

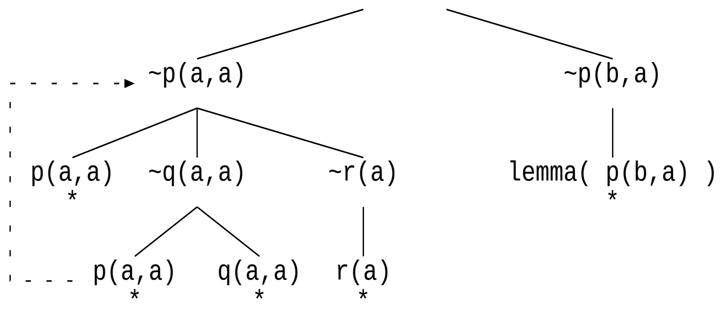


Fig. 3. A tableau with the generalized lemma $p(X,a)$

Theorem 2. *Let S be a set of clauses, C is a lemma produced by a subtableau T_s^L of S, and C' is a generalized lemma produced by gene(T_s^L). Then C' is a logical consequence of S. Moreover C' is the most proof-based general, that is, C' is* mpg(C).

3.3 Lemma Matching Rule

Definition 11 (Naive Lemma Rule). Given a tableau T and a set U of lemmas generated up to the current stage, then select a subgoal L and a lemma M from U, apply a tableau expansion with M to L and immediately perform a reduction step with L and one of its newly created successors.

The naive use of lemmas often increases the breadth of the search space, and consequently causes its explosion in many cases. Any naive methods utilizing lemmas often result in loss of efficiency of theorem prover. Of course, there is a fortunate case where search space is drastically reduced even with a naive lemma method. However, the most important and difficult problem of lemma methods is how to stabilize the effect of lemmaizing methods.

Definition 12 (Mandatory Inference Rule). Let I be an inference rule, T be a tableau and T' be the result of performing the rule I to T. We say I is *mandatory* when, if there is no TME refutation from T', then no TME refutation from T exists.

If an inference rule I is mandatory and is successfully applied to a tableau T, then we may immediately discard all other alternative inference rules to T at this choice point, without losing completeness. Therefore any mandatory rule I never causes the explosion of the search space if such an immediate cut operation is performed. all mandatory rules have the great ability for accelerating the refutation search by monotonically reducing the search space.

Definition 13 (Unit Lemma Matching Rule [11]). Given a tableau T and a set U of lemmas generated up to the current stage, then select a subgoal L and a unit lemma A from U. If there is a substitution θ such that $A\theta = \neg L$, then apply a tableau expansion with A to L and immediately perform the substitution θ to the newly created successor of L.

Obviously the branch of T expanded by unit lemma matching is closed. Notice that the unit lemma matching is a mandatory inference rule. This rule allows us to use unit lemmas without causing the explosion of the search space. It has no need of extra pruning operations, which would be necessary for cutting off some redundancy caused by naive use of lemmas [11]. In this paper, we extend this matching rule for non-unit lemmas.

Definition 14 (Non-Unit Lemma Matching Rule). Given a tableau T and a set U of lemmas generated up to the current stage, then select a subgoal L in T and non-unit lemma $A_1 \vee \ldots \vee A_n$ from U, and moreover select a literal, say A_1 from $A_1, \ldots, A_n$. If

1. there is a substitution θ such that $A_1\theta = \neg L$, and
2. there are literals $L_2, \ldots, L_n$ in T such that $L_2, \ldots, L_n$ dominate L and each L_i is complementary to $A_i\theta$ for $i = 2, \ldots, n$,

then apply a tableau expansion with $A_1 \vee \ldots \vee A_n$ to L, and immediately perform the substitution θ to all newly created n successors $A_1, \ldots, A_n$.

Obviously, the new n branches containing L, which have been expanded by the non-unit lemma matching rule, are closed. The non-unit lemma matching is clearly a mandatory rule, because no substitution is performed to any variables appearing in the tableau T. Notice the subgoal L can be solved in the most general form if non-unit lemma matching is successfully applied. Non-unit lemma matching has the great power for reducing a search space of a TME refutation without a harmful side-effect.

4 Experimental Results

We developed a TME theorem prover, named I-THOP, which is based on PTTP technology [23, 24], and has additional features of dynamic lemma generation and various sorts of lemma matching, such as unit lemma matching, identical C-reduction and strong contraction, etc [11, 13]. I-THOP participated in CASC-14 (CADE-14 Automated theorem prover System Competition) [12, 26], and is one of the advanced theorem provers among the world.

We are now developing an extended version of I-THOP. So far, we have finished implementing some restricted forms of lemma generalization and non-unit lemma matching, i.e., *unit lemma* generalization and *2-literal non-unit lemma* matching, respectively. We shall use this tentative version for experimental evaluation of our proposals.

We used as benchmark problems the set of eligible test problems for CASC-14. The eligible test problems are carefully selected from the TPTP problem library [25] in order to distinguish the theorem proving features of the competitors of CASC-14, all of which are state of the art theorem provers, such as OTTER [17], SETHEO [10, 18] etc. Thus these test problems are rather difficult and high-level problems. The eligible problems consists of 420 unsatisfiable clausal theories, more precisely 234 Horn problems and 186 non-Horn problems. We ran an extended version of I-THOP for these test problems on SGI O2 workstations (R10000/174MHz CPU and 96MB memory).

Table 1 shows the performance of unit lemma generalization. The column "I-THOP" is the result of I-THOP,[4] and "ULG" is for the extended I-THOP involving unit lemma generalization. The line "Solved Problems" indicates the number of problems which can be solved in the limited time 600 CPU seconds. "CPU time" shows the average CPU time (seconds) of all problems commonly solved by I-THOP and ULG.[5] "Inferences" also gives the average number of the inference steps needed for finding a refutation of commonly solved problems.

Table 1. Results of unit lemma generalization

	Horn		Non-Horn	
	I-THOP	ULG	I-THOP	ULG
Solved Problems	105	109	82	87
CPU time (sec)	82.7	63.2	30.2	28.5
Inferences	94522.5	82797.0	40334.2	39833.6

[4] Throughout this paper, I-THOP and its extended version always use the best lemma matching mode, i.e., unit-lemma matching, identical C-reduction and strong contraction and so on, which was previously developed in [11].

[5] Indeed, all problems solved by the original I-THOP can also be solved by the extended I-THOP. This shows the lemma generalization has no harmful side effects. Exactly, commonly solved problems are 105 problems of Horn, and are 82 problems of non-Horn.

Unit lemma generalization has the great effect for Horn problems. The average CPU time of ULG is 25% faster than the one of the original I-THOP. Additionally, four and five problems can newly be solved in Horn category and non-Horn category, respectively.

Table 2 shows the statistics of unit lemma generalization and its matching. "Total Lemmas" specifies the total number of unit lemmas generated during the entire refutation search processes for all test problems. "Generalized Lemmas" shows the number of unit lemmas properly generalized. "Failure" indicates the one of unit lemmas not properly generalized.

"Not-Supported Lemmas" shows the numbers of unit lemmas whose subproofs are constructed with some extra inference rules, such as C-reduction or Strong Contraction [11, 13]. These additional inference rules are valuable for solving non-Horn problems [11, 13], but unfortunately are not supported by the current implementation for unit lemma generalization. "Lemma Matchings" indicates the average numbers of the success of unit lemma matching for all problems, and "Matchings with GUL" shows those of lemma matching using properly generalized unit lemmas.

Table 2. Statistics of Generalization of unit lemmas

	Horn	Non-Horn
Total Lemmas	14502	13158
Generalized Lemmas	6406 (44.2%)	1859 (14.1%)
Failure	8096 (55.8%)	11206 (85.2 %)
Not-Supported Lemmas	0 (0 %)	93 (0.7%)
Lemma Matchings	547.7	320.1
Matchings with GUL	294.4 (53.4%)	221.0 (69.0%)

Unit lemma generalization is quite valuable for Horn problems. We can properly generalize 44% of generated unit lemmas in Horn problems. Moreover 54% of unit lemmas used in lemma matching are properly generalized lemmas. However non-Horn problems seem to have different features than Horn problems. Only 14% of unit lemmas can be properly generalized, but almost 70% of unit lemma matchings utilize some of the generalized lemmas.

I-THOP always retains the inference sequence performed for the current tentative tableau in a bit sophisticated way. Thus, at any lemma generation phases, we can immediately extract, from the entire sequence, an inference subsequence of constructing the subtableau of the target lemma. Furthermore a generalized inference sequence can be reconstructed within a linear time, using an indexing of inference rules. The experiment clarified that the overhead of the unit lemma generalization task is only less than 2% of the entire inference time, and thus is negligible. The details are omitted here due to space.

Table 3 shows the performance of the 2-literal non-unit lemma matching rule applied for 186 non-Horn test problems. The column "2L-mat" is the one of an extended I-THOP using 2-literal lemma matching. “2L-naive” is the result with the naive use of 2-literal lemmas. The last “2L-mat & ULG” indicates the performance of for an integrated version of 2-literal lemma matching and unit lemma generalization methods.

The meanings of “Solved Problems”, “CPU time” and “Inferences” are similar to those of Table 1. The numbers listed in “CPU time” and “Inferences” are the average numbers of those obtained from commonly solved 63 problems, respectively. “Used 2L-lemmas” indicates the average number of 2-literal lemmas used in each refutation search.

Table 3. Results of 2-literal lemma rules

	I-THOP	2L-mat	2L-naive	2L-mat & ULG
Solved Problems	82	88	64	93
CPU time (sec)	10.7	10.9	21.3	11.2
Inferences	15154.2	14251.5	23256.4	14239.6
Used 2L-lemmas	–	13.9	7235.8	13.9

This experimental result clearly shows that the naive use of 2-literal lemmas is quite harmful because there is a tendency of increasing the numbers of alternatives of each inference step in refutation search. The naive use of 2-literal lemmas often causes search space explosion. Contrasting with it, 2-literal lemma matching is quite stable and beneficial for solving more problems in a limited time. Moreover, two methods proposed here, unit lemma generalization and 2-literal lemma matching, can much cooperatively work with each other. The total number of solved problems rises to 93 problems by integrating them.

5 Conclusion

In this paper, we studied the lemma generalization based on the subproof. Also we gave the non-unit lemma matching rule for developing a new controlled use of non-unit lemmas. Moreover we showed good performances of some restricted forms of these methods, i.e., unit lemma generalization and 2-literal non-unit lemma matching, throughout experiments. We emphasize that the lemma generation of this paper is achieved during a refutation search process, dynamically and repeatedly. Thus lemma generalization should also be performed dynamically and repeatedly. The overhead of lemma generalization task was verified to be quite little and negligible, throughout an experiment. So far, the cooperative integration of dynamic lemma generation, real-time lemma generalization and the global use of non-unit lemmas has never been studied, to our knowledge, from a unified viewpoint.

Non-unit lemma generalization seems to have a great ability for making TME refutations much shorter. Our future work is an experimental research for the general forms of non-unit lemma generalization and non-unit lemma matching rule.

References

[1] Owen Astrachan, METEOR: exploring model elimination theorem proving, *J. Automated Reasoning* **13** (1994) 283-296.

[2] O.L. Astrachan and D.W. Loveland, The use of lemmas in the model elimination procedure, *J. Automated Reasoning* **19** (1997) 117-141.

[3] O.L. Astrachan and M.E. Stickel, Caching and lemmaizing in model elimination theorem provers, *Proc. CADE-11, LNAI* **607** (1992) 224-238.

[4] P. Baumgartner and U. Furbach, Model elimination without contrapositives and its application to PTTP, *J. Automated Reasoning* **13** (1994) 339-360.

[5] R.S. Boyer and J.S. Moore, *A Computational Logic Handbook* (Academic Press, Inc.) (1988)

[6] D. Brand, Proving theorems with the modification method, *SIAM Journal of Computing* **4** (4) (1975) 412-430.

[7] S. Fleisig, D. Loveland, A.K. Smiley III and D.L. Yarmush, An implementation of the model elimination proof procedure. *J. ACM* **21** (1974) 124-139.

[8] M. Fuchs, Controlled use of clausal lemmas in connection tableau calculi, Technical Report AR98-02, Institute für Informatik TU Müchen (1998)

[9] M. Fuchs, Lemma generation for model elimination by combining top-down and bottom-up inference, *Proc. IJCAI-99* (1999) 4-9.

[10] C. Goller, R. Letz, K. Mayr and J. Schumann, SETHEO V3.2: recent developments –system abstract–, *Proc. CADE-12, LNAI* **814** (1994) 778-782.

[11] K. Iwanuma, Lemma matching for a PTTP-based Top-down Theorem Prover, *Proc. CADE-14, LNAI* **1249** (1997) 146-160.

[12] K. Iwanuma, I-THOP Ver 1.01, in: C. Suttner and G. Sutcliffe, The CADE-14 ATP System Competition, *J. Automated Reasoning* **21** (1998) 108.

[13] K. Iwanuma and K. Oota, Strong contraction in model elimination calculus: implementation in a PTTP-based theorem prover, *IEICE trans. on Infor. & Systems* **E81-D** (5) (1998) 464-471.

[14] R. Letz, C. Goller and K. Mayr, Controlled integration of the cut rule into connection tableau calculi, *J. Automated Reasoning* **13** (1994) 297-338.

[15] D.W. Loveland, *Automated Theorem Proving: a logical basis* (North-Holland Publishing Company, Amsterdam, 1978).

[16] R. Manthey and F. Bry, SATCHMO: a theorem prover implemented in Prolog, *Proc. CADE-9, LNAI* **310** (1988) 415-434.

[17] W.W. McCune, *OTTER 3.0 reference manual and guide*, Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[18] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann and K. Mayr, SETHEO and E-SETHEO, *J. Automated Reasoning* **18** (1997) 237-246.

[19] T.M. Mitchell, R.M.Keller, S.T.Kedar-Cabelli, Explanation-based generalization: a unifying view, *Machine Learning* **1** (1986).

[20] D.A. Plaisted, Non-Horn clause logic programming without contrapositives, *J. Automated Reasoning* **4** (1988) 287-325.

[21] J.M.Ph. Schumann, A bottom-up preprocessor for top-down theorem provers – system abstract–, *Proc. CADE-12, LNAI* **814** (1994) 774–777.

[22] R.E. Shostak Refutation graphs, *Artif. Intell.* **7** (1976) 51-64.

[23] M.E. Stickel, A prolog technology theorem prover: Implementation by an extended prolog compiler, *J. Automated Reasoning* **4** (1988) 353-380.

[24] M.E. Stickel, A prolog technology theorem prover: a new exposition and implementation in prolog, *Theoret. Comput. Sci.* **104** (1992) 109-128.

[25] C. Suttner and G. Sutcliffe, *The TPTP problem library: TPTP v1.2.1,* Technical Report AR-96-02, Institute für Informatik TU Müchen (1996)

[26] C. Suttner and G. Sutcliffe, The CADE-14 ATP System Competition, *J. of Automated Reasoning* **21** (1998) 99-134.

On Automating Inductive and Non-inductive Termination Methods

Fairouz Kamareddine and François Monin[⋆]

Department of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh EH14 4AS, Scotland,
`fairouz@cee.hw.ac.uk`, `monin@cee.hw.ac.uk`

Abstract. The *Coq* and *ProPre* systems show the automated termination of a recursive function by first constructing a tree associated with the specification of the function which satisfies a notion of *terminal property* and then verifying that this construction process is formally correct. However, those two steps strongly depend on inductive principles and hence *Coq* and *ProPre* can only deal with the termination proofs that are inductive. There are however many functions for which the termination proofs are non-inductive. In this article, we attempt to extend the class of functions whose proofs can be done automatically à la *Coq* and *ProPre* to a larger class including functions whose termination proofs are not inductive. We do this by extending the terminal property notion and replacing the verification step above by one that searches for a decreasing measure which can be used to establish the termination of the function.

1 Introduction

Termination is an important property in the verification of programs defined on recursive data structure that use automated deduction. While the problem is undecidable, several proof methods of termination of functions have been developed using for instance formal proof methods in functional programming or orderings of rewriting systems. For instance we mention polynomial interpretations [3,9,21], recursive path orderings [14] and Knuth-Bendix orderings [7,12]. The latter methods are characterized by orderings called simplification orderings [5,18] and deal with the termination of functions called *simply terminating functions*. Some functions that are *non-simply terminating* can be proven to terminate with methods based on structural inductive proofs because they focus on recursive functions which can be viewed as *sorted constructor systems* that allow reasoning on the orderings' structure of the data objects.

But there are other recursive functions that are non-simply terminating for which inductive methods fail to prove the termination since precisely the structural orderings on the terms of the algebra cannot be used. These functions, which we call, *non-inductive-simply terminating functions* form an important class of functions used in recursive data structures and hence, automatically

[⋆] Supported by EPSRC GR/L15685.

establishing their termination is an important property. This is witnessed by the recent literature where techniques coming from rewriting [6,1,2,8] have been proposed to automate the termination of such functions.

We are interested in automating the termination (inductive or non-inductive) of recursive functions in a theorem proving framework. We choose the framework of the system called *ProPre* developed in [17,15,16] which is also the one used in Coq [4]. *ProPre* is devoted to the termination of recursively defined functions. The *Coq* and *ProPre* systems show the automated termination of a recursive function by first constructing a tree associated with the specification of the function which satisfies a notion of *terminal property* and then verifying that the process of constructing such a tree is formally correct. The search of such trees, from which it is also possible to extract decreasing measures [19,10] through the recursive call of the function, relies in particular on the structure of multi-sorted algebras and hence automated termination proofs in *Coq* and *ProPre* strongly depend on inductive principles. This means that *Coq* and *ProPre* can only deal with the termination proofs that are inductive.

Our aim is to extend the *Coq* and *ProPre* approaches to deal with automated non-inductive termination proofs. To do this, we introduce a new notion of terminal state property which has an algorithmic content that enables the method to be automated as an inductive one. From each tree that enjoys the terminal state property we associate an ordinal measure that couldn't be previously obtained from the *ProPre* system [19,10] and we show the decreasing property that ensures in this way the termination of the recursive function. As a consequence, the technique allows inductive methods to go further in the proof search when *natural* structural orderings are not enough to achieve the proof.

The paper is divided as follows: In Section 2, we set out the formal machinery. In Section 3, we introduce *ProPre* notion of terminal state property and our own extension of it. We show that our extension strictly includes the *ProPre* notion and establish in Theorem 1 that if a distributing tree $\mathcal{A}$ has the terminal state property in the system *ProPre*, then $\mathcal{A}$ has the new terminal state property and that the opposite does not hold. In Section 4, we explain how it is possible to define ordinal measures against trees of functions where if the ordinal measure decreases in the recursive call of the function, then this function terminates. We recall the ramified measures that come from the analysis of the *ProPre* system and we give new measures which will help in establishing terminations of functions where the proofs of terminations are non-inductive. Our main theorem of this section (Theorem 2) establishes that our new notion of terminal state and our extended notion of measures, enable us to establish the termination of functions (inductive and non inductive).

2 Preliminaries

2.1 Constructor Systems

In this paper we deal with constructor systems and more precisely with sorted constructor systems. The following standard definitions are needed.

Definition 2.1. We assume a set $\mathcal{F}$ of *function symbols*, called signature, and a set S of *sorts.* To each function $f \in \mathcal{F}$ we associate a natural number n that denotes its *arity* and a *type* $s_1, \ldots, s_n \to s$ with $s, s_1, \ldots, s_n \in S$. A function is called constant if its arity is 0.
We assume that the set of functions $\mathcal{F}$ is divided in two disjoint sets $\mathcal{F}_c$ and $\mathcal{F}_d$. Functions in $\mathcal{F}_c$ (which also include the constants) are called *constructor symbols* or *constructors* and those in $\mathcal{F}_d$ are called *defined symbols* or *defined functions.*

Definition 2.2. Let $\mathcal{X}$ be a set of *variables* disjoint from $\mathcal{F}$. We assume that each variable of $\mathcal{X}$ has a unique sort and that for each sort s there is a countable number of variables in $\mathcal{X}$ of sort s. If s is a sort, F and X are respectively sets included in $\mathcal{F}_c \cup \mathcal{F}_d$ and $\mathcal{X}$, then $\mathcal{T}(F, X)_s$ is the smallest set such that:

1. every element of X of sort s is a term of sort s,
2. if $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ respectively, and if f is a function of type $s_1, \ldots, s_n \to s$, then $f(t_1, \ldots, t_n)$ is a term of sort s.

If X is empty, we denote $\mathcal{T}(F, X)_s$ by $\mathcal{T}(F)_s$ whose elements are called ground terms. If the arity of c is 0, the *constant term* $c()$ is also denoted c. $\mathcal{V}ar(t)$ denotes the set of variables that occur in the term t, and $\mathcal{P}os(t)$ is the set of positions of t. If s and t are terms and q is a position of t, then the term $t[s]_q$ is the term t in which the term s is now at position q.

Definition 2.3. A *(sorted) equation* is a pair $(l, r)_s$ of terms l and r of a sort s, which is also called *rewrite rule* and written $l \to r$. A set of (sorted) equations is *non overlapping* iff no left-hand sides unify each other.

Definition 2.4. A *specification* or *constructor system* $\mathcal{E}$ of a function f : $s_1, \ldots, s_n \to s$ in $\mathcal{F}_d$ is a non overlapping set of left-linear equations $\{(e_1, e'_1)_s, \ldots, (e_p, e'_p)_s\}$ such that for all $1 \leq i \leq p$, e_i is of the form $f(t_1, \ldots, t_n)$ with $t_j \in \mathcal{T}(\mathcal{F}_c, \mathcal{X})_{s_j}$, $j = 1, \ldots, n$, and $e'_i \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_d, \mathcal{X})_s$.

Definition 2.5. Let $\mathcal{E}$ be a specification of a function f with type $s_1, \ldots, s_n \to s$. A *recursive call* of f is a pair $(f(t_1, \ldots, t_n), f(u_1, \ldots, u_n))$ where $f(t_1, \ldots, t_n)$ is a left-hand side of an equation of $\mathcal{E}$ and $f(u_1, \ldots, u_n)$ is a subterm of the corresponding right-hand side.

2.2 The Term Distributing Trees

We give some ingredients that will be needed in the next sections. A term distributing tree of a specification is a tree whose root can be seen as an uplet of distinct variables, each node matches its children and each leaf corresponds to a left-hand side of an equation. More precisely we have:

Definition 2.6. Let $\mathcal{E}$ be a specification of a function $f : s_1, \ldots, s_n \to s$. $\mathcal{A}$ is a *term distributing tree* for $\mathcal{E}$ iff it is a tree such that:

1. its root is of the form $f(x_1, \ldots, x_n)$ where x_i is a variable of sort s_i, $i \leq n$,
2. each left-hand side of an equation of $\mathcal{E}$ is a leaf of $\mathcal{A}$ (up to variable renaming)
3. each node $f(t_1, \ldots, t_n)$ of $\mathcal{A}$ admits one variable x' of a sort s' such that the set of children of the node is (for $x'_1, \ldots, x'_r$ are not in $t_1, \ldots t_n$):
   $\{f(t_1, \ldots t_n)[C(x'_1, \ldots x'_r)/x'], C : s'_1, \ldots, s'_r \to s' \in \mathcal{F}_c\}$.

Notation 2.7. Let $\mathcal{A}$ be a term distributing tree. A branch $\mathcal{B}$ from the root θ_1 to a leaf θ_k is denoted by $(\theta_1, x'_1), \ldots, (\theta_{k-1}, x'_{k-1}), \theta_k$ where θ_1 is the root, θ_k is the leaf, and for each $i \leq k-1$, x'_i is the variable x' for the node θ_i in the third clause of Definition 2.6.

It can be easily seen, according to Definition 2.6, that we have the following:

Fact 2.8. *Let $\mathcal{E}$ be a specification of a function f of type $s_1, \ldots, s_n \to s$ and $\mathcal{A}$ be a term distributing tree for $\mathcal{E}$.*

1. *For each $(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}_c)_{s_1} * \ldots * \mathcal{T}(\mathcal{F}_c)_{s_n}$ there exists one and only one leaf θ of $\mathcal{A}$ and a ground constructor substitution ρ such that $\rho(\theta) = f(t_1, \ldots, t_n)$.*
2. *For every branch of $\mathcal{A}$ from the root to a leaf $(\theta_1, x_1), \ldots, (\theta_{k-1}, x_{k-1})$, θ_k and for all $i \leq j \leq k$, there exists a constructor substitution $\sigma_{j,i}$ such that $\sigma_{j,i}(\theta_i) = \theta_j$. The substitutions $\sigma_{j,i}$ may also be written as $\sigma_{\theta_j,\theta_i}$*

We introduce here a well-founded ordering relation on the terms.

Definition 2.9. Assume a function m on the terms ranging over natural number that is closed under substitutions, i.e. $m(u) > m(v)$ implies $m(\sigma(u)) > m(\sigma(v))$ for all ground substitution σ. Let $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})_s$ for a given sort s. We say that $u \sqsubset v$ iff u is linear and $m(u) < m(v)$ with $\mathcal{V}ar(u) \subseteq \mathcal{V}ar(v)$.

3 Generalizing the Coq Termination Procedure and the ProPre System

The analysis of the termination proofs using the `Recursive Definition` of the Coq assistant and the *ProPre* system shows that writing a formal proof in these systems can be regarded as the search of a term distributing tree that enjoys a terminal state property. That is to say, if a formal tree for a specification of a function can be built having a terminal state property then the function terminates. We define in this section a new terminal state property for a term distributing tree generalizing that of `Recursive Definition` procedure and *ProPre*. We first give some notations that will be used in the rest of the paper.

Notation 3.1. Let $\mathcal{A}$ be a term distributing tree for a specification. If t is the left-hand side of an equation, $b(t)$ will denote the branch in the term distributing tree that leads to the term t. If b is a branch, then L_b will denote the leaf of the branch b. Note that b and $b(t)$ may denote two distinct branches.
If a node θ matches a term u of a recursive call (t, u), then the substitution will be denoted by $\rho_{\theta,u}$.

As every function that terminates with the procedure of the Coq assistant also terminates with the *ProPre* system, we do not give here the property in the setting of the `Recursive Definition` but only for the extended version of the *ProPre* system. Note that it is actually devised in a different way from below in [16]. However it has been shown [11] that for each distributing tree defined in [16] that enjoys the terminal state property of [16] there is a corresponding term distributing tree of Definition 2.6 that has the following property (Definition 3.2) which is more convenient for our purpose.

Definition 3.2. Let $\mathcal{A}$ be a term distributing tree for a specification. We say that $\mathcal{A}$ has the *terminal state property* (*tsp*) if there is an application $\mu : \mathcal{A} \to \{0, 1\}$ on the nodes of $\mathcal{A}$ such that if L is a leaf, $\mu(L) = 0$, and for all recursive call (t, u), there is a node (θ, x) in the branch $b(t)$ with $\mu(\theta) = 1$ such that θ matches u with $\rho_{\theta,u}(x) \sqsubset \sigma_{L_{b(t)},\theta}(x)$ and for all ancestor (θ', x') of θ in $b(t)$ with $\mu(\theta') = 1$, we have $\rho_{\theta',u}(x') \sqsubseteq \sigma_{L_{b(t)},\theta'}(x')$.

As already mentioned, if a specification $\mathcal{E}$ of a function admits a term distributing tree that has the terminal state property, then $\mathcal{E}$ is terminating [17].
The rest of this section is devoted to define a new terminal state property generalizing the previous one. We first need to introduce fresh variables as follows. For each position q and sort s, we will assume there is a new variable of sort s indexed by q and distinct from those of $\mathcal{X}$.

Definition 3.3. Let t be a term and q be a position. The term $[\![t]\!]_q$ is defined as follows: $[\![x]\!]_q = x$ if x is a variable, $[\![C(t_1, \dots, t_n)]\!]_q = C([\![t_1]\!]_{q\cdot 1}, \dots, [\![t_n]\!]_{q\cdot n})$ if $C \in F_c$, and $[\![g(t_1, \dots, t_n)]\!]_q = x_q$ if $g \in F_d$.

For a term $u = g(u_1, \dots, u_n)$ and a substitution φ, $g(\varphi[\![u]\!])$ will denote the term $g(\varphi([\![u]\!]_1), \dots, \varphi([\![u]\!]_n))$.

We introduce the following relations: For u, v in $\mathcal{T}(\mathcal{F}_c, \mathcal{X})_s$, we will say that $u \trianglerighteq v$ if $u \not\sqsubset v$ with $\neg$(b) or $\neg$(c) or $m(v) < m(u)$ in Definition 2.9; and we will say that $u \preccurlyeq v$ if $u \not\sqsubset v$ with (b) and (c) and $m(u) = m(v)$. We will use the so-called size measure $|\ .\ |_\#$ for the mentioned measure m. In the following definitions of the section we will consider a function $f : s_1, \dots, s_n \to s$, a specification or a split specification $\mathcal{E}$, and a term distributing tree $\mathcal{A}$ of $\mathcal{E}$.

Definition 3.4. For each node θ, C_θ will denote $\{b \in \mathcal{A}, \theta \in b\}$ and $\mathcal{R}_\theta$ the set of recursive call (t, u) such that $b(t) \in C_\theta$. If (t, u) is a recursive call, then $\mathcal{M}_\mathcal{A}(u) = \{b \in \mathcal{A}, \exists \varphi, \varphi'$ such that $f(\varphi[\![u]\!]) = \varphi'(f(L_b))\}$ and $\mathcal{Q}_\mathcal{A}(t, u) = \{\theta \in b(t), \exists \sigma, \sigma(f(\theta)) = u\}$.

Note that the set $\mathcal{Q}_{\mathcal{A}}(t,u)$ is not empty since the root node belongs to $\mathcal{Q}_{\mathcal{A}}(t,u)$. Let b be a branch and two nodes $\theta, \theta' \in b$, we say that $\theta < \theta'$ if θ is *closer* than θ' to the root (i.e. if θ is an ancestor of θ'). So we can write $\mathcal{N}_{\mathcal{A}}(t,u) = \max \mathcal{Q}_{\mathcal{A}}(t,u)$.

For each node θ of $\mathcal{A}$ we assume an associated subset $\mathcal{G}_\theta$ of $\mathcal{R}_\theta$ which will be made explicit in Definition 3.7. Notice that the definitions below should be given simultaneously but are introduced separately to ease the readability.

Definition 3.5. Let (θ, x) be a node of $\mathcal{A}$ and $\mathcal{G}_\theta$ be a subset of $\mathcal{R}_\theta$. For each recursive call (t,u) of $\mathcal{G}_\theta$ such that $\theta \in \mathcal{Q}_{\mathcal{A}}(t,u)$, we assume that one of the two following cases below holds and we define $\xi^\theta_{(t,u)}$, as follows:

1. If $\rho_{\theta,u}(x) \sqsubset \sigma_{L_{b(t)},\theta}(x)$ or $\rho_{\theta,u}(x) \trianglerighteq \sigma_{L_{b(t)},\theta}(x)$, then $\xi^\theta_{(t,u)} = 1$,
2. If $\rho_{\theta,u}(x) \preccurlyeq \sigma_{L_{b(t)},\theta}(x)$, then $\xi^\theta_{(t,u)} = 0$.

The meaning of the above definition and the following one is to give decreasing criteria extending those of Definitions 2.9 and 3.2. It relies in particular on the *hierarchical structure* of the trees.

Definition 3.6. Let (θ, x) be a node of $\mathcal{A}$ and $\mathcal{G}_\theta$ be a subset of $\mathcal{R}_\theta$. For each recursive call (t,u) of $\mathcal{G}_\theta$ such that $\theta \in \mathcal{Q}_{\mathcal{A}}(t,u)$ and for each branch $b \in \mathcal{C}_\theta$, we will define $\eta^\theta_{(t,u),b}$ in the following way:

1. First take all (t,u) such that $\rho_{\theta,u}(x) \trianglerighteq \sigma_{L_{b(t)},\theta}(x)$, and let for all $b \in \mathcal{C}_\theta$:
$$\eta^\theta_{(t,u),b} = \begin{cases} 0 & \text{if } b \in \mathcal{M}_{\mathcal{A}}(u), \\ 1 & \text{if not.} \end{cases}$$
2. Next, consider each (t,u) in $\mathcal{G}_\theta$ such that there is a (t',u') with $\eta^\theta_{(t',u'),b(t)} = 0$, and for which no $\eta^\theta_{(t,u),b'}$ is already defined for any $b' \in \mathcal{C}_\theta$. Then also take
$$\eta^\theta_{(t,u),b} = \begin{cases} 0 & \text{if } b \in \mathcal{M}_{\mathcal{A}}(u), \\ 1 & \text{if not.} \end{cases}$$
3. Finally if item 2 cannot be applied, put $\eta^\theta_{(t,u),b} = 1$ for each $b \in \mathcal{C}_\theta$.

Notice that the cases 1 and 2 are made distinct in the above definition as the value $\xi^\theta_{(t,u)}$ is algorithmically defined; namely case 1 is the initial case.

We define, for each node θ of $\mathcal{A}$ and each left-hand side t of an equation where θ with $b(t) \in C_\theta$, $\eta^\theta_t = \prod_{\substack{(t',u') \in \mathcal{G}_\theta \\ \theta \in \mathcal{Q}_{\mathcal{A}}(t',u')}} \eta^\theta_{(t',u'),b(t)}$ if $\mathcal{G}_\theta \neq \emptyset$, and $\eta^\theta_t = 0$ if not.

We now explicit the subset $\mathcal{G}_\theta$ of $\mathcal{R}_\theta$ for a node θ. The following states whether from each node, a recursive call can be eliminated from a set of recursive calls:

Definition 3.7. Let θ_1 be the root of the recursive distributing tree $\mathcal{A}$. We first put $\mathcal{G}_{\theta_1} = \mathcal{R}_{\theta_1}$. Now assume that $\mathcal{G}_\theta$ is defined for a node θ of $\mathcal{A}$ and let θ' be a child of θ with θ' in $\mathcal{A}$. The set $\mathcal{G}_{\theta'}$ is then defined as follows: $(t,u) \in \mathcal{G}_{\theta'}$ iff $(t,u) \in \mathcal{R}_{\theta'} \cap \mathcal{G}_\theta$ and $(\xi^\theta_{(t,u)}, \eta^\theta_t) \neq (1,1)$.

Now, we define F which is a necessary condition for the termination statement.

Definition 3.8. Let θ be a node of associated tree $\mathcal{A}$ of the recursive distributing tree $\mathcal{A}$ distinct from a leaf. We put $F(\theta) = 0$ if there is a child θ' of θ and (t, u) in $\mathcal{G}_{\theta'}$ such that $\theta > \mathcal{N}_{\mathcal{A}}(t, u)$; and we put $F(\theta) = 1$ if not.

Now the new terminal state property can be defined below.

Definition 3.9. The recursive distributing tree $\mathcal{A}$ is said to have the new terminal state property if for each node θ of $\mathcal{A}$ distinct from a leaf we have $F(\theta) = 1$ and for each branch b there is node θ' in b such that $\mathcal{G}_{\theta'} = \emptyset$.

We now come to Theorem 1 that states the above definition of the new terminal state property strictly includes the *ProPre* notion of terminal state property.

Theorem 1. *Let $\mathcal{E}$ be a specification of a function with a distributing tree $\mathcal{A}$. If $\mathcal{A}$ has the terminal state property in the system ProPre, then $\mathcal{A}$ has the new terminal state property. The opposite does not hold.*

A crucial point is of course to make sure that the new terminal state property leads a function to terminate. We prove this result in the next section by showing the existence of measures decreasing through the recursive calls of the functions. Note that there exist decreasing measures coming from the formal termination proofs in Coq or in *ProPre*. But in contrast with these measures, the new one, with the new terminal state property, will allow one to prove the termination functions that usually cannot be done with inductive methods.

4 Dealing with a Non Inductive Method

A close notion to term distributing trees of a specification that has the terminal state property is the ramified measures. The measures coming from *Coq* or *ProPre* characterize in some sense the induction proofs made in the systems. We recall the ramified measures and explain why we need to introduce other measures to deal with termination that usually cannot be proven with inductive methods. Among these measures, a particular class is defined that is related to term distributing trees enjoying the new terminal state property and we show that they have the decreasing property. This, therefore, implies that the corresponding functions terminate. As a consequence, this provides a method of reasoning about termination of recursive functions where the underlying proofs rely on non-inductive as well as inductive axioms.

4.1 The Ramified Measures and the *ProPre* System

Definition 4.1. Let $\mathcal{A}$ be a tree and θ a node of $\mathcal{A}$. The *height* of θ in $\mathcal{A}$, denoted by $\mathcal{H}(\theta, \mathcal{A})$, is the height of the subtree of $\mathcal{A}$ whose root is θ minus one.

For a term distributing tree $\mathcal{A}$, we assume that for each node θ_i different from a leaf there is an application m_i that maps on natural numbers. The general form of ordinal measures introduced in [19] is given by the following

Definition 4.2. Let $\mathcal{E}$ be a specification of a function $f : s_1, \dots, s_n \to s$, $\mathcal{A}$ be a term distributing tree for a specification of $\mathcal{E}$ and ω be the least infinite ordinal. The ramified measure $\Omega_{\mathcal{A}} : \mathcal{T}(F_c)_{s_1} * \dots * \mathcal{T}(F_c)_{s_n} \to \omega^{\omega}$ is defined by: Let $t = (t_1, \dots, t_n)$ be an element of the domain and θ be the leaf of $\mathcal{A}$ such that there is a substitution ρ with $\rho(\theta) = f(t)$ (Fact 2.8). Let $\mathcal{B}$ be the branch $(\theta_1, x_1), \dots, (\theta_{k-1}, x_{k-1}), \theta$ of $\mathcal{A}$ from the root to θ, let $\sigma_{r,s}$ be the substitutions of Fact 2.8 and for each θ_i the associated application m_i, $i \le k-1$. Then

$$\Omega_{\mathcal{A}}(t) = \sum_{i=1}^{k-1} \omega^{\mathcal{H}(\theta_i, \mathcal{A})} * m_i(\rho(\sigma_{k,i}(x_i))) \ .$$

The ramified measures can be illustrated by Figures 1 and 4. An interesting subclass of the above measures is the class of R-*measures.* It has been shown that to each formal termination proof of recursive functions made with the `Recursive Definition` procedure of the Coq assistant, there is a distributing tree that has the terminal state property implying the decreasing property of the R-measure associated to the distributing tree [19]. This class of measures could be enlarged with I-*measures* [10] that can be related to a more efficient version of *ProPre* [16].

The functions m_i that occur in the definition of these measures belonging to the class of Definition 4.2 are directly supplied from formal proofs made in the system. This can be illustrated by Figures 2 and 3, where m is the parameterized function of Definition 2.9.

Definition 4.3. The *recursive length* lg of a term t of sort s is defined by:

1. if t is a constant or a variable, then $lg(t) = 1$,
2. if $t = C(t_1, \dots, t_n)$ with $C : s_1, \dots, s_n \to s$ then $lg(t) = 1 + \sum_{s_j = s} lg(t_j)$.

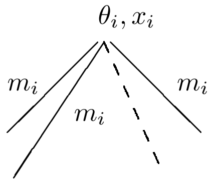


Fig.1. Ramified measures

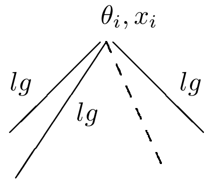


Fig.2. R-measures

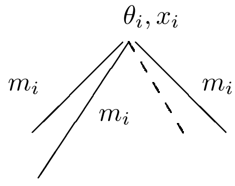


Fig.3. I-measures, $m_i \in \{m, \tilde{0}\}$

4.2 Extended Ordinal Measures

We motivate here the definition of new ordinals illustrated with some examples.

It is well known that the structural ordering is the most used among the well-founded orderings on natural numbers. As it is claimed in [20], other well-orderings on natural number are difficult to find automatically. As a simple illustration the constant function with value 0 is given in [20] that can be defined with the following specification $\mathcal{E}_1$.

$$f(0) \to f(s(0)); \quad f(s(0)) \to f(s(s(0)); \quad f(s(s(x)) \to 0. \qquad (1)$$

Though a well-founded ordering is of course easy to find by a human in this case, it is however difficult to obtain one in an automated way since it is a non-simply terminating function and not suited to inductive methods. Note that proving at the same time the correctness of the specification (i.e. $f(x) = 0$) and the termination seems not really relevant here as this is usually done with an ordering. Moreover the specification of the *quot* function given in this paper clearly shows that the correctness cannot be helpful in that case.

Note that there is no term distributing tree of $\mathcal{E}_1$ which has the terminal state property, but there is one that satisfies the new terminal state property.

The following example $\mathcal{E}_2$ of the function *evenodd* : $nat, nat \rightarrow Bool$ is borrowed from [1]. As mentioned in [1] the modifications of mutually recursive functions to obtain a function without mutual recursion leads to such specifications as that of *evenodd* below. We assume that *not* is already defined.

$$\begin{aligned} &evenodd(x, 0) \rightarrow not(evenodd(x, s(0))) \\ &evenodd(0, s(0)) \rightarrow false \\ &evenodd(s(x), s(0)) \rightarrow evenodd(x, 0). \end{aligned} \tag{2}$$

The second argument is used as a *flag* that enables *evenodd* to compute either the *even* function or the *odd* function. This function, which is a non simply terminating function, cannot be proven with usual inductive methods since precisely there is no *natural* orderings that can be used.

Consider the next example of specification of the function *quot* : $nat, nat, nat \rightarrow nat$, borrowed from T. Kolbe [13] and that can be found in [2].

The value of $quot(x, y, z)$ corresponds to $1 + \lfloor \frac{x-y}{z} \rfloor$ when $z \neq 0$ and $y \leq x$, that is to say $quot(x, y, y)$ computes $\lfloor \frac{x}{y} \rfloor$.

$$\begin{aligned} &quot(0, s(y), s(z)) \rightarrow 0 \\ &quot(s(x), s(y), z) \rightarrow quot(x, y, z) \\ &quot(x, 0, s(z)) \rightarrow s(quot(x, s(z), s(z))). \end{aligned} \tag{3}$$

The last rule shows that the specification is not simply terminating. The same rule also shows that the termination cannot be proven by usual inductions proofs.

It turns out that specification functions such as (1), the *evenodd* function (2) or the *quot* function (3) cannot be proven by the system *ProPre* and no R-measures neither I-measures [19,10] have the decreasing property for any of these specifications. However, the following ordinal function

$\Omega(u, 0) = \omega * |u|_{\#} + 1, \quad \Omega(u, s(v)) = \omega * |u|_{\#},$

where $| \cdot |_{\#}$ is the *size* function, i.e. $|0|_{\#} = 1$, $|s(u)|_{\#} = 1 + |u|_{\#}$, reflects the specification of *evenodd* in the sense that it decreases in the recursive call of the function. There is also an ordinal measure below that has the decreasing property for the specification of the *quot* function

$\Omega(u, s(v), w) = \omega * |u|_{\#}, \quad \Omega(u, 0, w) = \omega * |u|_{\#} + 1.$

It would be possible to find a decreasing measure in the class of Definition 4.2 for (1), (2) or (3), but the choice of the m_i is difficult to obtain in an automated way. In particular we want to have m_i functions that are as simple as possible, such as for instance the size functions that are found in the above ordinal. Furthermore we would like to relate decreasing measures to term distributing trees that satisfy the new notion of terminal state property generalizing those of Coq and *ProPre*. It turns out that such suitable measures actually belong to the extended ordinal measures defined below. We first introduce the following

Definition 4.4. Let $\mathcal{E}$ be a specification of a function $f : s_1, \dots , s_n \to s$ such that there exists a term distributing tree $\mathcal{A}$ for $\mathcal{E}$. For each node θ_i of $\mathcal{A}$, we will assume that there are associated applications $m_{i,1}, \dots , m_{i,j_i}$ that map on natural numbers whose number is equal to the number of the sub-branches starting from the node θ_i. Note that this number may be distinct from the number of the children of the node. These applications will be called *node measures*. If θ is a leaf of a branch where θ_i appears, we will also use $m_{\theta_i,\theta}$ to make explicit one of the node measures of θ_i when necessary.

Definition 4.5. Let $\mathcal{E}$ be a specification of a function $f : s_1, \dots , s_n \to s$ such that there exists a term distributing tree $\mathcal{A}$ for $\mathcal{E}$. The extended measure $\Omega_{\mathcal{A}} : \mathcal{T}(F_c)_{s_1} * \dots * \mathcal{T}(F_c)_{s_n} \to \omega^{\omega}$, is defined as follows:
Let $t = (t_1, \dots , t_n)$ be an element of the domain and θ be the leaf of $\mathcal{A}$ such that there is a substitution ρ with $\rho(\theta) = f(t)$ (Fact 2.8). Let $\mathcal{B}$ be the branch $(\theta_1, x_1), \dots , (\theta_{k-1}, x_{k-1}), \theta$ of $\mathcal{A}$ from the root to θ, let $\sigma_{r,s}$ be the substitutions of Fact 2.8. Then

$$\Omega_{\mathcal{A}}(t) = \sum_{i=1}^{k-1} \omega^{\mathcal{H}(\theta_i,\mathcal{A})} * m_{\theta_i,\theta}(\rho(\sigma_{k,i}(x_i))) \ .$$

An extended measure can be illustrated by Figures 5 and 6.

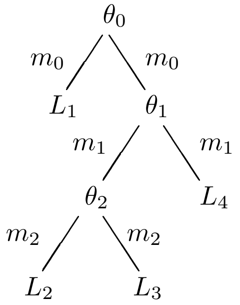


Fig.4. Term distributing with ramified measure

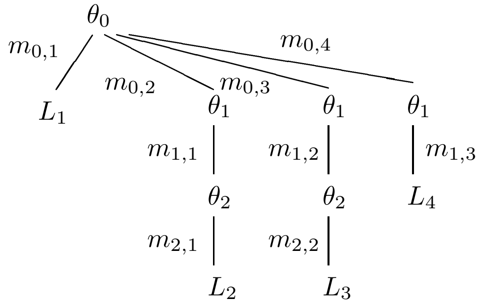


Fig.5. Term distributing with extended measure

For instance the specification $\mathcal{E}_1$ with the equations (1) in Section 4.2 admits a term distributing tree with an extended measure defined as follows
$\Omega_{\mathcal{A}}(0) = \omega * |0|_{\#}, \quad \Omega_{\mathcal{A}}(s(0)) = |0|_{\#}, \quad \Omega_{\mathcal{A}}(s(s(u))) = 0.$
This measure has obviously the decreasing property in the recursive calls of $\mathcal{E}_1$.

One may wonder whether the automation of decreasing measures belonging to Definition 4.5 is possible since we have to take account of the $m_{i,j}$ applications. We will show that it will be enough to consider a subclass of measures, called *hole-measures*, generalizing R- and I-measures. These measures will be associated to term distributing trees enjoying the new terminal state property. We will show that they have the decreasing property and that functions which admit a term distributing tree with the new terminal state property, are therefore terminating.

4.3 The Hole-Measures

Definition 4.6. Let $\mathcal{E}$ be a specification of a function $f : s_1, \dots, s_n \to s$ such that there exists a term distributing tree $\mathcal{A}$ for $\mathcal{E}$. The *hole measure* $\Omega_{\mathcal{A}} : \mathcal{T}(F_c)_{s_1} * \dots * \mathcal{T}(F_c)_{s_n} \to \omega^{\omega}$, is defined as follows:
Let $t = (t_1, \dots, t_n)$ be an element of the domain and θ be the leaf of $\mathcal{A}$ such that there is a substitution ρ with $\rho(\theta) = f(t)$ (Fact 2.8). Let $\mathcal{B}$ be the branch $(\theta_1, x_1), \dots, (\theta_{k-1}, x_{k-1}), \theta$ of $\mathcal{A}$ from the root to θ, let $\sigma_{r,s}$ be the substitutions of Fact 2.8. Then

$$\Omega_{\mathcal{A}}(t) = \sum_{i=1}^{k-1} \omega^{\mathcal{H}(\theta_i, \mathcal{A})} * (\eta_t^{\theta_i} * |(\rho(\sigma_{k,i}(x_i)))|_{\#}) .$$

That is to say $m_{\theta_i,\theta} = \eta_t^{\theta_i} * |\ .\ |_{\#}$.

Note that, due to the relation between the leaf θ and the term t, $\eta_t^{\theta_i} * |\ .\ |_{\#}$ depends both on θ_i and θ in the above definition.

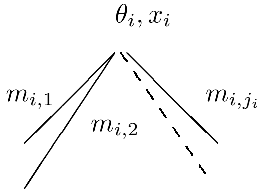
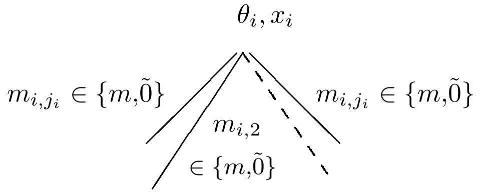


Fig.6. Extended measures


θ_i, x_i
$m_{i,j_i} \in \{m,\tilde{0}\}$
$m_{i,2}$
$\in \{m,\tilde{0}\}$
$m_{i,j_i} \in \{m,\tilde{0}\}$
Fig.7. Hole-measures with $m_{i,j} \in \{m,\tilde{0}\}$

Now we come to the main theorem of this paper. It states that our new notion of new terminal state property and our extended notion of measures enable us to establish the inductive and non inductive termination of functions.

Theorem 2. *Let $\mathcal{E}$ be a specification of a function $f : s_1, \dots, s_n \to s$ and $\mathcal{A}$ be a distributing tree $\mathcal{A}$ for $\mathcal{E}$ having the new terminal state property. The associated measure $\Omega_{\mathcal{A}}$ satisfies the decreasing property. I.e., for each recursive call $(f(t_1, \dots, t_n), f(u_1, \dots, u_n))$ of $\mathcal{E}$ and ground constructor substitution φ we have: $\Omega_{\mathcal{A}}(\varphi(t_1), \dots, \varphi(t_n)) > \Omega_{\mathcal{A}}(\varphi(\llbracket u_1 \rrbracket_1), \dots, \varphi(\llbracket u_n \rrbracket_n))$.*

5 Conclusion

In this paper we have proposed a method that extends the automation of the proofs of termination of recursive functions used in *ProPre* and Coq. Whereas Coq and *ProPre* could only deal with the automation of inductive proofs, the method allows the automation of a larger class of recursive functions because non structural orderings can be handled by the method. The method is also a good vehicle for extending the automation of termination proofs of recursive functions to deal with issues not yet incorporated in theorem provers.

References

1. T. Arts and J. Giesl. *Automatically proving termination where simplification orderings fail* in Proceeding TAPSOFT'97, LNCS, volume 1214, 261-272.
2. T. Arts and J. Giesl. *Proving innermost normalisation automatically* in Proceeding RTA'97, LNCS, volume 1232, 157-171.
3. A. Ben Cherifa and P. Lescanne. *Termination of rewriting systems by polynomial interpretations and its implementation.* Science of Computer Programming 9(2), 137-159, 1987.
4. C. Cornes *et al.*. The Coq proof assistant reference manual version 5.10. *Technical Report 077*, INRIA, 1995.
5. N. Dershowitz. Termination of rewriting. *Theoretical Computer Science 17*, 279-301, 1982.
6. N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science* 142(2), 179-207, 1995.
7. J. Dick, J. Kalmus and U. Martin. Automating the Knuth Bendix ordering. *Acta Informatica 28*, 95-119, 1990.
8. T. Genet and I. Gnaedig. Termination proofs using gpo ordering constraints. *TAPSOFT*, LNCS 1214, 249-260, 1997.
9. J. Giesl. *Generating polynomial orderings for termination proofs.* Proceedings of the 6th International Conference on Rewriting Techniques and Application, Kaiserlautern, LNCS, volume 914, 1995.
10. F. Kamareddine and F. Monin. On formalised proofs of termination of recursive functions. In *Proc. International Conference on Principles and Practice of Declarative Programming*, Paris, France, 1999.
11. F. Kamareddine and F. Monin. Induction Lemmas for Termination of Recursive Functions in a Typed System *Proc. submitted.*
12. D. E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational problems in abstract algebra*, Pergamon Press, 263-297, 1970.
13. T. Kolbe. Challenge problems for automated termination proofs of term rewriting systems. Technical Report IBN96-42, Technische Hochshule Darmstadt, Alexanderstr. 10, 64283 Darmstadt, Germany, 1996.
14. P. Lescanne. On the recursive decomposition ordering with lexicographical status and other related orderings. *Journal of Automated Reasoning* 6(1) 39-49, 1990.
15. P. Manoury. *A User's friendly syntax to define recursive functions as typed lambda-terms.* Proceedings of Type for Proofs and Programs TYPES'94, LNCS, volume 996, 1994.

16. P. Manoury and M. Simonot. *Des preuves de totalité de fonctions comme synthèse de programmes*. PhD thesis, University Paris 7, 1992.
17. P. Manoury and M. Simonot. Automatizing termination proofs of recursively defined functions. *Theoretical Computer Science* 135(2) 319-343, 1994.
18. A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science 175*, 127-158, 1997.
19. F. Monin and M. Simonot. An ordinal measure based procedure for termination of functions. To appear in *Theoretical Computer Science*.
20. M. Parigot. Recursive programming with proofs. *Theoretical Computer Science* 94(2) 335-356, 1992.
21. J. Steinbach. Generating polynomial orderings. *Information Processing Letters 49*, 85-93, 1994.

Asynchronous Links in the PBC and M-Nets

Hanna Klaudel and Franck Pommereau

Université Paris XII, 61 Avenue du Général de Gaulle
94 010 Créteil, France
{pommereau,klaudel}@univ-paris12.fr

Abstract. This paper aims at introducing an extension of M-nets, a fully compositional class of high-level Petri nets, and of its low-level counter part, Petri Boxes Calculus (PBC). We introduce a new operator with nice algebraic properties which allows to express asynchronous communications in a simple and flexible way. With this extension, asynchronous communications become at least as simple to express as (existing) synchronous ones. Finally, we show how this extension can be used in order to specify systems with timing constraints.

Keywords: Petri Net, Petri Box Calculus, M-Net, Semantics, Timed Specification.

1 Introduction

M-nets, constructed at the top of the algebra of Petri boxes [4,3,13], are a fruitful class of high-level Petri nets which nicely unfold into low-level nets and thus allow to represent large (possibly infinite) systems in a clear and compact way. They are widely used now as a semantic domain for concurrent system specifications, programming languages, or protocols, *cf.* [6,7,11,1,14,2,12]. The most original aspect of M-nets with respect to other high-level net classes is their full compositionality, thanks to their interfaces, and a set of various net operations defined for them. Their interest is augmented by the ability to use in practice an associated tool, PEP [5], which also offers various implemented verification and analysis methods.

This paper defines a possibility to express asynchronous communication links at the M-net and PBC algebras level by introducing a new operator (**tie**). This extension mainly concerns the labelling of transitions, and now, in addition to usual synchronous communications (with the synchronisation operator **sy**), transitions may also export or import data through asynchronous links. It turns out that the **tie** operator has nice algebraic properties: idempotence, commutativity with itself and with the synchronisation, and also coherence (*i.e.*, commutativity) with respect to the unfolding. These properties allow to preserve the full compositionnality of the model.

As an application, we present a modeling of discrete time constraints in the M-nets. This allows to specify time-constrained systems and to analyse their unfoldings with existing tools (*e.g.*, PEP). We use a high-level featured clock which handles *counting requests* for the rest of the system. Asynchronous links are used to perform the necessary communications between the actions related to each request.

The next three sections briefly introduce M-nets and PBC, including the basis for our extension. Then, sections 5 and 6 give the definitions and the algebraic properties of the **tie** operation. Section 7 is devoted to the application of asynchronous links to discrete time modelling.

2 Basic Definitions

Let E be a set. A *multiset* over E is a function $\mu : E \to \mathbb{N}$; μ is finite if $\{e \in E \mid \mu(e) > 0\}$ is finite. We denote by $\mathcal{M}_f(E)$ the set of finite multi-sets over E, by $\oplus$ and $\ominus$ the sum and difference of multi-sets, respectively. $\odot$ is used to relate an element of E to the number of its occurences in a multi-set over E; in particular, a multi-set μ over E may be written as $\bigoplus_{e \in E} \mu(e) \odot e$, or in extended set notation, *e.g.*, $\{a, a, b\}$ for $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(e) = 0$ for all $e \in E \setminus \{a, b\}$. We may also write $e \in \mu$ for $\mu(e) > 0$.

Let *Val* and *Var* be fixed but suitably large disjoint sets of *values* and *variables*, respectively. The set of all well-formed *predicates* built from the sets *Val*, *Var* and a suitable set of operators is denoted by Pr.

We assume the existence of fixed disjoint sets A_h of *high-level action symbols* (for transition synchronous communications) and B of *tie symbols* (for transition asynchronous links). We assume that each element $A \in \mathsf{A}_h$ has an arity $ar(A)$, and that there exists a bijection $\widehat{\ }$ on A_h (called *conjugation*), with $\widehat{\widehat{A}} = A$, $\widehat{A} \neq A$ and $ar(A) = ar(\widehat{A})$. We also assume that each element $b \in \mathsf{B}$ has a type $type(b) \subseteq Val$.

A *high-level action* is a construct $A(a_1, \ldots, a_{ar(A)})$ where $A \in \mathsf{A}_h$ (notice that we could have $\widehat{A}$ instead of A) and $a_i \in Val \cup Var$ $(1 \leq i \leq ar(A))$. A typical high-level action is, for instance, $A(a_1, a_2, 5)$ where $A \in \mathsf{A}_h$ and a_1, a_2 are variables. The set of all high-level actions is denoted by AX_h.

Similarly, a *low-level action* is a construct $A(v_1, \ldots, v_n) \in \mathsf{AX}_h$, where $v_i \in Val$ for all $i \in \{1, \ldots, n\}$. The set of all low-level actions is denoted by A_l. As for high-level case, we assume that A_l is provided with a bijection $\widehat{\ }$, with analogous constraints; moreover, we will write $\widehat{A}(v_1, \ldots, v_{ar(A)})$ instead of $\widehat{A(v_1, \ldots, v_{ar(A)})}$.

A *high-level link* over b is a construct $b^d(a)$, where $b \in \mathsf{B}$, $d \in \{+, -\}$ is a *link direction symbol*, and $a \in Val \cup Var$. The set of all high-level links is denoted by B_h and the set of all low-level links is denoted by $\mathsf{B}_l = \{b^d(v) \mid b \in \mathsf{B}, d \in \{+, -\}, v \in Val\}$.

The *deletion* is defined for a multi-set of links $\beta \in \mathcal{M}_f(\mathsf{B}_h)$ and a tie symbol $b \in \mathsf{B}$, as

$$\beta \,\mathbf{del}\, b = \beta \ominus \left(\bigoplus_{l \in \mathcal{M}_f(\{b^d(a) \mid d \in \{+,-\}, a \in Var \cup Val\})} l \right) ,$$

and analogously for low-level links.

A *binding* is a mapping $\sigma\colon \mathit{Var} \rightarrow \mathit{Val}$ and an *evaluation* of an entity η (which can be a variable, a vector or a (multi-)set of variables, a set of predicates, a (multi-)set of high-level actions, etc.) through σ is defined as usual and denoted by $\eta[\sigma]$. For instance, if $\sigma = (a_1 \mapsto 2, a_2 \mapsto 3)$, the evaluation of high-level action $A(a_1, a_2, 5)$ through σ is the low-level action $A(2, 3, 5)$. Similarly, the high-level link $b^+(a_1)$ evaluates through σ to the low-level link $b^+(2)$ and the predicate $a_1 = 2$ to *true*.

3 Petri Boxes and M-Nets

Petri Boxes are introduced in [4,3,6] as labeled place/transition Petri nets satisfying some constraints, in order to model concurrent systems and programming languages with full compositionality.

Definition 1. *A (low-level) labeled net is a quadruple $L = (S, T, W, \lambda)$, where S is a set of* places*, T is a set of* transitions*, such that $S \cap T = \emptyset$, $W : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ is a* weight function*, and λ is a function, called the* labeling *of L, such that:*

- *$\forall s \in S\colon \lambda(s) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$ gives the place status (entry, internal or exit, respectively);*
- *$\forall t \in T\colon \lambda(t) = \alpha(t).\beta(t)$ gives the transition label, where $\alpha(t) \in \mathcal{M}_f(\mathsf{A}_l)$ and $\beta(t) \in \mathcal{M}_f(\mathsf{B}_l)$.*

The behavior of such a net, starting from the *entry marking* (just one token in each e-place), is determined by the usual definitions for place/transition Petri nets.

M-nets are a mixture of colored net features and low-level labeled net ones. They can be viewed as abbreviations of the latter.

Definition 2. *An* M-net *is a triple (S, T, ι), where S and T are disjoint sets of* places *and* transitions*, and ι is an inscription function with domain $S \cup (S \times T) \cup (T \times S) \cup T$ such that:*

- *for every place $s \in S$, $\iota(s)$ is a pair $\lambda(s).\tau(s)$, where $\lambda(s) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$ is the* label *of s, and $\tau(s) \subseteq \mathit{Val}$, is the* type *of s;*
- *for every transition $t \in T$, $\iota(t) = \lambda(t).\gamma(t)$, with $\lambda(t) = \alpha(t).\beta(t)$, the* label *of t, where $\alpha(t) \in \mathcal{M}_f(\mathsf{AX}_h)$ is the* action label *and $\beta(t) \in \mathcal{M}_f(\mathsf{B}_h)$ is the* link label*; $\gamma(t)$, the* guard *of t, is a finite set of predicates from Pr;*
- *for every arc $(s, t) \in (S \times T) : \iota((s, t)) \in \mathcal{M}_f(\mathit{Val} \cup \mathit{Var})$ is a multi-set of variables or values (analogously for arcs $(t, s) \in (T \times S)$).*
  $\iota((s, t))$ will generally be abbreviated as $\iota(s, t)$.

A *marking* of an M-net (S, T, ι) is a mapping $M\colon S \rightarrow \mathcal{M}_f(\mathit{Val})$ which associates to each place $s \in S$ a multi-set of values from $\tau(s)$. In particular, we shall distinguish (like for low-level nets) the *entry marking*, where $M(s) = \tau(s)$ if $\lambda(s) = \mathsf{e}$ and the empty (multi-)set otherwise.

The transition rule specifies the circumstances under which a marking M' is reachable from a marking M. A transition t is *enabled* at a marking M if there is an *enabling binding* σ for variables in the inscription of t (making the guard true) and in arcs around t such that $\forall s \in S : \; \iota(s,t)[\sigma] \leq M(s)$, *i.e.*, there are enough tokens of each type to satisfy the required flow. The effect of an occurrence of t, under an enabling binding σ, is to remove tokens from its input places and to add tokens to its output places, according to the evaluation of arcs' annotations under σ.

The unfolding operation associates a labeled low-level net (see *e.g.* [4]) $\mathcal{U}(N)$ with every M-net N, as well as a marking $\mathcal{U}(M)$ of $\mathcal{U}(N)$ with every marking M of N.

Definition 3. *Let* $N = (S, T, \iota)$*; then* $\mathcal{U}(N) = (\mathcal{U}(S), \mathcal{U}(T), W, \lambda)$ *is defined as follows:*

- $\mathcal{U}(S) = \{s_v \mid s \in S \text{ and } v \in \tau(s)\}$,
  and for each $s_v \in \mathcal{U}(S) : \lambda(s_v) = \lambda(s)$;
- $\mathcal{U}(T) = \{t_\sigma \mid t \in T \text{ and } \sigma \text{ is an enabling binding of } t\}$,
  and for each $t_\sigma \in \mathcal{U}(T) : \lambda(t_\sigma) = \lambda(t)[\sigma]$;
- $W(s_v, t_\sigma) = \sum_{a \in \iota(s,t) \, \wedge \, a[\sigma]=v} \iota(s,t)(a)$, *and analogously for* $W(t_\sigma, s_v)$.

Let M be a marking of N. $\mathcal{U}(M)$ is defined as follows: for every place $s_v \in \mathcal{U}(S)$, $(\mathcal{U}(M))\,(s_v) = (M(s))\,(v)$. Thus, each elementary place $s_v \in \mathcal{U}(S)$ contains as many tokens as the number of occurrences of v in the marking $M(s)$.

4 Box and M-Net Algebras

The same operations are defined on both box and M-net levels. They can be divided in two categories: the control flow ones and the communication ones. The first group, which consists of sequential (;) and parallel (‖) compositions, choice (□) and iteration ([∗∗]), can be synthesized from refinement meta-operation [8,9] and they will not be concerned by our extension. The second group concerns the operations which are based on transition composition, and will be directly concerned here, so we introduce them with some details. Only low-level operations are defined formally while we give some intuition for the high-level (M-net) operations. We illustrate the low-level synchronization and restriction on an example and we refer to [6] for further illustrations.

A synchronization $L\,\mathbf{sy}\,A_l$, with $A_l \in \mathsf{A}_l$, adds transitions to the net L, and can be characterized as CCS-like synchronization, extended to multi-sets of actions. Intuitively, the synchronization operation of an M-net consists of a repetition of certain basic synchronizations. An example of such a basic synchronization over low-level action $A(2,3)$ of the (fragment of) net L is given in figure 1. Transitions t_1 and t_2 which contain actions $A(2,3)$ and $\widehat{A}(2,3)$ in their labels can be synchronized over $A(2,3)$ yielding a new transition (t_1, t_2). The repetition of such basic synchronizations over $A(2,3)$, for all matching pairs of transitions (containing $A(2,3)$ and $\widehat{A}(2,3)$), yields the synchronization of a net over $A(2,3)$.

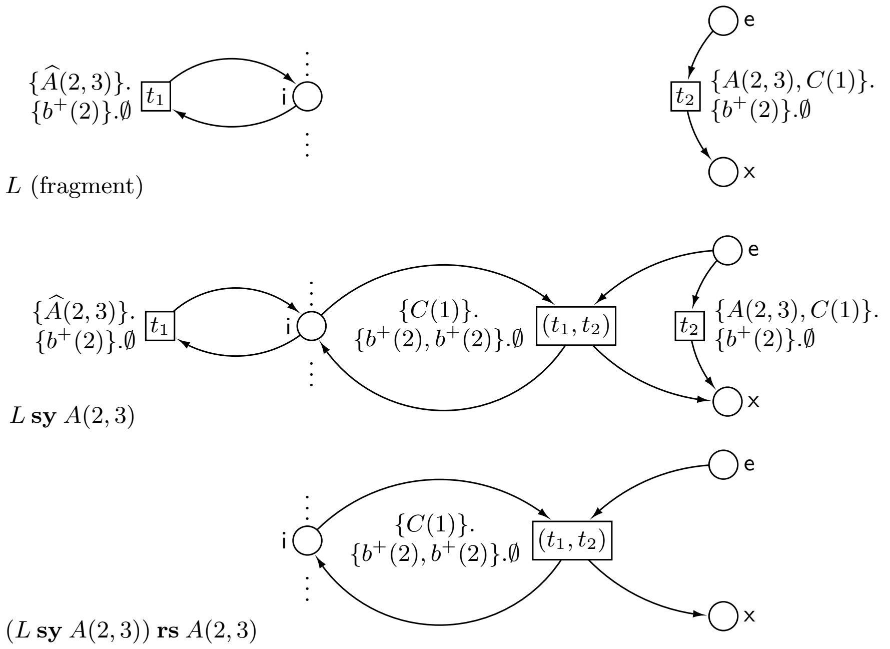


Fig. 1. Synchronization and restriction in PBC.

In M-nets, the actions $A(a_1, a_2)$ and $\widehat{A}(a'_1, a'_2)$ can synchronize through renaming and unification of their parameters.

Definition 4. *Let $L = (S, T, W, \lambda)$ be a low-level net and $A_l \in \mathsf{A}_l$ a low-level action. The synchronization L **sy** A_l, is defined as the smallest*[1] *low-level net $L' = (S', T', W', \lambda')$, satisfying:*

- *$S' = S$, $T' \supseteq T$, and $W'|_{(S\times T)\cup(T\times S)} = W$;*
- *if transitions t_1 and t_2 of L' are such that $A_l \in \alpha'(t_1)$ and $\widehat{A_l} \in \alpha'(t_2)$, then L' contains also a transition t with its adjacent arcs satisfying:*
  - *$\lambda'(t) = \Big(\alpha'(t_1) \oplus \alpha'(t_2) \ominus \{A_l, \widehat{A_l}\}\Big) . \Big(\beta'(t_1) \oplus \beta'(t_2)\Big)$,*
  - *$\forall s \in S'$: $W'(s,t) = W'(s,t_1) \oplus W'(s,t_2)$ and $W'(t,s) = W'(t_1,s) \oplus W'(t_2,s)$.*

The lowest part of figure 1 shows the restriction of L **sy** $A(2,3)$ over the action $A(2,3)$ which returns a net in which all transitions whose labels contain at least one action $A(2,3)$ or $\widehat{A}(2,3)$ are deleted (together with their surrounding arcs). The synchronization followed by the restriction is called *scoping*: $[\![a : L]\!] = (L\ \mathbf{sy}\ a)\ \mathbf{rs}\ a$, for an action a.

[1] with respect to the net inclusion, and up to renaming of variables.

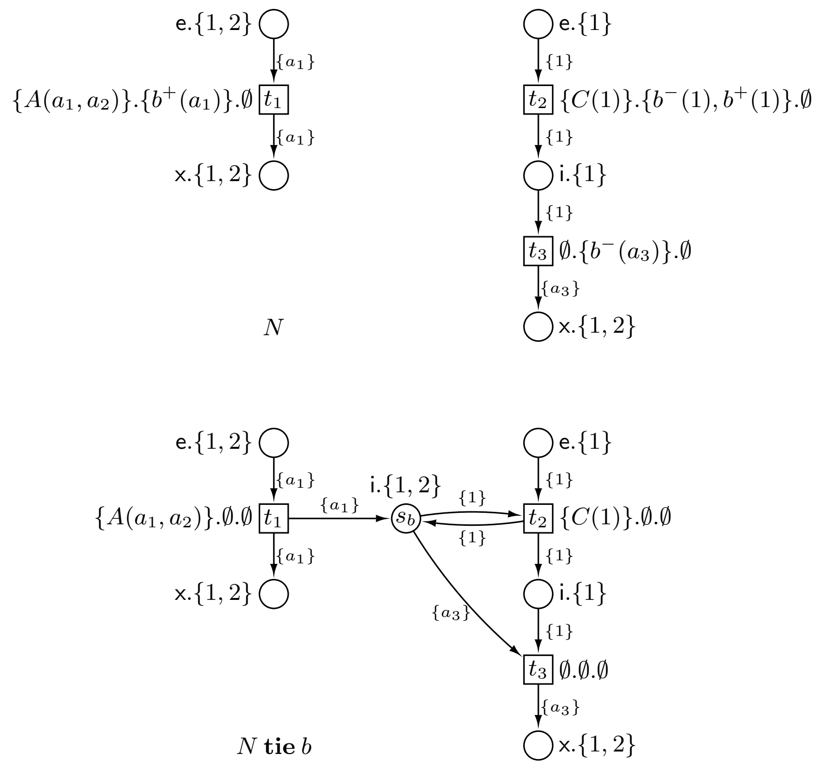


Fig. 2. High-level tie operation.

5 Asynchronous Link Operator: Tie

In this section, we introduce a new M-net algebra operator, **tie**, devoted to express asynchronous links between transitions. We give first an example in the high-level, and then, define it formally in both high and low-levels.

In figure 2, operator **tie** takes an M-net N and a tie symbol b (we assume that $type(b) = \{1, 2\}$), and gives an M-net N**tie**b which is like N but has an additional internal place s_b of the same type as b, and additional arcs between s_b and transitions which carry in their label (high-level) links over b. The inscriptions of these arcs are (multi-)sets of variables or values corresponding to links over b, and the labels of concerned transitions are as before minus all links over b. For instance, the arc from t_1 to s_b is inscribed by $\{a_1\}$ because there is a link $b^+(a_1)$ in the link label of t_1, which means that the variable a_1 has to be *exported* through b.

Definition 5. *Let* $N = (S, T, \iota)$ *be an M-net and* $b \in \mathsf{B}$ *a tie symbol.* N **tie** b *is an M-net* $N' = (S', T', \iota')$ *such that:*

– $S' = S \uplus \{s_b\}$, *and* $\forall s \in S' \colon \iota'(s) = \begin{cases} \mathsf{i}.\mathit{type}(b) & \text{if } s = s_b, \\ \iota(s) & \text{otherwise}; \end{cases}$
– $T' = T$ *and* $\forall t \in T' \colon \iota'(t) = \alpha(t).\big(\beta(t)\,\mathbf{del}\,b\big).\gamma(t)$, *if* $\iota(t) = \alpha(t).\beta(t).\gamma(t)$;
– $\forall s \in S'$ *and* $\forall t \in T'$ *we have:*
  - $\iota'(s,t) = \begin{cases} \displaystyle\bigoplus_{a \in \mathit{Var} \cup \mathit{Val}} \beta(t)(b^-(a)) \odot a & \text{if } s = s_b, \\ \iota(s,t) & \text{otherwise}, \end{cases}$
  - $\iota'(t,s) = \begin{cases} \displaystyle\bigoplus_{a \in \mathit{Var} \cup \mathit{Val}} \beta(t)(b^+(a)) \odot a & \text{if } s = s_b, \\ \iota(t,s) & \text{otherwise}. \end{cases}$

The tie operation in the low-level is defined similarly. In that case, a place is created for each value in *type*(b) and arcs are added accordingly.

Definition 6. *Let $L = (S, T, W, \lambda)$ be a low-level net and $b \in \mathsf{B}$ a tie symbol. $L\,\mathbf{tie}\,b$ is a low-level net $L' = (S', T', W', \lambda')$ such that:*

– $S' = S \uplus \{s_{b,v} \mid v \in \mathit{type}(b)\}$, *and* $\forall s \in S' \colon \lambda'(s) = \begin{cases} \lambda(s) & \text{if } s \in S, \\ \mathsf{i} & \text{otherwise}; \end{cases}$
– $T' = T$ *and* $\forall t \in T' \colon \lambda'(t) = \alpha(t).\big(\beta(t)\,\mathbf{del}\,b\big)$, *if* $\lambda(t) = \alpha(t).\beta(t)$;
– $\forall s \in S', \forall t \in T'$ *and* $\forall v \in \mathit{type}(b)$ *we have:*
  - $W'(s,t) = \begin{cases} \displaystyle\sum_{v \in \mathit{Val}} \beta(t)(b^-(v)) & \text{if } s = s_{b,v} \in S' \setminus S, \\ W(s,t) & \text{otherwise}, \end{cases}$
  - $W'(t,s) = \begin{cases} \displaystyle\sum_{v \in \mathit{Val}} \beta(t)(b^+(v)) & \text{if } s = s_{b,v} \in S' \setminus S, \\ W(t,s) & \text{otherwise}. \end{cases}$

6 Properties

Theorem 1. *Let L be a low-level net, $A_l \in \mathsf{A}_l$ and b_1, $b_2 \in \mathsf{B}$. Then:*

1. $(L\,\mathbf{tie}\,b_1)\,\mathbf{tie}\,b_1 = L\,\mathbf{tie}\,b_1$ *(idempotence)*
2. $(L\,\mathbf{tie}\,b_1)\,\mathbf{tie}\,b_2 = (L\,\mathbf{tie}\,b_2)\,\mathbf{tie}\,b_1$ *(commutativity)*
3. $(L\,\mathbf{tie}\,b_1)\,\mathbf{sy}\,A_l = (L\,\mathbf{sy}\,A_l)\,\mathbf{tie}\,b_1$ *(commutativity with synchronization)*

Proof. *1.* By definition 6, operation **tie** makes desired links and removes concerned tie symbols from the transitions labels. A second application of **tie** over the same tie symbol does not change anything in the net.

2. By definition 6, operations **tie** for different tie symbols are totally independent, the order of applications has no importance.

3. By definition 6 and 4, when **tie** is applied first, it creates arcs which are inherited by the new transitions created by **sy**. Conversely, if **sy** is applied first, it transmits links to the new transitions, allowing **tie** to create the expected arcs.

Since operation **tie** is commutative, it naturally extends to a set of tie symbols.

Theorem 2. *Let N be an M-net, and $b \in \mathsf{B}$. Then:*
$$\mathcal{U}(N\,\mathbf{tie}\,b) = \mathcal{U}(N)\,\mathbf{tie}\,b.$$

Proof. It is enough to remark that the high-level tie operation creates a place s_b with type $type(b)$ and adds arcs to/from transitions which carry links on b in their inscriptions. The unfolding gives for this new place a set of places $\{s_{b,v} \mid v \in type(b)\}$, and the set of arcs to/from transitions t_σ. The weight of an arc between place $s_{b,v}$ and transition t_σ corresponds to the multiplicity of value v in the evaluation through σ of the high-level arc inscription between s_b and t. By definition, this is exactly what is done by the low-level tie operation.

Now, corollary 1 comes directly from the two above theorems and from the commutativity of synchronization with unfolding [6] (notice that, after the introduction of links, it is obvious that this commutativity is preserved).

Corollary 1. *Let N be an M-net, $A_h \in \mathsf{A}_h$ and b_1, $b_2 \in \mathsf{B}$. Then:*

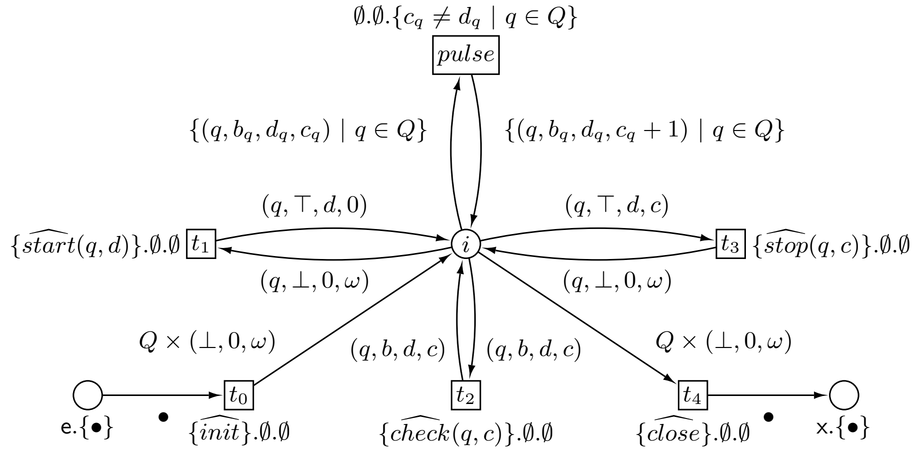
1. $(N\,\mathbf{tie}\,b_1)\,\mathbf{tie}\,b_1 = N\,\mathbf{tie}\,b_1$ *(idempotence)*
2. $(N\,\mathbf{tie}\,b_1)\,\mathbf{tie}\,b_2 = (N\,\mathbf{tie}\,b_2)\,\mathbf{tie}\,b_1$ *(commutativity)*
3. $(N\,\mathbf{tie}\,b_1)\,\mathbf{sy}\,A_h \equiv (N\,\mathbf{sy}\,A_h)\,\mathbf{tie}\,b_1$ *(commutativity with synchronization)*

where $\equiv$ identifies M-nets which are equivalent modulo renaming of variables [6].

7 An Application to Discrete Time Modeling

The newly introduced **tie** operator has an immediate application in a modeling of discrete time within M-nets. A clock is built over principles described in [10,15]: an arbitrary event (*i.e.*, a transition occurrence) is counted and used to build the time scale which all the system refers to. Two points can be puzzling: the time scale is not even and the clock can be frozen to allow the system meeting its deadlines. Both these points are discussed in [10] and are shown to be irrelevant in the context of Petri nets used for *specification* (by opposition to *programming*).

Our clock (which actually is a server) is implemented by the M-net N_{clock} depicted in figure 3. It can handle concurrent *counting requests* for different parts of the system: each request is assigned an identifier which is allocated by the clock on a *start* action; next, this identifier can be used to perform *check* actions which allows to retrieve current pulse-count for the request; finally, a *stop* can delete the request, sending back the pulse-count again.



$\emptyset.\emptyset.\{c_q \neq d_q \mid q \in Q\}$
$pulse$
$\{(q, b_q, d_q, c_q) \mid q \in Q\}$
$\{(q, b_q, d_q, c_q + 1) \mid q \in Q\}$
$(q, \top, d, 0)$
$(q, \top, d, c)$
$\{\widehat{start}(q, d)\}.\emptyset.\emptyset$
t_1
i
t_3
$\{\widehat{stop}(q, c)\}.\emptyset.\emptyset$
$(q, \bot, 0, \omega)$
$(q, \bot, 0, \omega)$
$Q \times (\bot, 0, \omega)$
(q, b, d, c)
(q, b, d, c)
$Q \times (\bot, 0, \omega)$
t_0
t_2
t_4
e.$\{\bullet\}$
$\bullet$
$\{\widehat{init}\}.\emptyset.\emptyset$
$\{\widehat{check}(q, c)\}.\emptyset.\emptyset$
$\{\widehat{close}\}.\emptyset.\emptyset$
$\bullet$
x.$\{\bullet\}$

Fig. 3. The M-net N_{clock}: the place i is labeled i.$Q \times \{\bot, \top\} \times \widetilde{\mathbb{N}} \times \widetilde{\mathbb{N}}$.

The clock N_{clock}, depicted in figure 3, works as follows:

- it starts when the transition t_0 fires and put tokens in the central place of N_{clock}, each token being a tuple (q, b, d, c), corresponding to a request, where $q \in Q$ is the identifier for the request, $b \in \{\bot, \top\}$ tells if the request is idle $(b = \bot)$ or in use $(b = \top)$, d is the maximum number of pulse to count for the request and c is the current pulse-count. Both d and c are values in $\widetilde{\mathbb{N}} = \mathbb{N} \cup \omega$ where ω is greater than any integer and $\omega + 1 = \omega$;
- a request begins when t_1 fires. *start* has two parameters: the request identifier, q, is sent to the client which provides the duration, d, as the maximum pulse-count for the request;
- next, the current pulse-count can be *check*ed with transition t_2: the identifier is provided by the client which gets the current pulse-count returned back through c;
- t_3 is used to *stop* a request, it acts like t_2 but also sets the request's token idle;
- the clock can be terminated with transition t_4;
- at any time, provided its guard is true, *pulse* can fire, incrementing the pulse-count for *all* requests. This explains the idle value $(q, \bot, 0, \omega)$ for the tokens in i: it does neither change after a pulse since $\omega = \omega + 1$ nor affect the guard on *pulse* because $\omega \neq 0$. This guard is used to prevent *pulse* from firing if a request is about to end (pulse-count is equal to the maximum announced on *start*), this ensures a timed sub-net will always meet its deadline if it has been announced on *start*.

In order to use this clock, one just has to add "clock actions" (*start*, *check* and *stop*) on the transitions intended to be timed and asynchronous links should be

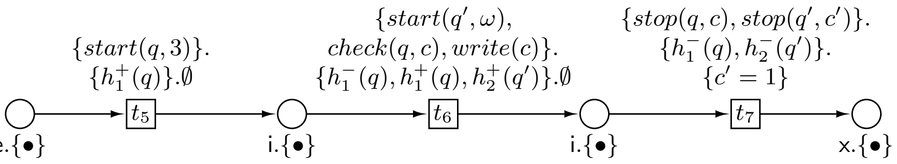

$\{start(q,3)\}.$
$\{h_1^+(q)\}.\emptyset$
t_5
$\{start(q',\omega),$
$check(q,c), write(c)\}.$
$\{h_1^-(q), h_1^+(q), h_2^+(q')\}.\emptyset$
t_6
$\{stop(q,c), stop(q',c')\}.$
$\{h_1^-(q), h_2^-(q')\}.$
$\{c'=1\}$
t_7
e.{●}
i.{●}
i.{●}
x.{●}

Fig. 4. An example of time-constrained M-net.

used to transport the request identifiers between a *start* and the corresponding *check*(s) and *stop*(s), like in figure 4.

In the net of figure 4, we specify the following constraints:

– the $start(q,3)$ on t_5 corresponds to the $stop(q,c)$ on t_7 (thanks to the links on h_1) so there can be at most 3 pulses between t_5 and t_7 because of the guard on *pulse*;
– similarly, there must be exactly 1 pulse between t_6 and t_7 (here we use the links on h_2 to transmit the identifier) but the constraint is expressed through the guard on t_7. In this case, since the deadline was not announced in the start, it is possible to have a deadlock if the system runs too slow, *i.e.*, does not meet its deadline;
– on t_6, we use a *check* for the request started on t_5 (the identifier is imported *and* exported on h_1) to retrieve the pulse count between t_5 and t_6. This count is sent to another part of the system with the action $write(c)$ which should be synchronized with a $\widehat{write}$, in a piece of the specification not represented here.

8 Conclusion

We presented an extension of M-nets and PBC in order to cope with asynchronous communications at the (net) algebra level. This extension led to a simple and elegant mean to model discrete time within M-nets, so we hope it would be useful to work with timed specifications. Moreover, we could expect to apply asynchronous links for a wider range of applications. In particular, we yet study: real-time programming with B(PN)^2 [7] (a parallel programming language with M-nets and PBC semantics), discrete timed automata simulation and a new implementation of B(PN)^2's FIFO channels.

References

1. V. Benzaken, N. Hugon, H. Klaudel, E. Pelz, and R. Riemann. M-net Based Semantics for Triggers. *ICPN'98* LNCS Vol. 1420 (1998).
2. E. Best. A Memory Module Specification using Composable High Level Petri Nets. Formal Systems Specification, The RPC-Memory Specification Case Study, LNCS Vol. 1169 (1996).

3. E. Best, R. Devillers, and J. Esparza. General Refinement and Recursion for the Box Calculus. *STACS'93*. Springer, LNCS Vol. 665, 130–140 (1993).
4. E. Best, R. Devillers, and J.G. Hall. The Box Calculus: a New Causal Algebra with Multilabel Communication. *APN'92*. Springer, LNCS Vol. 609, 21–69 (1992).
5. E. Best and H. Fleischhack, editors. *PEP: Programming Environment Based on Petri Nets*. Number 14/95 in Hildesheimer Informatik-Berichte. Univ. Hildesheim (1995).
6. E. Best, W. Fraczak, R.P. Hopkins, H. Klaudel, and E. Pelz. M-nets: an Algebra of High Level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. *Acta Informatica:35* (1998).
7. E. Best, R.P. Hopkins. $B(PN)^2$ – a Basic Petri Net Programming Notation. *PARLE'93*. Springer, LNCS Vol. 694, 379–390 (1993).
8. R. Devillers and H. Klaudel. Refinement and Recursion in a High Level Petri Box Calculus. *STRICT'95*. Springer, ViC, 144–159 (1995).
9. R. Devillers, H. Klaudel and R.-C. Riemann. General Refinement for High Level Petri Nets. *FST & TCS'97*, Springer, LNCS Vol. 1346, 297–311 (1997).
10. R. Durchholz. Causality, time, and deadlines. *Data & Knowledge Engineering*, 6:496–477 (1991).
11. H. Fleischhack and B. Grahlmann. A Petri Net Semantics for $\mathrm{B(PN)}^2$ with Procedures. *Parallel and Distributed Software Engineering*, Boston Ma. (1997).
12. H. Klaudel and R.-C. Riemann. Refinement Based Semantics of Parallel Procedures. In proceedings of *PDPTA'99* (1999).
13. M. Koutny, J. Esparza, E. Best. Operational Semantics for the Petri Box Calculus. *CONCUR'94*. Springer, LNCS Vol. 836, 210–225 (1994).
14. J. Lilius and E. Pelz. An M-net Semantics for $\mathrm{B(PN)}^2$ with Procedures. *ISCIS XI*, Antalya, Vol. I, 365–374 (1996).
15. G. Richter. Counting interfaces for discrete time modeling. Technical Report repset-1998-26, GMD – German National Research Center for Information Technology, SET (1998).

Demand-Driven Model Checking for Context-Free Processes

Jens Knoop

Universität Dortmund, D-44221 Dortmund, Germany
phone: ++49-231-755-5803, fax: ++49-231-755-5802
`knoop@ls5.cs.uni-dortmund.de`
`http://sunshine.cs.uni-dortmund.de/~knoop`

Abstract. We propose a *demand-driven* model-checking algorithm, which decides the alternation-free modal mu-calculus for *context-free* processes. This algorithm enjoys advantages known from local model checking in that it avoids the investigation of certain irrelevant parts of a process, and simultaneously improves on its classical counterpart of [5] in that it avoids the computation of irrelevant portions of property transformers. In essence, this algorithm evolves from combining the spirit of second-order model checking underlying the algorithm of [5] with the idea of demand-drivenness developed in the field of interprocedural data-flow analysis. Though the new algorithm has the same worst-case time complexity as its counterpart, we expect a substantial performance gain in practice because its demand-drivenness reduces the computational effort of those parts, which are responsible for the exponentiality of the classical second-order algorithm.

1 Motivation

Model checking (cf. [7, 10, 16]) has proved to be a powerful and practically relevant means for the automatic verification of behavioral systems (cf. [8, 15, 19]). Particularly ambitious are approaches aiming at model checking of infinite state systems, which has been pioneered by approaches for *local* model checking (cf. [20, 21]). The point of local model checking is to avoid the investigation of parts of a process, which are irrelevant for the verification of the property under consideration. Straightforward applications to infinite state systems, however, are generally not effective because they cannot guarantee termination (cf. [3, 4]).

The construction of effective algorithms, however, becomes possible as soon as one restricts one's attention to specific classes of infinite state systems as e.g. context-free (cf. [5, 12]) or pushdown process systems (cf. [2, 6, 22]). In this article we focus on *context-free process systems* (*CFPSs*) and reconsider the classical algorithm proposed by Burkart and Steffen for this setting (cf. [5]). Their algorithm decides the alternation-free modal mu-calculus for *context-free* process systems (CFPSs), i.e., for processes which are given in terms of a context-free grammar. The central idea underlying their algorithm is to raise the standard model-checking techniques to *second* order: In contrast to the usual approaches, in which the set of formulae which are satisfied by a certain state are iteratively computed, their algorithm iteratively computes a *property transformer* for each state class of the finite process representation, which is subsequently used for solving the original model-checking problem. Later on, this algorithm has been complemented by Hungar and Steffen by a *local* model-checking algorithm [12], which improves on the algorithm of [5] in that irrelevant parts of the process under consideration need not to be investigated.

In this article we develop a *demand-driven* model-checking algorithm for the setting considered in [5, 12], which decides the alternation-free modal mu-calculus for CFPSs. Like the algorithm of [12] the new algorithm aims at improving the average-time complexity of the algorithm of [5]. It enjoys advantages known from local model checking in that it avoids the investigation of irrelevant parts of a process, and simultaneously improves on the original algorithm of [5] in that it avoids the computation of irrelevant property transformers and irrelevant portions of property transformers. To achieve this we adopt as in [5] the view of CFPSs as mutually recursive systems of finite state-labeled transition systems. This establishes the link to *interprocedural data-flow analysis* (cf. [13]), which provides the key to our approach. In essence, our algorithm evolves from combining the spirit underlying the classical second-order algorithm of [5] with the idea of demand-drivenness of interprocedural data-flow analysis (cf. [9, 11, 17]).

It is worth noting that the investigation of irrelevant parts of a process and of the computation of completely irrelevant property transformers could be avoided in the approach of [5], too; however, at the price of an additional preprocess only. Avoiding the computation of irrelevant portions of specific property transformers, however, is out of the scope of this approach. This, however, is quite important in practice because the worst-case time complexity of the algorithm of [5] is linear in the size of the finite representation of the process under consideration, but exponential in the size of the formula to be verified, which, in essence, determines the complexity of computing the property transformers.[1] Though generally demand-driven algorithms have the same worst-case time complexity as their standard counterparts, in practice they are usually much more efficient as e.g. empirically confirmed in interprocedural data-flow analysis (IDFA) (cf. [9]). As in IDFA, we expect that the new, demand-driven algorithm performs significantly better in practice than its counterpart of [5], since the demand-drivenness of the new algorithm reduces the computational effort of those parts which are responsible for the exponentiality of its counterpart. In general, the performance gain will be the larger the more complex the property under consideration is.

2 Processes and Formulae

In this section we introduce our setting following the lines of [5, 12]. Central are context-free process systems as finite representations of infinite process graphs, and the (alternation-free) modal mu-calculus as our logic for specification.

[1] More precisely, it is exponential in the *weight* of the formula (cf. [5]). Deciding the alternation-free modal mu-calculus for CFPSs was recently shown to be EXPTIME-complete (cf. [14]), while it is polynomial for any fixed formula (cf. [5]).

Definition 1 (Process graph). *A* process graph (PG) *is a quintuple* $G = \langle \mathcal{S}, Act, \rightarrow, s_0, s_e \rangle$*, where (1)* $\mathcal{S}$ *is a set of* states*, (2)* Act *is a set of* actions*, (3)* $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ *is the* transition relation*, and (4)* $s_0, s_e \in \mathcal{S}$ *are distinguished elements, the "start state" and the "end state."* s_0 *and* s_e *must be* originating *and* terminating*, respectively, i.e., there are no* $a \in Act$ *and* $s \in \mathcal{S}$ *with* $\langle s, a, s_0 \rangle \in \rightarrow$ *or* $\langle s_e, a, s \rangle \in \rightarrow$*. A PG G is said to be* finite-state*, when its sets of states* $\mathcal{S}$ *and actions* Act *are finite.*

Intuitively, a process graph encodes the operational behavior of a process. The set $\mathcal{S}$ represents the set of states the process can enter, Act the set of actions the process can perform, and $\rightarrow$ the state transitions which may result upon execution of the actions. As usual we will write $s \xrightarrow{a} s'$ instead of $\langle s, a, s' \rangle \in \rightarrow$. Moreover, we will write $s \xrightarrow{a}$, when there is an s' such that $s \xrightarrow{a} s'$.

As in [5], we represent context-free processes, which may have infinitely many states, by means of context-free process systems (CFPSs). In essence, a CFPS is a set of named finite process graphs, whose set of actions contains the names of the system's process graphs. Transitions labeled with such a name are meant to represent the denoted process graph. In this view, the names of the process graphs correspond to the non-terminals and the atomic actions to the terminals of a context-free grammar. Alternatively, transitions labeled by a name can be considered a call of the denoted process graph. As in [5] we prefer this more dynamic procedural point of view: It establishes the link to interprocedural data-flow analysis [18, 13], which is essential for developing our demand-driven algorithm for model checking CFPSs.

Definition 2 (Procedural process graph). *A* procedural process graph (PPG) *is a quintuple* $P = \langle \Sigma_P, Trans, \rightarrow_P, \sigma_P^s, \sigma_P^e \rangle$*, where (1)* Σ_P *is a set of* state classes*, (2)* $Trans =_{df} Act \cup \mathcal{N}$ *is a set of* transformations*, where* Act *is a set of atomic* actions*, and* $\mathcal{N}$ *a set of* names *with* $Act \cap \mathcal{N} = \emptyset$*, (3)* $\rightarrow_P = \rightarrow_P^{Act} \cup \rightarrow_P^{\mathcal{N}}$ *is the* transition relation*, where* $\rightarrow_P^{Act} \subseteq \Sigma_P \times Act \times \Sigma_P$ *and* $\rightarrow_P^{\mathcal{N}} \subseteq \Sigma_P \times \mathcal{N} \times \Sigma_P$*, and (4)* $\sigma_P^s \in \Sigma_P$ *and* $\sigma_P^e \in \Sigma_P$ *are a class of "start states" and "end states." A PPG P is called* finite*, if* Σ_P *and* $Trans$ *are finite.*

Essentially, a procedural process graph is a process graph, where the set of actions is divided into two disjoint classes, atomic actions Act and (action) names $\mathcal{N}$. A PPG P is called *guarded*, if all initial transitions of P are labeled by atomic actions. It is called *simple*, if it does not contain any calls. Following [5] we denote the set of all simple PPGs by $\mathcal{G}$. In the following we will only consider guarded PPGs P, where the class of end states σ_P^e is *terminating*.

Definition 3 (Context-free process system). *A* context-free process system (CFPS) *is a sextuple* $\mathcal{P} = \langle \mathcal{N}, Act, \Delta, P_0, s_0, s_e \rangle$*, where (1)* $\mathcal{N} = \{N_0, \ldots, N_{n-1}\}$ *is a set of* names*, (2)* Act *is a set of* actions*, (3)* $\Delta =_{df} \{N_i = P_i \mid 0 \leq i < n\}$ *is a finite set of PPG-definitions, where the* P_i *are finite PPGs with names in* $\mathcal{N}$ *and atomic actions in* Act*, (4)* P_0 *is the "main" PPG, and (5)* s_0 *and* s_e *are the start state and the end state of the system.*

We denote the union of all state classes of $\mathcal{P}$ *by* $\Sigma =_{df} \bigcup_{i=0}^{n-1} \Sigma_{P_i}$*, and the union of all transition relations of* $\mathcal{P}$ *by* $\rightarrow =_{df} \bigcup_{i=0}^{n-1} \rightarrow_{P_i}$*.*

Figure 1 shows a CFPS consisting of three components, which is sufficient to illustrate the essence of our approach, and to highlight the essential differences between our algorithm for demand-driven model checking of CFPSs and the classical algorithm of [5] (cf. Section 4.2). Actually, the CFPS shown is a variant of the encoding machine used as the running example in [5]. Intuitively, in its first phase the encoding machine of Figure 1 reads a word over the alphabet $\{a, b\}$ until it encounters the word delimiter #. Subsequently, in its second phase it outputs the corresponding sequence of encoded characters in reverse order. Note that the end state of P_0 is not reachable from P_2. This will be discussed in detail in Section 4.2. It allows us to illustrate an important difference between the algorithm of [5] and the algorithm here.

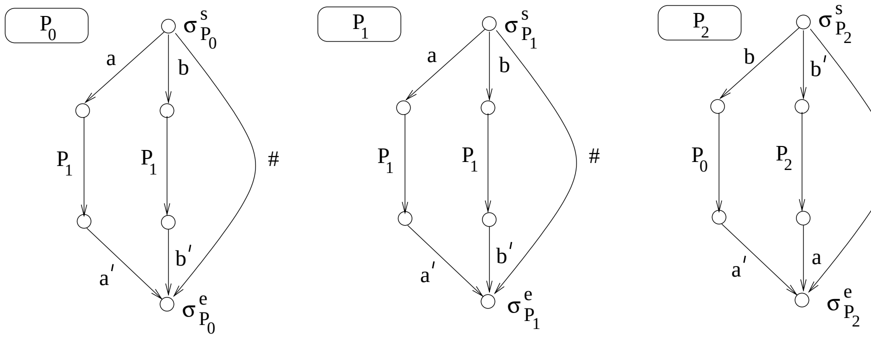


Fig. 1. The running example: The encoding machine.

Definition 4 (Complete expansion). *Let P be a PPG of a CFPS $\mathcal{P}$. The* complete expansion *of P with respect to $\mathcal{P}$ is the simple PPG, which results from successively replacing in P each transition $\sigma \xrightarrow{P_i} \sigma'$ by a copy of the corresponding PPG P_i, while identifying σ with $\sigma^s_{P_i}$ and σ' with $\sigma^e_{P_i}$. We denote the complete expansion of P with respect to $\mathcal{P}$ by $Exp_{\mathcal{P}}(P)$.*

Figure 2 illustrates the flavour of this stepwise expansion process, which reminds to the copy-rule semantics of imperative languages (cf. [1]). A flavour, which is also reflected in the following definition. Given a state $s \in \mathcal{S}$ of a complete expansion $Exp_{\mathcal{P}}(P)$, s is called to belong to the state class $\sigma \in \Sigma_{P_i}$, if it emerges as a copy of σ during the expansion. Thus, a state class stands for a possibly infinite set of states of the corresponding complete expansion.

If $P = \langle \Sigma_P, Trans, \rightarrow_P, \sigma^s_P, \sigma^e_P \rangle$ is a PPG and σ a state class of P, we denote by $P^{(\sigma)}$ the PPG $\langle \Sigma_P, Trans, \rightarrow_P, \sigma, \sigma^e_P \rangle$. If P_1 and P_2 are two PPGs, we define their *sequential composition* $P_1; P_2$ as the PPG $P_1; P_2 =_{df} \langle \Sigma_{P_1} \cup \Sigma_{P_2} \backslash \{\sigma^s_{P_2}\}$, $Trans_{P_1} \cup Trans_{P_2}, \rightarrow_{P_1} \cup \rightarrow_{P'_2}, \sigma^s_{P_1}, \sigma^e_{P_2} \rangle$, where $\rightarrow_{P'_2}$ denotes the transition relation resulting from substituting all occurrences of $\sigma^s_{P_2}$ in $\rightarrow_{P_2}$ by $\sigma^e_{P_1}$. Finally, $\sigma^e_{ppg(\sigma)}$ denotes the end state of the particular PPG-definition of the underlying CFPS containing σ.

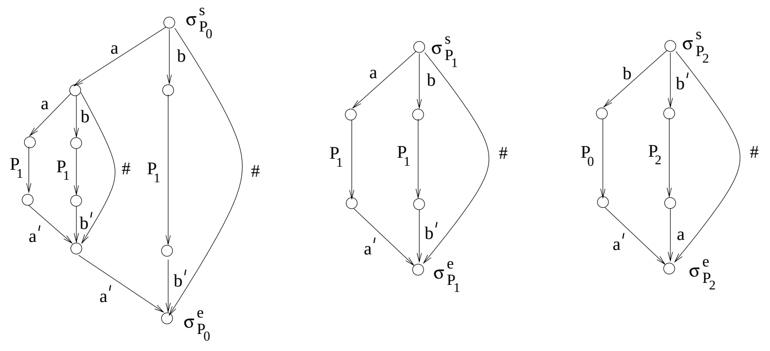


Fig. 2. Expanding a procedural transition system.

The Mu-Calculus. As in [12] we consider a negation-free sublanguage of the modal mu-calculus, which, however, as in [5], is based on an explicit set of atomic propositions. Its syntax is given by the grammar $\Phi ::= A \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi \mid [a]\Phi \mid \nu X.\Phi \mid \mu X.\Phi$, where X ranges over a (countable) set of variables Var, A over a set of atomic propositions $\mathcal{A}$, and a over a set of actions Act. Properties will later be specified by *closed* formulae, i.e., formulae which do not contain any free variable. A formula is called *alternation-free*, if no μ-subformula has a free variable which, in the context of the complete formula, is bound by a ν, and vice versa. We denote the set of all closed alternation-free formulae by $\mathcal{M}$.

As usual the semantics of formulae is defined with respect to a specific (possibly infinite) process graph $G = \langle \mathcal{S}, Act, \rightarrow, s_0, s_e \rangle$, a valuation $\mathcal{V} : \mathcal{A} \rightarrow 2^{\mathcal{S}}$, and an environment $e : Var \rightarrow 2^{\mathcal{S}}$. Intuitively, it maps a formula (with free variables) to the set of states for which the formula is "true." Correspondingly, a state s satisfies X, if s is an element of the set bound to X in environment e. Note that the semantics of a closed formula Φ is independent of the environment e. We will thus write $s \models \Phi$ instead of $s \in [\![\Phi]\!] e$ for all environments e.

$$
\begin{aligned}
[\![A]\!]e &= \mathcal{V}(A) & [\![\langle a \rangle \Phi]\!]e &= \{s \mid \exists s'.\ s \xrightarrow{a} s' \ \wedge\ s' \in [\![\Phi]\!]e\} \\
[\![X]\!]e &= e(X) & [\![[a]\Phi]\!]e &= \{s \mid \forall s'.\ s \xrightarrow{a} s' \ \wedge\ s' \in [\![\Phi]\!]e\} \\
[\![\Phi_1 \vee \Phi_2]\!]e &= [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e & [\![\nu X.\Phi]\!]e &= \bigcup\{S' \subseteq S \mid S' \subseteq [\![\Phi]\!]e[X \mapsto S']\} \\
[\![\Phi_1 \wedge \Phi_2]\!]e &= [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e & [\![\mu X.\Phi]\!]e &= \bigcap\{S' \subseteq S \mid S' \supseteq [\![\Phi]\!]e[X \mapsto S']\}
\end{aligned}
$$

Table 1. The semantics of the subset of the modal mu-calculus under consideration.

Second-Order Semantics. According to Table 1 the validity of a formula is defined with respect to single states. An analogous definition for a state class is in general not meaningful because the truth value of a formula is in general different for different representatives of a state class. However, two states s and s' of the same state class satisfy the same set of formulae if this holds for their corresponding end states $end(s)$ and $end(s')$. As pointed out in [12], this is the key to the second-order semantics defined in [5], which can be considered the analogue of the second-order *functional* approach of interprocedural data-flow analysis (cf. [18, 13]). It considers the semantics of a (named) PPG as a *property transformer*, i.e., as a function which yields the set of formulae being valid at the start state relative to the assumption that the set of formulae it is applied to are valid at the end state.

Definition 5 (Second-order semantics). *Let* $P = \langle \Sigma_P, Trans, \rightarrow_P, \sigma_P^s, \sigma_P^e \rangle$ *be a simple PPG. Then we interpret* P *as the function* $[\![P]\!] : 2^{\mathcal{M}} \rightarrow 2^{\mathcal{M}}$ *with* $[\![P]\!](M) =_{df} \{\Phi' \in \mathcal{M} \mid \forall P' \in \mathcal{G}.(P' \cap P = \emptyset).\ \sigma_P^e \models_{P;P'} M \Rightarrow \sigma_P^s \models_{P;P'} \Phi'\}$.

Theorem 1 guarantees the consistency of the second-order semantics with respect to the usual semantics of simple PPGs in terms of its valid formulae (cf. [12]).

Theorem 1 (Consistency of the second-order semantics). *Let* P *be a simple PPG and* Φ *be a closed formula, and let* $\mathcal{F}_{deadlock}$ *denote the set of all propositions which are "true" at a "dead-lock state," i.e.,* $\mathcal{F}_{deadlock} =_{df} \{\Phi \mid s \models_{\mathcal{T}} \Phi$ *with* $\mathcal{T} = \langle \{s\}, Act, \emptyset \rangle\}$. *Then we have:* $\sigma_P^s \models \Phi \iff \Phi \in [\![P]\!](\mathcal{F}_{deadlock})$.

3 Hierarchical Equational Systems

As in [5] we present our algorithm for demand-driven model checking of CFPSs in a version where logic formulae are represented by hierarchical equational systems. These are equally expressive as the alternation-free modal mu-calculus $L\mu_1$ (cf. Theorem 3 ([10])), but simplify the technical presentation of the algorithm and its comparison with the classical algorithm of [5]. Therefore, we briefly recall the syntax and semantics of hierarchical equational systems in this section.

Syntax. Fundamental for the definition of hierarchical equational systems is the notion of a *basic* formula. Their syntax is given by the grammar $\Phi ::= A \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle a \rangle \Phi \mid [a]\Phi$. Because of the absence of fixpoint operators, they are less expressive than the formulae defined in Section 2. This, however, is overcome by introducing (mutually recursive) equational systems.

Definition 6 ((Equational) blocks and systems). *An* (equational) block B *has one of the two forms,* $min\{E\}$ *or* $max\{E\}$, *where* E *is a list of (mutually recursive) equations* $\langle X_1 = \Phi_1, \ldots, X_n = \Phi_n \rangle$, *in which each* Φ_i *is a basic formula and the* X_i *are pairwise disjoint, and where min and max indicate the fixed point of* E *desired, i.e., the least and greatest fixed point, respectively. An* equational system $\mathcal{B} = \langle B_1, \ldots, B_m \rangle$ *is a list of equational blocks, where the variables appearing on the left-hand sides of the blocks are all pairwise disjoint.*

Intuitively, a block defines n (mutually recursive) propositions, i.e., one per variable (see Section 4.2 for an example). Several blocks may be used to express complex formulae. In particular, the left-hand side of an equation in one block may be referred to in the right-hand sides of equations in other blocks. Since we are focusing on the alternation-free subset of the mu-calculus, we restrict our attention to *hierarchical equational systems.* The restrictions they obey (see Figure 3 for illustration) guarantee the desired absence of *alternating fixed points.* In terms of Section 2, the formulae they represent are alternation-free (cf. [10]).

Definition 7 (Hierarchical equational system). *An equational system $\mathcal{B} = \langle B_1, \dots, B_m \rangle$ is* hierarchical, *if the existence of a left-hand-side variable of a block B_j appearing in a right-hand-side formula of a block B_i implies $i \leq j$.*

Semantics. The semantics of a hierarchical equational system $\mathcal{B}$ is defined on top of the semantics of individual blocks B. To this end, let E be the list of equations $\langle X_1 = \Phi_1, \dots, X_n = \Phi_n \rangle$. For every environment ρ, we can now build a function $f_E^\rho : (2^{\mathcal{S}})^n \to (2^{\mathcal{S}})^n$ as follows. Let $\bar{S} = \langle S_1, \dots, S_n \rangle \in (2^{\mathcal{S}})^n$, and let $\rho_{\bar{S}} = \rho[X_1 \mapsto S_1, \dots, X_n \mapsto S_n]$ be the environment, which results from ρ by updating the binding of X_i to S_i. Then: $f_E^\rho(\bar{S}) =_{df} \langle [\![\Phi_1]\!] \rho_{\bar{S}}, \dots, [\![\Phi_n]\!] \rho_{\bar{S}} \rangle$.

The domain $(2^{\mathcal{S}})^n$ forms a complete lattice, where the ordering, join, and meet operations are the pointwise extensions of the set-theoretic inclusion $\subseteq$, union $\cup$, and intersection $\cap$, respectively. Moreover, for any equational system B and environment ρ, the function f_E^ρ is monotonic with respect to the order relation of this lattice. According to Tarski's well-known fixed-point theorem, it has thus a *greatest* fixed point, νf_E^ρ, and a *least* fixed point, μf_E^ρ, satisfying $\nu f_E^\rho = \bigcup \{ \bar{S} \mid \bar{S} \subseteq f_E^\rho(\bar{S}) \}$ and $\mu f_E^\rho = \bigcap \{ \bar{S} \mid \bar{S} \supseteq f_E^\rho(\bar{S}) \}$ (cf. [5]).

Blocks $max\{E\}$ and $min\{E\}$ are interpreted as *environments*: $[\![max\{E\}]\!] \rho = \rho_{\nu f_E^\rho}$ and $[\![min\{E\}]\!] \rho = \rho_{\mu f_E^\rho}$. This means that $max\{E\}$ and $min\{E\}$ represent the "greatest" and the "least" fixed point of B. The relative semantics of a hierarchical equational system $\mathcal{B} = \langle B_1, \dots, B_m \rangle$ with respect to ρ is then defined in terms of a sequence of environments: $\rho_m = [\![B_m]\!] \rho$, $\rho_{m-1} = [\![B_{m-1}]\!] \rho_m$, $\dots$, $\rho_1 = [\![B_1]\!] \rho_2$. The semantics of $\mathcal{B}$, finally, is defined by (cf. [5]): $[\![\mathcal{B}]\!] \rho =_{df} \rho_1$.

The notion of closed formulae can easily be extended to hierarchical equational systems $\mathcal{B}$. A basic proposition Φ is *closed* with respect to $\mathcal{B}$, if every variable in Φ appears on the left-hand side of some equation of some block of $\mathcal{B}$. $\mathcal{B}$ is *closed*, if each right-hand side of each block of $\mathcal{B}$ is closed with respect to $\mathcal{B}$. Theorem 2 yields that the

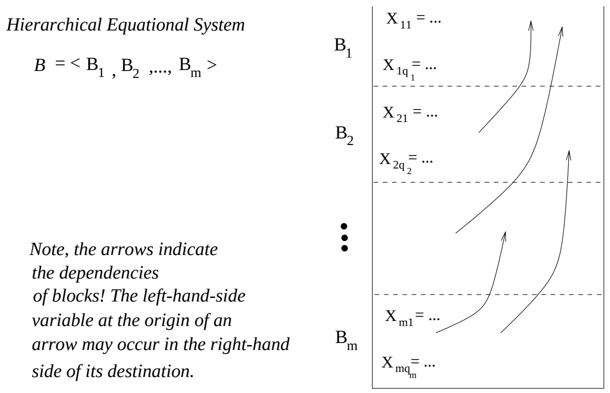

Hierarchical Equational System
$B = \langle B_1, B_2, \dots, B_m \rangle$
Note, the arrows indicate the dependencies of blocks! The left-hand-side variable at the origin of an arrow may occur in the right-hand side of its destination.
B_1
B_2
B_m
$X_{11} = \dots$
$X_{1q_1} = \dots$
$X_{21} = \dots$
$X_{2q_2} = \dots$
$X_{m1} = \dots$
$X_{mq_m} = \dots$

Fig. 3. Hierarchical equational systems.

consideration of variables is even sufficient (cf. [5]). Theorem 3, subsequently, establishes the desired link between the alternation-free modal mu-calculus recalled in Section 2 and hierarchical equational systems (cf. [10]).

Theorem 2. *Let $\mathcal{B}$ be a closed hierarchical equational system, and let Φ be a (basic) proposition which is closed with respect to $\mathcal{B}$. Then we have: (1) $[\![\,\Phi\,]\!]_{[\![\,\mathcal{B}\,]\!]_\rho} = [\![\,\Phi\,]\!]_{[\![\,\mathcal{B}\,]\!]_{\rho'}}$ for any ρ and ρ'. (2) There is a closed hierarchical equational system $\mathcal{B}'$ having X' as the first variable of its first block such that $[\![\,\Phi\,]\!]_{[\![\,\mathcal{B}\,]\!]} = [\![\,X'\,]\!]_{[\![\,\mathcal{B}'\,]\!]}$.*

Theorem 3 (Expressiveness). *Let $\mathcal{T}$ be a labeled transition system, and let ρ be a corresponding environment. Then we have: (1) Every formula Φ in $L\mu_1$ can be translated in time linear to the size of Φ into a hierarchical equational system $\mathcal{B}$ with $[\![\,\Phi\,]\!]_{[\![\,\rho\,]\!]} = [\![\,X\,]\!]_{[\![\,\mathcal{B}\,]\!]_\rho}$ for some left-hand-side variable X of $\mathcal{B}$. (2) For every hierarchical equational system $\mathcal{B}$ and every variable X there is a formula Φ in $L\mu_1$ with $[\![\,X\,]\!]_{[\![\,\mathcal{B}\,]\!]_\rho} = [\![\,\Phi\,]\!]_{[\![\,\rho\,]\!]}$.*

4 Demand-Driven Model Checking of CFPSs

In this section we present our algorithm for demand-driven model checking of CFPSs. In order to simplify a direct comparison with its counterpart of [5], we present our algorithm for *min*-blocks. This is sufficient because the treatment of *max*-blocks is completely dual, and the hierarchical extension required for hierarchical equational systems is straightforward in an innermost fashion.

Conventions. In order to simplify the presentation of our algorithm, we assume as in [5] without loss of generality the following structure for the (*min*-) block B under consideration: (1) The right-hand sides of blocks consist of *simple* basic formulae, which are characterized by the grammar $\Phi^{simple} ::= A \mid X \vee X \mid X \wedge X \mid \langle a \rangle X \mid [a]X$. (2) The graph of the *unguarded dependencies* on the variables of B, which is defined by having an arc $X_i \rightarrow X_j$ if and only if X_j appears unguardedly in the right-hand side of the equation for X_i, is acyclic. Importantly, a block can always be transformed into an equivalent block of the same size satisfying these constraints. The first constraint is established by a step introducing a new variable for every complex subexpression. The second constraint is established by a step, whose essence is summarized below, and where it should be noted that X_1 is unguarded in the right-hand-side terms.

$$\begin{array}{ll} min\{X_1 = X_1 \vee \Phi\} \leadsto min\{X_1 = \Phi\} & max\{X_1 = X_1 \wedge \Phi\} \leadsto max\{X_1 = \Phi\} \\ min\{X_1 = X_1 \wedge \Phi\} \leadsto ff & max\{X_1 = X_1 \vee \Phi\} \leadsto tt \end{array}$$

Additional Notations. Let B be a closed equational *min*-block with left-hand-side variables $\mathcal{X} = \{X_i \mid 1 \leq i \leq r\}$, where X_1 represents the property to be investigated. Additionally, let (1) $\mathcal{D} =_{df} 2^{\mathcal{X}} \rightarrow 2^{\mathcal{X}}$ be the set of all functions on $2^{\mathcal{X}}$, let (2) $PT_\emptyset \in \mathcal{D}$ be the function which maps every $M \subseteq \mathcal{X}$ to the empty set, and let (3) PT_{Id} be the identity on $\mathcal{D}$. Fundamental is then to associate with each $\sigma \in \Sigma$ of a CFPS $\mathcal{P}$ a function $PT_\sigma \in \mathcal{D}$ and with each transition

$\xrightarrow{\alpha}$ a function $PT_{[\![\,\alpha\,]\!]} \in \mathcal{D}$. The function $PT_{[\![\,\alpha\,]\!]} \in \mathcal{D}$ represents the property transformer of the process graph $\sigma^s \xrightarrow{\alpha} \sigma^e$. It is defined according to the following two cases:

- *Case 1*: $\alpha \equiv P_j \in \mathcal{N}$: $\quad PT_{[\![\,P_j\,]\!]} =_{df} PT_{\sigma^s_{P_j}}$
- *Case 2*: $\alpha \equiv a \in Act$: Let $M \subseteq \mathcal{X}$ and $X_j = \Phi_j$ be some equation of B.

$$\text{Then } X_j \in PT_{[\![\,\alpha\,]\!]}(M) \text{ iff } \begin{cases} \Phi_j = \langle a \rangle X_i \text{ and } X_i \in M \\ \Phi_j = [a] X_i \text{ and } X_i \in M \\ \Phi_j = [b] X_i \text{ and } b \neq a \end{cases}$$

If PT_i, $i \in \{1,\dots,k\}$, is a property transformer in $\mathcal{D}$ and $\sigma \in \Sigma$, then the function $\diamond^{\sigma}_{i=1,\dots,k} PT_i$ is defined as follows. Given $M \subseteq \mathcal{X}$ and the equation $X_j = \Phi_j$ of B, $M' =_{df} (\diamond^{\sigma}_{i=1,\dots,k} PT_i)(M)$ is defined by

$$X_j \in M' \text{ iff } \begin{cases} \Phi_j = A & \text{and } \sigma \in \mathcal{V}(A) \\ \Phi_j = X_{j_1} \wedge X_{j_2} & \text{and } X_{j_1} \in M' \text{ and } X_{j_2} \in M' \\ \Phi_j = X_{j_1} \vee X_{j_2} & \text{and } (X_{j_1} \in M' \text{ or } X_{j_2} \in M') \\ \Phi_j = \langle a \rangle X' & \text{and there is an } i \in \{1,\dots,k\} \text{ with } X_j \in PT_i(M) \\ \Phi_j = [a] X' & \text{and } X_j \in PT_i(M) \text{ holds for all } i \in \{1,\dots,k\} \end{cases}$$

The acyclicity of the graph of unguarded dependencies guarantees that this definition is well-founded. This allows us to present our algorithm now in detail.

4.1 The Algorithm

The classical algorithm of [5] has a three-step structure. The first step computes for every state class of the finite process representation the complete property transformer representing their second-order semantics. The second step computes the set of subformulae of the property Φ to be checked, which are satisfied by a deadlock state, i.e., $\mathcal{B}^{\Phi}_{deadlock}$. The third step, finally, applies the property transformer of the main PPG P_0 to the argument $\mathcal{B}^{\Phi}_{deadlock}$ computed in step two in order to decide according to Theorem 1 the model-checking problem.

Our demand-driven algorithm enjoys a similar structure. However, the computation of property transformers is only triggered for arguments which are really needed in order to decide the model-checking problem. Our algorithm thus starts with computing the set of formulae, which are satisfied by a deadlock state $\mathcal{B}^{\Phi}_{deadlock}$; a step, which belongs to the epilogue of the original algorithm. The key component of our algorithm is the procedure *Process*. It is the analogue of the procedure *update* of the algorithm of [5]. In contrast to *update*, whose domain are predicate transformers, and which thus updates a predicate transformer always as a whole, i.e., for all arguments of its domain, *Process* updates a predicate transformer pointwise, i.e., for a single argument at a time. This is essential for the demand-drivenness of our approach. The procedure *DynamicInitialization*, finally, is the analogue of the initialization procedure of the algorithm of [5]. Again, the essential difference is that *DynamicInitialization* works pointwise, i.e., demand-drivenly. Whenever the computation of the predicate transformers of a process graph is triggered for some argument, they are initialized for this particular argument only, instead of for their complete domain.

Next the demand-driven algorithm is given for the basic case of a closed formula $\Phi \in \mathcal{M}$ of the alternation-free modal μ-calculus. We recall that Φ is assumed to be equivalently given in terms of a (hierarchical) equational system $\mathcal{B} = \langle B_1 \rangle$, which is assumed to be a *min*-block, where X_1 denotes the left-hand-side variable of the first equation of B_1. As its counterpart of [5] this algorithm can straightforwardly be hierarchically extended. After termination (of the complete) algorithm the value stored in *answerToModelCheckingProblem* specifies the truth value of Φ with respect to $\mathcal{P}$, i.e., whether $\sigma^s_{P_0} \in [\![\Phi]\!]$.

```
( Prologue )        $M_{deadlock} := \mathcal{B}^{\Phi}_{deadlock}$;
( Main process )    $alreadyTriggered := \emptyset$; $workset := \emptyset$;
                    $DynamicInitialization(P_0, M_{deadlock})$;
                    WHILE $workset \neq \emptyset$ DO
                        CHOOSE $(\sigma, M) \in workset$;
                            $workset := workset \setminus \{ (\sigma, M) \}$;
                            $Process((\sigma, M))$
                        ESOOHC OD;
( Epilogue )        $answerToModelCheckingProblem := X_1 \in PT_{\sigma^s_{P_0}}(M_{deadlock})$.

PROCEDURE $DynamicInitialization(P, M)$;
    $PT_{\sigma^e_P}(M) := M$;
    FORALL $\sigma \in \Sigma_P \setminus \{ \sigma^e_P \}$ DO $PT_\sigma(M) := \emptyset$ OD;
    $alreadyTriggered := alreadyTriggered \cup \{(\sigma^e_P, M)\}$;
    $workset := workset \cup \{ (\sigma^e_P, M) \}$ END $DynamicInitialization$;

PROCEDURE $Process\ ((\sigma, M))$;
    FORALL $\sigma' \xrightarrow{\alpha} \sigma \in \{\sigma'' \xrightarrow{\alpha} \sigma \mid \sigma'' \in \Sigma \wedge \alpha \in Trans\}$ DO
        IF $\alpha \in \mathcal{N}$ THEN
            IF $(\sigma^e_\alpha, PT_\sigma(M)) \notin alreadyTriggered$ THEN
                $DynamicInitialization(\alpha, PT_\sigma(M))$ FI FI;
        $tmp := PT_{\sigma'}(M)$; $PT_{\sigma'}(M) := (\diamond^{\sigma'}\{PT_{\sigma'}, PT_{[\![ \alpha ]\!]} \circ PT_\sigma\})(M)$;
        IF $PT_{\sigma'}(M) \neq tmp$ THEN
            IF $\sigma' = \sigma^s_{P_i}$ for some $i \in I$ THEN
                $workset := workset \cup \{(\sigma'', M'') \mid \sigma' \xrightarrow{P_i} \sigma'' \wedge \sigma'' \in \Sigma \wedge$
                                $(\sigma^e_{ppg(\sigma'')}, M'') \in alreadyTriggered \wedge PT_{\sigma''}(M'') = M\}$
            ELSE $workset := workset \cup \{(\sigma', M)\}$ FI FI END $Process$;
```

4.2 Discussion

In this section we compare our demand-driven model-checking algorithm with its conventional counterpart and illustrate the benefits it offers by means of the running example and the property Φ considered for illustration in [5]. This property is as follows: The encoding machine of Figure 1 reads the full (finite) sequence of input symbols before it outputs the first code. Restricting the attention to finite inputs which are terminated by the character #, Φ can be rephrased as follows: After outputting an encoded character a' or b' the machine will always refuse to read #. Below, this property Φ is expressed in terms of a *max*-block, where X_1 corresponds to Φ.

In this example the new demand-driven algorithm improves in two respects on the algorithm of [5]: (1) The property transformers corresponding to the states of PPG P_2 are not computed at all. (2) The property transformers corresponding to the states of the PPGs P_0 and P_1 are only computed as far as it is necessary to decide the model-checking problem. Considering P_0, this is particularly evident: The computation of the property transformers of the states of P_0 is only triggered for a single argument, the set of formulae satisfied by a deadlock state, instead of for all elements of the power set of subformulae of the complete property.

In practice, this can be the source of substantial performance gains.

$$max \left\{ \begin{array}{llll} X_1 = X_2 \wedge X_3 & X_6 = [\#]X_1 & Y_1 = Y_2 \wedge Y_3 & Y_6 = [\#]Y_7 \\ X_2 = [a]X_1 & X_7 = X_8 \wedge X_9 & Y_2 = [a]Y_1 & Y_7 = ff \\ X_3 = X_4 \wedge X_5 & X_8 = [a']Y_1 & Y_3 = Y_4 \wedge Y_5 & Y_8 = Y_9 \wedge Y_{10} \\ X_4 = [b]X_1 & X_9 = [b']Y_1 & Y_4 = [b]Y_1 & Y_9 = [a']Y_1 \\ X_5 = X_6 \wedge X_7 & & Y_5 = Y_6 \wedge Y_8 & Y_{10} = [b']Y_1 \end{array} \right\}$$

5 Conclusions

Based on the algorithm of [5], we developed a *demand-driven* model-checking algorithm deciding the alternation-free modal mu-calculus for context-free processes. The key to this approach was to adopt the idea of demand-drivenness developed in interprocedural data-flow analysis to model checking. Though the details are different, the resulting algorithm combines advantages known from local model checking, i.e., avoiding the investigation of certain irrelevant portions of a process, with those of demand-driven data-flow analysis, i.e., avoiding the computation of irrelevant (portions of) property transformers. Besides adding a new instrument to the orchestra of model-checking approaches, we expect that the new algorithm behaves significantly better in practice than its classical counterpart because the demand-drivenness reduces the computational effort of those parts, which are responsible for the exponentiality of the classical algorithm. An empirical confirmation of this fact would comply with practical results obtained for demand-driven data-flow analysis. Additionally, we are investigating how to adopt and generalize the idea of demand-drivenness to model checking of push-down process systems in the fashion of [6].

References

[1] J. W. Backus, F. L. Bauer, J. Green, and et al. Revised report on the algorithmic language ALGOL60. *Numer. Math.*, 4:420 – 453, 1963.

[2] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Int. Conf. on Concurrency Theory* (*CONCUR'97*), LNCS 1243, pages 135 – 150. Springer-V., 1997.

[3] J. C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhäuser, Boston, 1992.

[4] J. C. Bradfield and C. P. Stirling. Verifying temporal properties of processes. In *Proc. 1st Int. Conf. on Concurrency Theory* (*CONCUR'90*), LNCS 458, pages 115 – 125. Springer-V., 1990.

[5] O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. 3rd Int. Conf. on Concurrency Theory* (*CONCUR'92*), LNCS 630, pages 123 – 137. Springer-V., 1992.

[6] O. Burkart and B. Steffen. Pushdown processes: Parallel composition and model checking. In *Proc. 5th Int. Conf. on Concurrency Theory* (*CONCUR'94*), LNCS 836, pages 98 – 113. Springer-V., 1994.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. IBM Workshop on Logic of Programs* (*LoP'81*), LNCS 131, pages 52 – 71. Springer-V., 1981.

[8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics–based verification tool for finite state systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36 – 72, 1993.

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Prog. Lang. Syst.*, 19(6):992 – 1030, 1997.

[10] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proc. 1st IEEE Annual Symp. on Logic in Computer Science* (*LICS'86*), pages 267 – 278. IEEE Computer Society, 1986.

[11] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering* (*FSE'95*), pages 104 – 115, 1995.

[12] H. Hungar and B. Steffen. Local model checking for context-free processes. In *Proc. 20th Colloquium on Automata, Languages, and Programming* (*ICALP'93*), LNCS 700, pages 593 – 605. Springer-V., 1993.

[13] J. Knoop. *Optimal Interprocedural Program Optimization: A new Framework and its Application.* PhD thesis, Univ. of Kiel, Germany, 1993. LNCS Tutorial 1428, Springer-V., 1998.

[14] R. Mayr. Strict lower bounds for model checking BPA. *ENTCS*, 18:12 pages, 1998.

[15] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[16] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. on Programming* (*SoP'82*), LNCS 137, pages 337 – 351. Springer-V., 1982.

[17] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proc. 5th Int. Conf. on Compiler Construction* (*CC'94*), LNCS 786, pages 389 – 403. Springer-V., 1994.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189 – 233. Prentice Hall, Englewood Cliffs, NJ, 1981.

[19] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *Proc. 6th Int. Conf. on Concurrency Theory* (*CONCUR'95*), LNCS 962, pages 72 – 87. Springer-V., 1995. Invited contribution.

[20] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. In *Proc. 3rd Int. Joint Conf. on Theory and Practice of Software Development* (*TAPSOFT'89*), LNCS 351, pages 369 – 383. Springer-V., 1989.

[21] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *TCS*, 89(1):161 – 177, 1991.

[22] I. Walukiewicz. Pushdown processes: Games and model checking. In *Proc. 8th Int. Workshop on Computer Aided Verification* (*CAV'96*), LNCS 1102, pages 62 – 74. Springer-V., 1996.

Programming the Mobility Behaviour of Agents by Composing Itineraries

Seng Wai Loke[1], Heinz Schmidt[2], and Arkady Zaslavsky[2]

[1] CRC for Enterprise Distributed Systems Technology
[2] School of Computer Science and Software Engineering
Monash University, Caulfield VIC 3145, Australia
swloke@dstc.monash.edu.au, hws@dstc.monash.edu.au,
A.Zaslavsky@csse.monash.edu.au

Abstract. We present an algebra for programming the itineraries of mobile agents. The algebra contains operators for modelling sequential, parallel, nondeterministic, and conditional mobility behaviour. Iterative behaviour is also modelled by a language of regular itineraries borrowing ideas from regular expressions. We give an operational semantics for the operators using Plotkin-style transition rules and provide examples of itineraries for meeting scheduling, sales order processing, and network modelling.

1 Introduction

Mobile agents can be regarded as software components which can move from one host to another to perform computations. Mobile agents have many benefits for distributed computing such as reducing network load by moving computation to data and asynchronous computation [5]. With the growing popularity of the mobile agent paradigm (e.g., [12, 10]) and the increasing ubiquity of networked computers, we envision future mobile agent applications involving complex movements of agents over large scale environments with hundreds or thousands of nodes within and between enterprises and homes. Programming the mobility behaviour of these agents would be a challenge.

Recent work such as Java-based Moderator Templates (JMTs) [7] which enable agent itineraries or plans to be composed and coordination patterns [13] provide abstractions for the mobility behaviour of agents. This paper aims to take such work further by providing an algebra of *itineraries* for reasoning with and programming the mobility behaviour of mobile agents. The goals of this language are fourfold:

– Encourage *separation of concerns* to simplify mobile agent programming. The mobility aspect of a group of agents are abstracted away from code details implementing the computations the agents are to perform on hosts. Herein lies a similarity between our approach to mobile agent programming and the emerging paradigm of aspect-oriented programming (AOP) [4].
– Provide *top-level structuring* of a mobile agent application.
– Provide *economy of expression* of mobility behaviour. The programmer expresses behaviour such as "move agent A to place p and perform action a" in a simple direct succinct manner without the clutter of the syntax of a full programming language.
– Facilitate *identification and reuse of patterns* in mobility behaviour.

The rest of this paper is organized as follows. In §2, we describe the mobile agent model on which our work is based. In §3, we describe the syntax and operational semantics of agent itineraries, and discuss two example itineraries. In §4, we discuss allowing the destination of agents to be dynamically determined by operations, and give an itinerary for network modelling as example. In §5, we present related work and conclude in §6.

2 Mobile Agent Model

A mobile agent architecture consists of *agents* and *places*. A place receives agents and is an environment where an agent executes. A place has an address which we call a *place address*. Typically, as in many agent libraries, a place is a server which can be addressed using a hostname and a port number (e.g., `www.dstc.edu.au:888`).

We use an object-oriented model of agents. We assume that an agent is an instance of a class, and roughly, we define a mobile agent as follows:

mobile agent = state + action + mobility

State refers to an agent's state (values of instance variables) possibly including a reflection of the agent's context. Action refers to operations the agent performs to change its state or that of its context. Mobility comprises all operations modifying an agent's location, including moving its state and code to other than the current location. While mobility assumes that an agent moves at the agent's own volition, the itineraries may be viewed as a specification or plan of agent movements. A community of agents is considered to implement the movement plan correctly, if their observed behaviour corresponds to (i.e., "simulates") the plan. This links our calculus with other more general calculi for concurrent objects, such as the pi-calculus.

We assume that the agents have the capability of *cloning*, that is, creating copies of themselves with the same state and code. We also assume that agents can communicate to synchronize their movements, and an agent's code is runnable in each place it visits.

3 An Algebra of Itineraries

Let $\mathbf{A}^{\{0,1\}*}$ (where $\{0,1\}*$ denotes strings of zero or more zeroes or ones), $\mathbf{O}$ and $\mathbf{P}$ be finite sets of agent, action and place symbols, respectively. In expressions we use letters $A, B, C, \ldots \in \mathbf{A}^{\{0,1\}*}$, $a, b, c, \ldots \in \mathbf{O}$ and $p, q, r, \ldots \in \mathbf{P}$. Also, we use the following convention for generating names for agents and their clones. When an agent A is cloned, agent A is renamed A^0 and its clone is named A^1. Similarly, when A^0 is cloned, we have $(A^0)^0$ (or A^{00}) and $(A^0)^1$ (or A^{01}), and for A^1, we have A^{10} and A^{11}, and so on. More generally, to any $A \in \mathbf{A}^{\{0,1\}*}$, we add superscripts 0 and 1 to form A^0 and A^1. Note that with this convention, the name of the original agent is in $\mathbf{A}^{\{0\}*}$. For example, A^{000} and B^{000} denote original agents.

Itineraries (denoted by $\mathcal{I}$) are now formed as follows representing the null activity, atomic activity, parallel, sequential, nondeterministic, conditional nondeterministic behaviour, and have the following syntax:

$$\mathcal{I} ::= \mathbf{0} \;\mid\; A_p^a \;\mid\; A_l^a \;\mid\; (\mathcal{I} \parallel \mathcal{I}) \;\mid\; (\mathcal{I} \cdot_{\oplus} \mathcal{I}) \;\mid\; (\mathcal{I} \mid \mathcal{I}) \;\mid\; (\mathcal{I} :_{\Pi} \mathcal{I})$$

where $A \in \mathbf{A}^{\{0,1\}*}$, $a \in \mathbf{O}$, $p \in \mathbf{P}$, $\mathbf{PO} \subseteq \mathbf{P} \to \mathbf{P}$ is a set of placement operations described in §4, $l \in L_{\mathbf{PO}}$ which is the language of regular itineraries described in §4, $\oplus$ is an operator which combines an agent with its clone to form a new agent, and Π is an operator which returns a boolean value to model conditional behaviour. Whenever $\oplus$ and Π are not part of current dicussions, for simplicity, we write "$\cdot$" and ":".

Below, we give an operational semantics to the above operators.

3.1 Operational Semantics

We first define $ags(I)$ to mean the set of agents (symbols) occurring in the itinerary I. $ags(I)$ is defined inductively as the minimal subset of $\mathbf{A}^{\{0,1\}*}$ satisfying: $ags(\mathbf{0}) = \emptyset$, $ags(A_p^a) = \{A\}$, $ags(A_l^a) = \{A\}$, $ags(I \cdot J) = ags(I) \cup ags(J)$, $ags(I \parallel J) = ags(I) \cup ags(J)$, $ags(I \mid J) = ags(I) \cup ags(J)$, and $ags(I : J) = ags(I) \cup ags(J)$.

We also define *configuration* which is a relation involving agents, actions and places, denoted by Σ (and by Σ', Σ'' ...), where $\Sigma \subseteq \mathbf{A}^{\{0,1\}*} \times \mathbf{O} \times \mathbf{P}$. Given a collection of agents, the configuration represents the location and action of each agent. A constraint on a configuration is that it defines a function from $\mathbf{A}^{\{0,1\}*}$ to $\mathbf{P}$, the same agent cannot be in two places in the same configuration.

The operational semantics is given by Plotkin-style rules defining transitions from one configuration to another of the form $\frac{premises}{conclusion}[condition]$. Declaratively, the *conclusion* holds if the *premises* and *condition* hold. Procedurally, to establish the *conclusion*, establish the *premises* and then check that *condition* holds. We write $\Sigma \xrightarrow{I} \Sigma'$ to mean an itinerary I causes a transition from configuration Σ to configuration Σ'.

We assume that all agents in an itinerary have a starting place (which we call the agent's *home*) denoted by *h* $\in \mathbf{P}$. Given a collection of agents $Agts$, we define a distinguished configuration called the *home configuration* denoted by Σ_h^{Agts}. The home configuration is the configuration where all the agents are at home. For example, the home configuration for a collection of agents A, B and C is:

$$\{(A, id, h), (B, id, h), (C, id, h)\}$$

Note that the operations at home are the identity action $id \in \mathbf{O}$. Also, all agents used in an itinerary must be mentioned in the home configuration, i.e. none of the rules which follows can add new agents unless they are clones of existing agents.

Agent Movement (A_p^a). A_p^a means "move agent A to place p and perform action a". This expression is the smallest granularity mobility abstraction. It involves one agent, one move and one action at the destination. The underlying mobility mechanisms are hidden. So are the details of the action which may change the agent state or the context in which it is operating at the destination place:

$$a : states(A) \times states(p) \to states(A) \times states(p)$$

In our agent model, each action is a method call of the class implementing A. The implementation must check that a is indeed implemented in A.

$\mathbf{0}$ represents, for any agent A, the empty itinerary A_{here}^{id}, where the agent performs the identity operation $id \in \mathbf{O}$ on the state at its current place $here$.

The transition rule for agent movement replaces a triple from a configuration Σ:

$$\frac{}{\Sigma \xrightarrow{A_p^a} (\Sigma \setminus \{(A,b,q)\}) \cup \{(A,a,p)\}} \qquad (1)$$

For example, applying this rule to the home configuration with the expression A_p^a gives a new configuration with only the location and action of A changed:

$$\{(A,id,h),(B,id,h),(C,id,h)\} \xrightarrow{A_p^a} \{(A,a,p),(B,id,h),(C,id,h)\}$$

Parallel Composition ("$\|$"). Two expressions composed by "$\|$" are executed in parallel. For instance, $(A_p^a \parallel B_q^b)$ means that agents A and B are executed concurrently. Parallelism may imply cloning of agents. For instance, to execute the expression $(A_p^a \parallel A_q^b)$, where $p \neq q$, cloning is needed since agent A has to perform actions at both p and q in parallel. In the case where $p = q$, the agents are cloned as if $p \neq q$. In general, given an itinerary $(I \parallel J)$ the agents in $ags(I) \cap ags(J)$ are cloned and although having the same name are different agents. In the transition rule below, we show how clones are distinguished for the purposes of specifying the operational semantics.

Before presenting the rule, given $Agts \subseteq \mathbf{A}^{\{0,1\}*}$ and a configuration Σ, we define the operator "$-$" which removes triples mentioned in $Agts$ from Σ:
$\Sigma \;-\; Agts = \Sigma \setminus \{(A,a,p) \mid A \in Agts, (A,a,p) \in \Sigma\}$

The transition rule for parallel composition is as follows:

$$\frac{renamed^0(\Sigma) \xrightarrow{renamed^0(I)} \Sigma' \quad \wedge \quad renamed^1(\Sigma) \xrightarrow{renamed^1(J)} \Sigma''}{\Sigma \xrightarrow{I \| J} \Sigma' \cup \Sigma''} \qquad (2)$$

$renamed^0(\Sigma)$ is Σ without the triples concerning agents in $ags(J)$ and with agents in $ags(I) \cap ags(J)$ renamed: $renamed^0(\Sigma) = (\Sigma - ags(J)) \cup \{(A^0,a,p) \mid A \in (ags(I) \cap ags(J)), (A,a,p) \in \Sigma\}$. Similarly, $renamed^1(\Sigma) = (\Sigma - ags(I)) \cup \{(A^1,a,p) \mid A \in (ags(I) \cap ags(J)), (A,a,p) \in \Sigma\}$.

This means that agents to be cloned (i.e. those mentioned in $ags(I) \cap ags(J)$) and clones are renamed apart. With the naming of clones, when the resulting configurations are combined in $\Sigma' \cup \Sigma''$, the clones retain their identities.

We use $renamed^0(I)$ to denote the corresponding renaming of I (defined recursively on the algebra): $renamed^0(\mathbf{0}) = \mathbf{0}$, $renamed^0(A_p^a) = A_p^{0\ a}$ if $A \in ags(I) \cap ags(J)$, $renamed^0(A_p^a) = A_p^a$ if $A \notin ags(I) \cap ags(J)$, $renamed^0(A_l^a) = A_l^{0\ a}$ if $A \in ags(I) \cap ags(J)$, $renamed^0(A_l^a) = A_l^a$ if $A \notin ags(I) \cap ags(J)$, $renamed^0(I \cdot J) = renamed^0(I) \cdot renamed^0(J)$, $renamed^0(I \parallel J) = renamed^0(I) \parallel renamed^0(J)$, $renamed^0(I \mid J) = renamed^0(I) \mid renamed^0(J)$, and $renamed^0(I : J) = renamed^0(I) : renamed^0(J)$. $renamed^1$ is defined similarly.

Parallel composition splits the configuration Σ into $renamed^0(\Sigma)$ and $renamed^1(\Sigma)$ each acted on separately by I and J respectively. An example illustrates this.

Let $\Sigma = \{(A,a,p),(B,b,q),(C,c,r),(D,d,s)\}$, $I = A_t^e \cdot B_v^g$, and $J = A_u^f \cdot C_w^h$. Then, $renamed^0(\Sigma) = \{(A^0,a,p),(B,b,q),(D,d,s)\}$, $renamed^1(\Sigma) = \{(A^1,a,p),(C,c,r),(D,d,s)\}$, $renamed^0(I) = A_t^{0\ e} \cdot B_v^g$, and $renamed^1(J) =$

${A_u^1}^f \cdot C_w^h$. And so, $\Sigma' = \{(A^0, e, t), (B, g, v), (D, d, s)\}$ and $\Sigma'' = \{(A^1, f, u), (C, h, w), (D, d, s)\}$ (by the semantics of "$\cdot$" given below). The resulting configuration is $\Sigma' \cup \Sigma'' = \{(A^0, e, t), (A^1, f, u), (B, g, v), (C, h, w), (D, d, s)\}$ which contains A^0 and its clone A^1. In the next operator, combining of clones is carried out.

Sequential Composition ("$\cdot$"). Two expressions composed by the operator "$\cdot$" are executed sequentially. For example, $(A_p^a \cdot A_q^b)$ means move agent A to place p to perform action a and then to place q to perform action b. Sequential composition is used when order of execution matters. In the example, state changes to the agent from performing a at p must take place before the agent goes to q.

Sequential composition imposes synchronization among agents. For example, in the expression $(A_p^a \parallel B_q^b) \cdot C_r^c$ the composite action $(A_p^a \parallel B_q^b)$ must complete before C_r^c starts. Implementation of such synchronization requires message-passing between agents at different places or via shared memory abstractions.

When cloning has occurred, sequential composition performs decloning, i.e. clones are combined. For example, given the expression $(A_s^d \parallel A_t^e) \cdot A_u^f$ and suppose that after the parallel operation, the configuration has clones. Then, decloning is carried out before continuing with A_u^f.

The transition rule for sequential composition is:

$$\frac{\Sigma \xrightarrow{I} \Sigma' \wedge decloned(\Sigma') \xrightarrow{J} \Sigma''}{\Sigma \xrightarrow{I \cdot J} \Sigma''} \tag{3}$$

where $decloned(\Sigma')$ is the configuration Σ' with all clones substituted by their combinations.

We now define $decloned$. When two clones A^{x0} and A^{x1}, for some $x \in \{0,1\}*$, are combined, we name the combined agent A^x. We denote the combination operation by $\oplus : (\mathbf{A}^{\{0,1\}*} \times \mathbf{A}^{\{0,1\}*}) \to \mathbf{A}^{\{0,1\}*}$. The semantics of $\oplus$, i.e. how the states and code of the clones are combined is left to the implementation. Also, the place where an agent is combined with its clone is the agent's place. The clone which is not at that location will move to that location before combination takes place. For example, when combining A^{x0} and A^{x1}, A^x resides at A^{x0}'s place and A^{x1} must move to that place before the combination takes place. Let Σ and Σ' be configurations such that $decloned(\Sigma) = \Sigma'$. Then Σ' is computed by the following algorithm where, in each iteration, two clones are replaced by their combination until no combinations are possible:

1. If there exists $(A^{x0}, a, p), (A^{x1}, b, q) \in \Sigma$, for some $x \in \{0,1\}*$, then (2), else $\Sigma' := \Sigma$ (finish).
2. Modify Σ: $\Sigma := (\Sigma \setminus \{(A^{x0}, a, p), (A^{x1}, b, q)\}) \cup \{(A^x, id, p)\}$ where $A^x = A^{x0} \oplus A^{x1}$. Go to (1).

For example, decloning the configuration $\{(A^0, a, p), (A^{10}, b, q), (A^{11}, c, r), (B, d, s)\}$ gives the following result after the first iteration: $\{(A^0, a, p), (A^1, id, q), (B, d, s)\}$ and after the second (final) iteration, we have: $\{(A, id, p), (B, d, s)\}$. The combined agent has the identity operation id at p.

We can associate the operator "$\oplus$" with "$\cdot$" by writing "$\cdot_\oplus$".

In the expression $(A_s^d \parallel A_t^e) \cdot A_u^f$, cloning could take place at the following destination u. However, in other expressions such as $(A_s^d \parallel A_t^e) \cdot (A_u^f \parallel A_v^g)$ there are two possible destinations u and v. Also, note that since the clones have the same names at this level (they are given distinguished names only for the purposes of specifying their operational semantics), it is also unclear from the expression which agent (the original or its clone) goes to u or to v. Our operational semantics solves this ambiguity by viewing the above expression as causing a cloning (in the first parallel composition), a combination (in the sequential composition), and another cloning (in the second parallel composition).

Independent Nondeterminism ("$|$"). An itinerary of the form $(I \mid J)$ is used to express nondeterministic choice: "I don't care which but perform one of I or J". If $ags(I) \cap ags(J) \neq \emptyset$, no clones are assumed, i.e. I and J are treated independently. It is an implementation decision whether to perform both actions concurrently terminating when either one succeeds (which might involve cloning but clones are destroyed once a result is obtained), or trying one at a time (in which case order may matter).

Two transition rules are used to represent the nondeterminism:

$$\frac{\Sigma \xrightarrow{I} \Sigma'}{\Sigma \xrightarrow{I|J} \Sigma'} \quad (4a) \qquad\qquad \frac{\Sigma \xrightarrow{J} \Sigma''}{\Sigma \xrightarrow{I|J} \Sigma''} \quad (4b)$$

These rules show that $I \mid J$ leads to one of two possible configurations.

Conditional Nondeterminism ("$:$"). Independent nondeterminism does not specify any dependencies between its alternatives. We introduce conditional nondeterminism which is similar to short-circuit evaluation of boolean expressions in programming languages such as C.

We first introduce status flags and *global state function*:

– A status flag is always part of the agent's (say, A's) state, written as $A.status$. Being part of the state, $A.status$ is affected by an agent's actions. $A.status$ might change as the agent performs actions at different places.
– A global state function $\Pi : \wp(\mathbf{A}^{\{0,1\}*} \times \mathbf{O} \times \mathbf{P}) \rightarrow \{true, false\}$ maps a configuration to a boolean value. Π need not be defined in terms of status flags but it is useful to do so. For example, we can define Π as the conjunction of the status flags of agents in a configuration Σ: $\quad \Pi(\Sigma) = \bigwedge_{(A,a,p)\in\Sigma} A.status$
We can view Π as producing a global status flag. From the implementation viewpoint, if a configuration involves more than one agent, these agents must communicate to compute Π.

The semantics of conditional nondeterminism depends on some given Π in the way described above. We express this dependency of "$:$" on a Π by writing "$:_\Pi$". The transition rule for "$:_\Pi$" is as follows:

$$\frac{\Sigma \xrightarrow{I} \Sigma'}{\Sigma \xrightarrow{I:_\Pi J} \Sigma'}[\Pi(\Sigma') = true] \text{ (5a)} \quad \frac{\Sigma \xrightarrow{I} \Sigma' \wedge decloned(\Sigma') \xrightarrow{J} \Sigma''}{\Sigma \xrightarrow{I:_\Pi J} \Sigma''}[\Pi(\Sigma') = false] \text{ (5b)}$$

Rule (5b) is similar to rule (3).

Note that we can introduce variations of ":" with different global state functions such as "$:_\Pi$" and "$:_{\Pi'}$", and similarly with "$\cdot$": "$\cdot_\oplus$" and "$\cdot_{\oplus'}$". The operational semantics of the operators and how $\oplus$ and Π are used is specified as above (and so, application-independent), but definitions for $\oplus$ and Π are application-specific.

Given the above rules, an itinerary I, a set of agents $ags(I)$, and home configuration $\Sigma_h^{ags(I)}$, establishing the conclusion $\Sigma_h^{ags(I)} \xrightarrow{I} \Sigma'$ (i.e. computing Σ') will form a derivation tree whose root is that conclusion, whose internal nodes are derived using the above rules and whose leaves are empty.

Algebraic Properties of Itineraries. Due to space limitations, we will not discuss the algebraic properties of the operators in detail in this paper but state without proof the following properties used in our examples:

– **associativity**: from the above definitions, sequential, parallel and both nondeterministic compositions are associative, i.e. for example, $I \cdot (J \cdot K) = (I \cdot J) \cdot K$. Note that with parallel composition, starting with the same configuration Σ, if $I \parallel (J \parallel K)$ brings Σ to Σ' and $(I \parallel J) \parallel K$ brings Σ to Σ'', then Σ' is same as Σ'', except for naming of clones (e.g., Σ' contains A^0, A^{10} and A^{11}, and Σ'' contains A^{00}, A^{01} and A^1). Associativity lets us leave some brackets out in composite expressions.
– **distributivity of "$\cdot$" over "$|$"**: $(I \mid J) \cdot K = (I \cdot K) \mid (J \cdot K)$

3.2 Two Examples

Meeting Scheduling. We use a two phase process for demonstration purposes: (1) Starting from home, the meeting initiator sends an agent which goes from one participant to another with a list of nominated times. As each participant marks the times they are not available, the list of nominated times held by the agent shortens as the agent travels from place to place. After visiting all places in its itinerary, the agent returns home. (2) At home, the meeting initiator selects a meeting time from the remaining unmarked times and informs the rest.

With four participants (excluding the initiator), the mobility behaviour is given by: $A_p^{ask} \cdot A_q^{ask} \cdot A_r^{ask} \cdot A_s^{ask} \cdot A_h^{finalize} \cdot (A_p^{inform} \parallel A_q^{inform} \parallel A_r^{inform} \parallel A_s^{inform})$
ask is an action which displays unmarked nominated times to a participant and allows a participant to mark times he/she is unavailable. $finalize$ allows the meeting initiator to select a meeting time from the remaining unmarked times, and $inform$ presents the selected meeting time to a participant. Note that the expression of mobility is separated from the coding of these three actions.

Sales Order Processing. We consider a scenario adapted from [9] for processing sales orders in a virtual enterprise. Each sales order is carried out by a mobile agent which moves through several entities to process the order. We first name the entities. Let us_sc be a place where the agent can interact with the US stock control, $asia\_sc$ be a place where the agent can interact with the Asian stock control, mat be a place where the agent can purchase raw materials for manufacturing the products requested in a sales order, man be the place where the agent can place an order for products to be manufactured (i.e., man represents the manufacturer), and ext be a place where the agent can interact with an external buyer. Also, let $query$ be an action where the agent queries a stock control, $report$ be an action where the agent reports the results of a completed sales order, buy_raw be the action of purchasing raw materials, buy_prod be the action of buying products for a sales order, and $order$ be an action of placing an order to have some products manufactured.

The business logic for processing a sales order is as follows. The agent first receives an order while at home. Then, one of the following takes place.

1. The agent checks with the US stock control to see if the requested products are available. If so, the agent returns home reporting this. We can represent this behaviour as $A^{query}_{us\_sc} \cdot A^{report}_{h}$.
2. Otherwise, the agent checks with the Asian stock control, and if the requested products are available, reports this at home. This behaviour is captured by $A^{query}_{asia\_sc} \cdot A^{report}_{h}$.
3. If the Asian stock control does not have the products available, the agent purchases raw materials for manufacturing the product and places an order for the product with the manufacturer. Thereafter, the agent reports what it has done at home. We write this behaviour as $A^{buy\_raw}_{mat} \cdot A^{order}_{man} \cdot A^{report}_{h}$.
4. Alternatively, if the agent cannot fulfill (3), for example, the raw materials are too expensive. The agent buys the products from an external buyer and reports this: $A^{buy\_prod}_{ext} \cdot A^{report}_{h}$.

In essence, there are four ways to process a sales order and we just want to perform one of them (each in the sense described above). We can capture the essence of the business logic as follows:

$$
\begin{aligned}
&(A^{query}_{us\_sc} \cdot A^{report}_{h}) \mid (A^{query}_{asia\_sc} \cdot A^{report}_{h}) \mid (A^{buy\_raw}_{mat} \cdot A^{order}_{man} \cdot A^{report}_{h}) \mid (A^{buy\_prod}_{ext} \cdot A^{report}_{h}) \\
= &(A^{query}_{us\_sc} \mid A^{query}_{asia\_sc} \mid (A^{buy\_raw}_{mat} \cdot A^{order}_{man}) \mid A^{buy\_prod}_{ext}) \cdot A^{report}_{h} \\
&\text{(by distribution of } \cdot \text{ over } |)
\end{aligned}
$$

However, the above itinerary does not model the fact that the four ways of processing a sales order are tried sequentially and the next way is used only when one way has "failed" (e.g., if the product is not in stock, then get it manufactured).

Using conditional nondeterminism, the sales order agent's behaviour is:
$(A^{query}_{us\_sc} :_{\Pi} A^{query}_{asia\_sc} :_{\Pi} (A^{buy\_raw}_{mat} \cdot A^{order}_{man}) :_{\Pi} A^{buy\_prod}_{ext}) \cdot A^{report}_{h}$
This more precisely models the business logic. The operator is non-deterministic in the sense that the resulting configuration of an operation $I :_{\Pi} J$ is either decided by (5a) or (5b). However, the decision of which rule depends on Π. For example, assuming we use the definition of Π in §3.1 which is in terms of status flags, if no stock is available at us_sc, then, $A.status$ would be set to false. Note that it is left to actions of A to properly set $A.status$ to reflect the intended purpose of the application.

4 Regular Itineraries

So far, the place p in A^{a}_{p} is assumed to be a constant. This section replaces p with computed destinations and introduces iterative behaviour produced by repeatedly applying the same computation to an agent's current location.

Placement Operation. Given a set $\mathbf{P}$ of place symbols, we define a partial function $o : \mathbf{P} \rightarrow \mathbf{P}$ which maps a place to another which we call a *placement operation*. For some places, o is undefined, i.e. returns $\perp$, and always $o(\perp) = \perp$. In place of a place constant, we can write a placement operation.

A^a_o denotes "move agent A to the place obtained by applying o to A's current position":

$$\frac{}{\Sigma \xrightarrow{A^a_o} (\Sigma \setminus \{(A,b,p)\}) \cup \{(A,a,o(p))\}} [o(p) \neq \perp] \tag{1'a}$$

In case $o(p) = \perp$, we define $A^a_\perp$ as the empty itinerary **0**, i.e. the agent A does nothing and stays at its current position:

$$\frac{}{\Sigma \xrightarrow{A^a_o} (\Sigma \setminus \{(A,b,p)\}) \cup \{(A,id,p)\}} [o(p) = \perp] \tag{1'b}$$

We consider $A.here$ denoting agent A's current location as part of the agent's state like $A.status$. This permits the agent to record its own location for its own use in its actions. For example, given the current configuration $\{(A,d,s),(A^1,e,t),(B,b,q),(C,c,r)\}$, $A.here = s$, $A^1.here = t$, $B.here = q$, $C.here = r$. Note that the value of $A.here$ for any agent A might be changed by itineraries involving A but always has the initial value of h.

We do not specify how o computes its value which is application-specific, but note that evaluation of o occurs at the agent's current location. Placement operations are intended for capturing the idea that the agent's destination could be dynamically computed at run-time rather than declared statically as in the previous section.

Itineraries involving mix of place symbols and placement operations is admitted, such as $A^a_p \cdot A^a_o$, which moves agent A to p and then to $o(p)$.

We also define operators to allow placement operations to be combined. The operators are similar to those in regular expressions as we show below.

Placement Language. Let $\mathbf{PO} \subseteq (\mathbf{P} \rightarrow \mathbf{P})$ be a set of placement operations. We define a *placement language* $L_{\mathbf{PO}}$ representing combinations of placement operations in $\mathbf{PO}$ as follows. The members of a placement language are called *regular itineraries*.

Let $l_1, l_2 \in L_{\mathbf{PO}}$. Then $\mathbf{PO} \subseteq L_{\mathbf{PO}}$, i.e. $L_{\mathbf{PO}}$ contains all placement operations in $\mathbf{PO}$, $(l_1 \parallel l_2) \in L_{\mathbf{PO}}$, $(l_1 \cdot l_2) \in L_{\mathbf{PO}}$, $(l_1 \mid l_2) \in L_{\mathbf{PO}}$, $(l_1 : l_2) \in L_{\mathbf{PO}}$, ${l_1}^n \in L_{\mathbf{PO}}$, $l_1 * \in L_{\mathbf{PO}}$, and no other strings are in $L_{\mathbf{PO}}$.

The semantics of the above operators for placement operations are induced by that for the operators for itineraries. Also, we always apply the regular itineraries on $A.here$. Let $l_1, l_2 \in L_{\mathbf{PO}}$, $A \in \mathbf{A}$, $a \in \mathbf{O}$. Then, $A^a_{l_1 \parallel l_2} = A^a_{l_1} \parallel A^a_{l_2}$, $A^a_{l_1 \cdot l_2} = A^a_{l_1} \cdot A^a_{l_2}$, $A^a_{l_1 | l_2} = A^a_{l_1} \mid A^a_{l_2}$, $A^a_{l_1 : l_2} = A^a_{l_1} : A^a_{l_2}$, and $A^a_{{l_1}^n} = A^a_{\underbrace{l_1 \cdot \ldots \cdot l_1}_{n \text{ times}}}$.

Also, let $f^n(x)$ be n repeated applications of function f on x (e.g., $f^3(x) = (f.f.f)(x) = f(f(f(x)))$, where "." is function composition), $l \in L_{\mathbf{PO}}$ be some sequential composition of placement operations (i.e., $l = o_1 \cdot o_2 \cdot \ldots o_k$, where $k \in \mathbb{Z}^+$ and $o_i \in \mathbf{PO}$ for $1 \leq i \leq k$), and $rew(l)$ be a rewriting of l in reverse by replacing "$\cdot$"

with "." (e.g. $rew(o_1 \cdot o_2 \cdot o_3) = o_3.o_2.o_1$). Then, we define A^a_{l*} as $A^a_{l*} = A^a_{\underbrace{l\cdot\ldots\cdot l}_{n \text{ times}}}$, where n is the smallest integer such that $rew(l)^{(n+1)}(x) = \perp$, and x is the current location of A just before the start of the operation A^a_{l*}. For example, if the current location of A is p, $l = o_1 \cdot o_2$, $o_2(o_1(p)) \neq \perp$, $o_2(o_1(o_2(o_1(p)))) \neq \perp$ and $o_2(o_1(o_2(o_1(o_2(o_1(p)))))) = \perp$, then $rew(l) = o_2.o_1$, and $(o_2.o_1)^3(p) = \perp$, and so, $A^a_{l*} = A^a_{(o_1\cdot o_2)\cdot(o_1\cdot o_2)}$. If there is no such n, we define $A^a_{l*} = \mathbf{0}$. In addition, we define $A^a_{l*} = \mathbf{0}$ for l not a sequential composition of placement operations since expressions like $(o_1 \parallel o_2)(p)$ are not defined as functions over **P**, though we have lifted sequential composition to function composition using rew. We can think of l^n and $l*$ as iterators over the places obtained from repeatedly applying l.

An Example: Network Modelling. In [14], mobile agents are applied to network modelling, where the agent visits a collection of nodes identifying those nodes satisfying some criteria (e.g., host with CPU utilization > 90% and has less than 1GB of disk space). The agent which is injected into the network carries a pre-configured itinerary describing a traversal of a collection of nodes.

Let $\mathbf{P}$ be the collection of nodes (places), $next : \mathbf{P} \to \mathbf{P}$ be the placement operation which given a node returns the next node in the traversal, A be the agent, and $test$ be the action performed by the agent on a node to determine if the node satisfies a specified criteria. Nodes passing the test are recorded by the agent. The last node p_{last} in the traversal is modelled by $next(p_{last}) = \perp$. We send the agent home after it has completed its traversal which reports its results via the operation $report$. The agent's itinerary is represented by: $A^{test}_{next*} \cdot A^{report}_{home}$.

We can extend the above itinerary so that the agent traverses three different domains, each domain a collection of nodes, and two adjacent domains connected by a gateway node. Let $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$ be the collection of nodes (places) in each domain, $next_{\mathbf{P}} : \mathbf{P} \to \mathbf{P}$, $next_{\mathbf{Q}} : \mathbf{Q} \to \mathbf{Q}$, $next_{\mathbf{R}} : \mathbf{R} \to \mathbf{R}$ be placement operations for each respective domain, pq be the gateway node between domains P and Q and qr be the gateway node between domains Q and R, and sec be an action where the agent requests permission to cross into another domain. Then, the agent's extended itinerary is as follows: $A^{test}_{next_{\mathbf{P}}*} \cdot A^{sec}_{pq} \cdot A^{test}_{next_{\mathbf{Q}}*} \cdot A^{sec}_{qr} \cdot A^{test}_{next_{\mathbf{R}}*} \cdot A^{report}_{h}$

5 Related Work

In many Java mobile agent toolkits such as IBM Aglets [7], a sequential itinerary can be defined as a separate object, where each element of the itinerary is a pair: (place address, method call). Such itineraries can be defined using our "·" operator.

Emerging work attempts to provide further aid for programming more complex agent itineraries. For example, JMTs mentioned in §1 enable agent itineraries or plans to be composed where the plans may involve parallelism, cloning and combining of clones. We contend that our itineraries provide greater economy of expression for representing mobility behaviour. Moreover, although we have adopted an object-based agent model, our itineraries are programming language independent - actions could have been written in C, say. However, JMT would provide a convenient platform for implementing our itineraries with Java.

The coordination patterns for information agents described in [13] have more details about how they are to be used but are presented less formally than our itineraries. Finite state machines (FSMs) are used to represent agent itineraries in [6]. Each state of the machine represents a task which is either some computation at a place or a movement to another place. Sequential and nondeterministic dependencies between tasks are expressed conveniently with FSMs. In [2], interaction diagrams are used to represent agent mobility behaviour, vertical lines represent places and arrows between these lines represent movements of agents between places. Such a visual notation is highly appealing for non-programmers. In the PLANGENT mobile agent system [8], plans are dynamically created from rules expressed in the declarative KIF language. Their advantage is dynamic planning but the plans are sequences of tasks: parallelism is not expressed. In contrast to these techniques, we give a formal and compositional approach to itineraries.

The Pict language [11] derived from the Pi-Calculus and the Join-Calculus language [3] derived from a variant of the Pi-Calculus both contain a parallel composition operator for concurrently executing processes. However, Pict does not have the concept of agent places - where processes execute is not explicit. Join-Calculus has the concept of addressable locations corresponding to agent places. The locations are themselves mobile and have embedded code, and so, locations also correspond to mobile agents. However, an individual agent view of mobility is employed: a location's code contains a primitive `go(address)` which moves the location itself to another location at `address`. In contrast, our itineraries take a top-level ("God-view") of mobile agents where mobility is expressed outside of the agent's actions. Moreover, Join-Calculus does not have implicit cloning.

The ambient calculus [1] focuses on movements across administrative domains and boundaries. An ambient is a named location which may contain processes or subambients. Ambient's capabilities are moving inside or outside other ambients, and removing the boundary of an ambient exposing its contents. There are operators for replicating and parallel composition of processes and ambients. In contrast to the ambient calculus, our algebra focuses on itineraries of movements. We have assumed a flat space of places (modelled simply as a set $\mathbf{P}$). We could explore a more structured space of places by modelling $\mathbf{P}$ as an ambient with subambients.

6 Conclusion

We have presented an algebra of itineraries for programming the mobility aspect of agents, and illustrated our operators via examples in three scenarios: meeting scheduling, sales order processing and network modelling. We are exploring templates for itinerary reuse, and extensions to atomic movement (given in §3): (1) When an agent moves from p to q, it goes from p to a mediator place s and then to q. The advantage of this is greater fault tolerance and allowance for mobile places: the agent can wait at s for q to reconnect to s. Similarly, we can also model domain crossings if the mediator place is a gateway. (2) Optimise bandwidth usage by first moving the agent's state over to the destination, and then, only transfer the code if the agent's code for actions is not already available at the destination. If places run on small devices such as PDAs, a further optimization is not to send the entire agent's code but only code required for the specified action. (3) Generalize our place-level units to domain level units each consisting of an agent group - a collection of mobile agents perhaps with a leader agent, a domain - an Intranet consisting of a network of places, and a domain action - not a method call but refers to the agent group's action in the domain. For instance, in the following expression, an agent group G moves sequentially through domains P, Q, and R collecting the information from each domain: $G_P^{col\_all} \cdot G_Q^{col\_all} \cdot G_R^{col\_all}$.

Acknowledgements. The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science & Tourism of the Commonwealth Government of Australia.

References

[1] L. Cardelli and A.D. Gordon. Mobile Ambients. *LNCS*, 1378, 1998.

[2] B. Falchuk and A. Karmouch. Visual Modeling for Agent-Based Applications. *Computer*, 31(12):31–38, December 1998.

[3] C. Fournet and L. Maranget. The Join-Calculus Language Release 1.04: Documentation and User's Manual. Available from <http://pauillac.inria.fr/cdrom_a_graver/www/join/manual/index.html>, January 1999.

[4] G. Kiczales. Aspect Oriented Programming. *ACM SIGPLAN Notices*, 32(10):162–162, October 1997.

[5] D.B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *CACM*, 42(3):88–89, March 1999.

[6] R.P. Lentini, G.P. Rao, and J.N. Thies. Agent Itineraries. *Dr. Dobb's Journal of Software Tools*, 24(5):60, 62, 64, 66, 68, 70, May 1999.

[7] K. Minami and T. Suzuki. JMT (Java-Based Moderator Templates) for Multi-Agent Planning. In *OOPSLA'97 Workshop: Java-based Paradigms for Agent Facilities*, 1997. Available at <http://www.trl.ibm.co.jp/aglets/jmt/oopsla97/jmt-oopsla97.html>.

[8] A. Ohsuga, Y. Nagai, Y. Irie, M. Hattori, and S. Honiden. PLANGENT: An Approach to Making Mobile Agents Intelligent. *IEEE Internet Computing*, 1(4):50–57, 1997.

[9] T. Papaioannou and J. Edwards. Manufacturing System Performance and Agility: Can Mobile Agents Help? *Special Issue of Integrated Computer-Aided Engineering (to appear)*, 1999. Available at <http://luckyspc.lboro.ac.uk/Docs/Papers/Icae98.pdf>.

[10] T. Papaioannou and N. Minar, editors. *Workshop on Mobile Agents in the Context of Competition and Co-operation at Agents'99*, May 1999. Available at <http://mobility.lboro.ac.uk/MAC3/>.

[11] B.C. Pierce and D.N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[12] K. Rothermel and F. Hohl, editors. *MA '98: Proceedings of the 2nd International Workshop on Mobile Agents*, number 1477 in LNCS. Springer-Verlag, September 1998.

[13] R. Tolksdorf. Coordination Patterns of Mobile Information Agents. *Cooperative Information Agents II, LNCS 1435*, pages 246–261, 1998. Available at <http://www.cs.tu-berlin.de/˜tolk/papers/cia98.ps.gz>.

[14] T. White, B. Pagurek, and A. Bieszczad. Network Modeling for Management Applications Using Intelligent Mobile Agents. *Journal of Network and Systems Management - Special Issue on Mobile Agents (to appear)*, September 1999. Available at <ftp://ftp.sce.carleton.ca/pub/netmanage/jnsm98-draft.zip>.

Faster Model Checking for Open Systems

Madhavan Mukund[1]⋆, K. Narayan Kumar[1], and Scott A. Smolka[2]

[1] Chennai Mathematical Institute, 92 G.N. Chetty Road, Chennai 600 017, India.
{madhavan,kumar}@smi.ernet.in
[2] Department of Computer Science,
State University of New York at Stony Brook, NY 11794-4400, USA.
sas@cs.sunysb.edu.

Abstract. We investigate OrEx, a temporal logic for specifying open systems. Path properties in OrEx are expressed using ω-regular expressions, while similar logics for open systems, such as ATL* of Alur et al., use LTL for this purpose. Our results indicate that this distinction is an important one. In particular, we show that OrEx has a more efficient model-checking procedure than ATL*, even though it is strictly more expressive. To this end, we present a single-exponential model-checking algorithm for OrEx; the model-checking problem for ATL* in contrast is provably double-exponential.

1 Introduction

Reactive systems are computing systems where the computation consists of an ongoing interaction between components. This is in contrast to functional systems whose main aim is to compute sets of output values from sets of input values. Typical examples of reactive systems include schedulers, resource allocators, and process controllers.

Pnueli [Pnu76] proposed the use of linear-time temporal logic (LTL) to specify properties of reactive systems. A system satisfies an LTL specification if all computations of the system are models of the formula. Later, branching-time temporal logics such as CTL and CTL* [CE81, EH86] were developed, which permit explicit quantification over computations of a system. For both linear-time and branching-time logics, the *model checking* problem—verifying whether a given system satisfies a correctness property specified as a temporal logic formula—has been well studied. Model checkers such as Spin for LTL [Hol97] and SMV for CTL [McM93] are gaining acceptance as debugging tools for industrial design of reactive systems.

Alur et al. [AHK97] have argued that linear-time and branching-time temporal logics may not accurately capture the spirit of reactive systems. These logics treat the entire system as a *closed* unit. When specifying properties of reactive systems, however, it is more natural to describe how different parts of the system behave when placed in *contexts*. In other words, one needs to specify properties of *open* systems that interact with environments that cannot be controlled.

⋆ Partly supported by IFCPAR Project 1502-1.

A fruitful way of modelling open systems is to regard the interaction between the system and its environment as a game. The system's goal is to move in such a way that no matter what the environment does, the resulting computation satisfies some desired property. In this formulation, the verification problem reduces to checking whether the system has a winning strategy for such a game.

The *alternating temporal logic* ATL* proposed in [AHK97] reflects the game-theoretic formulation of open systems. In ATL*, the property to be satisfied in each run of the system is described using an LTL formula, and LTL formulas may be nested within game quantifiers. The quantifiers describe the existence of winning strategies for the system. Moreover, a restricted version of ATL* called ATL has an efficient model-checking algorithm. However, the model-checking problem for the considerably more expressive logic ATL* is shown to be 2EXPTIME-complete which compares unfavourably with the PSPACE-completeness of CTL*.

In this paper, we propose a logic for open systems called $\text{O}\textsc{r}_\text{E}\textsc{x}$, which is interpreted over Σ-labelled transition systems and uses ω-regular expressions rather than LTL formulas to specify properties along individual runs. As in ATL*, ω-regular expressions may be nested within game quantifiers. Since ω-regular expressions are more expressive than temporal logics for specifying properties of infinite sequences, our logic is more expressive than ATL*. Moreover, it turns out that $\text{O}\textsc{r}_\text{E}\textsc{x}$ has an exponential-time model checking algorithm, which is considerably better than the complexity of ATL*.

The paper is organized as follows. Section 2 contains some basic definitions about transition systems and games. Section 3 introduces the syntax and semantics of $\text{O}\textsc{r}_\text{E}\textsc{x}$ and gives some examples of how to write temporal specifications in the logic. Section 4 describes an automata-theoretic algorithm for $\text{O}\textsc{r}_\text{E}\textsc{x}$'s model-checking problem. Section 5 discusses related work and Section 6 offers our concluding remarks.

2 Transition Systems, Games, Strategies

Transition systems A *Σ-labelled transition system* is a tuple $TS = \langle X, \Sigma, \longrightarrow\rangle$ where X is a finite set of *states*, Σ is a finite alphabet of *actions* and $\longrightarrow \subseteq X \times \Sigma \times X$ is the *transition relation*. We use $x, y, \ldots$ to denote states and $a, b, \ldots$ to denote actions.

An element $(x, a, y) \in \longrightarrow$ is called an a-transition. For $S \subseteq \Sigma$, the set of S-transitions is given by $\longrightarrow_S = \{(x, a, y) \mid a \in S\}$. As usual, we write $x \xrightarrow{a} y$ to denote that $(x, a, y) \in \longrightarrow$. The set of transitions enabled at a state x, denoted *enabled*(x), is the set of transitions of the form $x \xrightarrow{a} y$ that originate at x. To simplify the presentation, we assume that there are no deadlocked states—that is, for each $x \in X$, *enabled*$(x) \neq \emptyset$.

A path in TS is a sequence $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots x_i \xrightarrow{a_{i+1}} x_{i+1} \cdots$. Let *paths*$(TS)$ and *finite_paths*(TS) denote the set of infinite and finite paths in TS, respectively. For a finite path p, let *last*(p) denote the final state of p.

Games and strategies Let $S \subseteq \Sigma$. An *S-strategy* is a function f from *finite_paths*(TS) to the set of S-transitions $\longrightarrow_S$ such that for each $p \in$ *finite_paths*(TS), $f(p) \subseteq$ *enabled*(*last*(p)). If $\longrightarrow_S \cap$ *enabled*(*last*(p)) $\neq \emptyset$, we require that $f(p) \neq \emptyset$.

An infinite path $\rho = x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots x_n \xrightarrow{a_n} \cdots$ in *paths*(TS) is said to be *consistent with the S-strategy f* if for each $n \geq 1$ the transition $x_{n-1} \xrightarrow{a_n} x_n$ belongs to the set $f(x_0 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} x_{n-1})$ whenever $a_n \in S$.

A *game* over TS is a pair (Π, x) where Π is a subset of *paths*(TS) and x is a state in X. We say that S has a *winning strategy* for the game (Π, x) if there is an S-strategy f such that every infinite path that begins at x and is consistent with f belongs to Π.

Often, we are interested in restricting our attention to fair computations. An infinite path $\rho \in$ *paths*(TS) is said to be *weakly S-fair* if there are infinitely many states along ρ where no S-transition is enabled or if there are infinitely many S-transitions in ρ. The path is said to be *strongly S-fair* if there are only finitely many positions where an S-transition is enabled or if there are infinitely many S-transitions in ρ.

We can relativize our notion of winning strategies to fair paths in the obvious way—S has a weakly (strongly) fair winning strategy for (Π, x) if there is an S-strategy f such that every weakly (strongly) S-fair path that begins at x and is consistent with f belongs to Π.

3 The Logic

3.1 Syntax

Fix an alphabet Σ and a set of propositions *Prop*. We simultaneously define the sets *FPE* of *finite path expressions*, *PE* of *(infinite) path expressions*, and Φ of *state formulas* as follows. We use e to denote a typical element of *FPE*, α to denote a typical element of *PE*, and φ to denote a typical element of Φ.

$$
\begin{aligned}
e &::= \varphi \in \Phi \mid a \in \Sigma \mid e \cdot e \mid e + e \mid e^* \\
\alpha &::= e \cdot e^\omega \mid \alpha + \alpha \\
\varphi &::= \mathtt{tt} \mid P \in Prop \mid \neg\varphi \mid \varphi \vee \varphi \mid E_S\alpha,\ (S \subseteq \Sigma) \mid A_S\alpha,\ (S \subseteq \Sigma)
\end{aligned}
$$

Observe that finite path expressions are just regular expressions whose alphabet may include state formulas. Similarly, path expressions are traditional ω-regular expressions [Tho90] built from finite path expressions. For state formulas, we can use $\neg$ and $\vee$ to derive the usual Boolean connectives such as $\wedge$ and $\Rightarrow$. Finally, note that the wffs of OR$_\text{E}$X are just the state formulas.

3.2 Semantics

OR$_\text{E}$X formulas are interpreted over Σ-labelled transitions systems equipped with valuations. Let $TS = \langle X, \Sigma, \longrightarrow \rangle$ be a Σ-labelled transition system. A *valuation* $v : X \to 2^{Prop}$ specifies which propositions are true in each state.

We interpret finite path expressions over finite paths in TS, path expressions over infinite paths in TS, and state formulas at states of TS. If $p \in$ *finite_paths*(TS) and $e \in FPE$, we write $TS, p \models e$ to denote that e is satisfied along p. Similarly, for $\rho \in$ *paths*(TS) and $\alpha \in PE$, $TS, \rho \models \alpha$ denotes α is satisfied along ρ. Finally, for $x \in X$ and $\varphi \in \Phi$, $TS, x \models \varphi$ denotes that φ is satisfied at the state x.

We define concatenation of paths in the usual way. Let $p = x_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} x_n$ and $p' = x'_0 \xrightarrow{a'_1} \cdots \xrightarrow{a'_m} x'_m$ be paths such that $x_n = x'_0$. The path $p \cdot p'$ is the path $x_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} x_n \xrightarrow{a'_1} \cdots \xrightarrow{a'_m} x'_m$. The concatenation of a finite path p with an infinite path ρ is defined similarly.

The three forms of the satisfaction relation $\models$ are defined through simultaneous induction as follows.

Finite path expressions

$TS, p \models \varphi$	iff $p = x_0$ and $TS, x_0 \models \varphi$.
$TS, p \models a$	iff $p = x_0 \xrightarrow{a_1} x_1$ and $a_1 = a$.
$TS, p \models e_1 \cdot e_2$	iff $p = p_1 \cdot p_2$, $TS, p_1 \models e_1$ and $TS, p_2 \models e_2$
$TS, p \models e_1 + e_2$	iff $TS, p \models e_1$ or $TS, p \models e_2$
$TS, p \models e^*$	iff $p = x_0$ or $p = p_1 \cdot p_2 \cdots p_m$ and for $i \in \{1, 2, \ldots, m\}$, $TS, p_i \models e$

Path expressions

$TS, \rho \models e_1 \cdot e_2^\omega$	iff there is a prefix $p_0 \cdot p_1 \cdot p_2 \cdots$ of ρ such that $TS, p_0 \models e_1$ and for $i \in \{1, 2, \ldots\}$, $TS, p_i \models e_2$
$TS, \rho \models \alpha_1 + \alpha_2$	iff $TS, \rho \models \alpha_1$ or $TS, \rho \models \alpha_2$

We can associate with each path expression α a set Π_α of infinite paths in TS in the obvious way—$\Pi_\alpha = \{\rho \in \textit{paths}(TS) \mid TS, \rho \models \alpha\}$. We let $\Pi_{\neg\alpha}$ denote the set *paths*$(TS) \setminus \Pi_\alpha$.

State formulas

$TS, x \models \texttt{tt}$	always.
$TS, x \models P \in Prop$	iff $P \in v(x)$.
$TS, x \models \neg\varphi$	iff $TS, x \not\models \varphi$.
$TS, x \models \varphi_1 \vee \varphi_2$	iff $TS, x \models \varphi_1$ or $TS, x \models \varphi_2$.
$TS, x \models E_S\alpha$	iff S has a winning strategy for the game (Π_α, x).
$TS, x \models A_S\alpha$	iff S does not have a winning strategy for the game $(\Pi_{\neg\alpha}, x)$.

It is useful to think of $E_S\alpha$ as asserting that S has a strategy to *enforce* α along every path. Conversely, $A_S\alpha$ asserts that S does *not* have a strategy to *avoid* α along every path—in other words, no matter what strategy S chooses, there is at least one path consistent with the strategy along which α holds.

In the absence of fairness constraints, the games we have described are *determined* and $E_S\varphi$ is logically equivalent to $A_{\Sigma\setminus S}\varphi$. In other words, we can derive one quantifier from the other. However, once we introduce fairness constraints, the games are no longer determined, in general, and the two quantifiers need to be introduced independently, as we have done here. However, even with fairness constraints, $E_S\varphi$ implies $A_{\Sigma\setminus S}\varphi$.

3.3 Examples

Let e be a finite path expression. We shall use the following abbreviations for convenience.

- The formula $E_{\mathcal{S}}e$ (respectively, $A_{\mathcal{S}}e$) abbreviates the formula $E_{\mathcal{S}}e\mathtt{tt}^{\omega}$ (respectively, $A_{\mathcal{S}}e\mathtt{tt}^{\omega}$). Every path in $\Pi_{e\mathtt{tt}^{\omega}}$ has a finite prefix that satisfies the property e.
- The formula $E_{\mathcal{S}}e^{\omega}$ (respectively, $A_{\mathcal{S}}e^{\omega}$) abbreviates the formula $E_{\mathcal{S}}\mathtt{tt}e^{\omega}$ (respectively, $A_{\mathcal{S}}\mathtt{tt}e^{\omega}$).

The operators E and A of logics like CTL* are just abbreviations for E_{Σ} and $E_{\emptyset}$, respectively. Using these branching-time operators, we can specify some traditional temporal properties as follows. Let the set of actions be $\{a_1, a_2, \ldots, a_m\}$. In the examples, Σ abbreviates the finite path expression $(a_1 + a_2 + \cdots + a_m)$.

- The LTL formula $\varphi\mathcal{U}\psi$ (φ holds *until* ψ holds) is captured by the path expression $(\varphi\Sigma)^* \cdot \psi$. Thus, the state formula $A(\varphi\Sigma)^* \cdot \psi$ asserts that $\varphi\mathcal{U}\psi$ is true of every computation path.
- In LTL, the formula $\Diamond\varphi$ (φ holds eventually) can be written as $\mathtt{tt}\mathcal{U}\varphi$. The corresponding path expression is $(\mathtt{tt}\Sigma)^* \cdot \varphi$, which we shall denote *Eventually*(φ).
- Let *Invariant*(φ) denote the path expression $(\varphi\Sigma)^{\omega}$. The expression *Invariant*(φ) asserts that the state formula φ is *invariant* along the path, corresponding to the LTL formula $\Box\varphi$. Thus, the state formula A *Invariant*(φ) says that along all paths φ always holds.
- The formula $A((E(\Sigma^* \cdot \varphi) \cdot \Sigma)^{\omega})$ asserts that along every path φ is always attainable (*Branching Liveness*). This property has been used in the verification of the Futurebus protocol [CGH+95].

We can also assert properties that are *not* expressible in CTL* (or ATL*). For instance, $A(\Sigma a)^{\omega}$ asserts that along all paths, every even position is an a.

Now, let us see how to use OrEx to describe properties of a typical open system. Figure 1 shows a variant of the train gate controller described in [AHK97]. In this system, the atomic propositions are {in_gate, out_of_gate, waiting, admitted}. In the figure, the labels on the states indicate the valuation. The actions can be partitioned into two sets, *train* and *ctr*, corresponding to two components of the system, the train and the controller. The set *train* is given by {*idle*, *request*, *relinquish*, *enter*, *leave*} while *ctr* is given by {*grant*, *reject*, *eject*}.

Here are some properties that one might like to assert about this system.

- If the train is outside the gate and has not yet been granted permission to enter the gate, the controller can prevent it from entering the gate:

$$A\ \mathit{Invariant}((\mathsf{out\_of\_gate} \wedge \neg\mathsf{admitted}) \Rightarrow E_{ctr}\mathit{Invariant}(\mathsf{out\_of\_gate}))$$

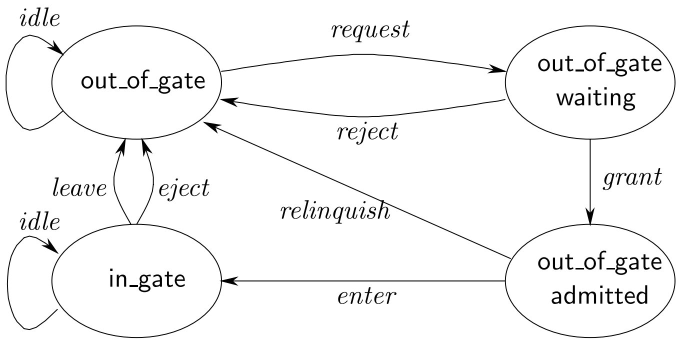


Fig. 1. A train gate controller

- If the train is outside the gate, the controller cannot force it to enter the gate:

  $$A\ \mathit{Invariant}(\mathsf{out\_of\_gate} \Rightarrow A_{ctr}\mathit{Invariant}(\neg \mathsf{in\_gate}))$$

  This property can also expressed by the dual assertion that the train can choose to remain out of the gate.

  $$A\ \mathit{Invariant}(\mathsf{out\_of\_gate} \Rightarrow E_{train}\mathit{Invariant}(\neg \mathsf{in\_gate}))$$

- If the train is outside the gate, the train and the controller can cooperate so that the train enters the gate:

  $$A\ \mathit{Invariant}(\mathsf{out\_of\_gate} \Rightarrow E\ \mathit{Eventually}(\mathsf{in\_gate}))$$

- If the train is in the gate, the controller can ensure that it eventually leaves the gate.

  $$A\ \mathit{Invariant}(\mathsf{in\_gate} \Rightarrow E_{ctr}\mathit{Eventually}(\mathsf{out\_of\_gate}))$$

  Notice that this property is not satisfied by the system unless fairness is introduced—after entering the gate, the train may execute the action *idle* forever. However, since *eject* is continuously enabled at this state, even weak fairness suffices to ensure that the controller can eventually force the train to leave the gate.
- Actually, it is unrealistic to assume that the controller can eject the train once it is in the gate. If we eliminate the transition labelled *eject*, we can make the following assertion which guarantees that as long as the train does not idle continuously, it eventually leaves the gate.

  $$A\ \mathit{Invariant}(\mathsf{in\_gate} \Rightarrow (\neg(\mathit{idle})^{\omega} \Rightarrow \mathit{Eventually}(\mathsf{out\_of\_gate}))$$

  To state this property, we need to integrate assertions about actions and states in the formula. This formula cannot be conveniently expressed in ATL*, where formulas can only refer to properties of states.

4 The Model-Checking Algorithm

We now describe an automata-theoretic model checking algorithm for $\text{Or}_{\text{E}}\text{x}$ formulas. As we remarked earlier, path expressions can be regarded as ω-regular expressions over the infinite alphabet consisting of the finite set of actions Σ together with the set of all state formulas. However, if we fix a finite set of state formulas Ψ, we can regard each path expression which does not refer to state formulas outside Ψ as an ω-regular expression over the finite alphabet $\Sigma \cup \Psi$. We can associate with each such path expression α a language $L(\alpha)$ of infinite words over $\Sigma \cup \Psi$ using the standard interpretation of ω-regular expressions [Tho90].

Unfortunately, this naïve translation does not accurately reflect the meaning of path expressions: $\overline{L(\alpha)}$, the complement of $L(\alpha)$, may contain sequences that are models of α. For instance, if $L(\alpha)$ contains $a\varphi_1\varphi_2 b^\omega$ but does not contain $a\varphi_2\varphi_1 b^\omega$, the second path will be present in $\overline{L(\alpha)}$ though it models α. We require more structure in the infinite sequences we associate with path expressions to faithfully translate logical connectives into automata-theoretic operations.

We begin by defining an alternative model for finite path expressions and path expressions in terms of sequences over an extended alphabet. Let Ψ be a finite set of state formulas. A *Ψ-decorated sequence* over Σ is a sequence in which subsets of Ψ alternate with elements of Σ. More precisely, a Ψ-decorated sequence over Σ is an element of $(2^\Psi \cdot \Sigma)^*(2^\Psi) \cup (2^\Psi \cdot \Sigma)^\omega$. We use s to denote a finite Ψ-decorated sequence and σ to denote an arbitrary (finite or infinite) Ψ-decorated sequence. A finite Ψ-decorated sequence s can be concatenated with a Ψ-decorated sequence σ if the last element of s is identical to the first element of σ. The concatenation is obtained by deleting the last element of s.

We denote the set of finite path expressions and path expressions that do not refer to any state formulas outside Ψ as FPE_Ψ and PE_Ψ, respectively. We can interpret expressions from FPE_Ψ and PE_Ψ over Ψ-decorated sequences. The satisfaction relation is defined as follows:

$$
\begin{array}{ll}
s \models \varphi & \text{iff } s = X_0 \text{ and } \varphi \in X_0 \\
s \models a & \text{iff } s = X_0 a X_1 \\
s \models e_1 \cdot e_2 & \text{iff } s = s_1 \cdot s_2,\ s_1 \models e_1 \text{ and } s_2 \models e_2 \\
s \models e_1 + e_2 & \text{iff } s \models e_1 \text{ or } s \models e_2 \\
s \models e^* & \text{iff } s = X_0 \text{ or } s = s_1 \cdot s_2 \cdots s_n \\
 & \qquad\qquad \text{and for } i \in \{1,2,\ldots,n\},\ s_i \models e \\
\sigma \models e_1 \cdot e_2^\omega & \text{iff there is a prefix } s_0 \cdot s_1 \cdot s_2 \cdots \text{ of } \sigma \text{ such that} \\
 & \quad s_0 \models e_1 \text{ and for } i \in \{1,2,\ldots\},\ s_i \models e_2 \\
\sigma \models \alpha_1 + \alpha_2 & \text{iff } \sigma \models \alpha_1 \text{ or } \sigma \models \alpha_2
\end{array}
$$

Let $TS = \langle X, \Sigma, \longrightarrow \rangle$ be a Σ-labelled transition system and $v : X \to Prop$ be a valuation over TS. For each path $\rho = x_0 \xrightarrow{a_1} x_1 \ldots$ in TS, the corresponding Ψ-decorated sequence σ_ρ is $X_0 a_1 X_1 \ldots$ where $X_i = \{\varphi \mid \varphi \in \Psi \text{ and } TS, x_i \models \varphi\}$. Then, for any path expression $\alpha \in PE_\Psi$, $TS, \rho \models \alpha$ if and only if $\sigma_\rho \models \alpha$.

There is a natural connection between the language $L(\alpha)$ defined by $\alpha \in PE_\Psi$ and the semantics of α in terms of Ψ-decorated sequences. To formalize this, we need to define when a Ψ-decorated sequence embeds a sequence over $\Sigma \cup \Psi$.

Embedding Let $w = w_0 w_1 \ldots$ be an infinite word over $\Sigma \cup \Psi$ and $\sigma = \sigma_0 \sigma_1 \ldots$ be a Ψ-decorated sequence. Observe that $\sigma_i \subseteq \Psi$ if i is even and $\sigma_i \in \Sigma$ if i is odd. We say that σ *embeds* w if there is a monotone function $f : \mathbb{N}_0 \to \mathbb{N}_0$ mapping positions of w into positions of σ such that:

(i) If i is in the range of f, $j < i$ and j is odd, then j is in the range of f.
(ii) If $w_i \in \Sigma$ then $w_i = \sigma_{f(i)}$.
(iii) If $w_i \in \Psi$ then $w_i \in \sigma_{f(i)}$.

We can then establish the following connection between $L(\alpha)$ and the set of Ψ-decorated sequences that model α.

Lemma 4.1. *A Ψ-decorated sequence σ models a path expression $\alpha \in PE_\Psi$ if and only if σ embeds at least one of the sequences in the language $L(\alpha)$ over the alphabet $\Sigma \cup \Psi$.*

Let $|\alpha|$ denote the number of symbols in a path expression α. From the theory of ω-regular languages [Tho90], we know that for each path expression $\alpha \in PE_\Psi$, we can construct a nondeterministic Büchi automaton $\mathcal{A}_\alpha$ over $\Sigma \cup \Psi$ whose size is polynomial in $|\alpha|$ and whose language is precisely $L(\alpha)$.

Using our notion of embedding, we can associate with each Büchi automaton $\mathcal{A}$ over $\Sigma \cup \Psi$ an extended language $L^+(\mathcal{A})$ of Ψ-decorated sequences as follows:

$$L^+(\mathcal{A}) = \{\sigma \mid \exists w \in L(\mathcal{A}) : \sigma \text{ embeds } w\}$$

Lemma 4.2. *For any Büchi automaton $\mathcal{A}$ over $\Sigma \cup \Psi$, we can construct a Büchi automaton $\mathcal{A}^+$ over $\Sigma \cup 2^\Psi$ such that $\mathcal{A}^+$ reads elements from Σ and 2^Ψ alternately, $L^+(\mathcal{A}) = L(\mathcal{A}^+)$ and the number of states of $\mathcal{A}^+$ is polynomial in the number of states of $\mathcal{A}$.*

Proof Sketch: Let $\mathcal{A} = (Q, \delta, q_0, F)$ be a given nondeterministic Büchi automaton over $\Sigma \cup \Psi$. For states $p, q \in Q$ and $U \subseteq \Psi$, we write $p \overset{U}{\Rightarrow} q$ if there is a sequence of transitions from p ending in q whose labels are drawn from the set U. We write $p \overset{U}{\Rightarrow}_F q$ to indicate that there is a sequence of transitions from p to q that passes through a state from F, all of whose labels are drawn from the set U. We write $p \overset{U}{\Rightarrow}_A$ if there is an accepting run starting at p, all of whose labels are drawn from the set U.

The automaton $\mathcal{A}^+ = (Q^+, \Sigma \cup 2^\Psi, \delta^+, F^+, q_0)$ where:

$$\begin{aligned}
Q^+ &= (Q \times \{0,1,2\}) \cup \{q_f, p_f\} \\
F^+ &= (Q \times \{2\}) \cup \{p_f\} \\
\delta^+ &= \{((p,0), U, (q,1)) \mid p \overset{U}{\Rightarrow} q\} \cup \{((p,0), U, (q,2)) \mid p \overset{U}{\Rightarrow}_F q\} \cup \\
&\quad \{((p,0), U, q_f) \mid p \overset{U}{\Rightarrow}_A\} \cup \\
&\quad \{((p,1), a, (q,0)), ((p,2), a, (q,0)) \mid (p,a,q) \in \delta\} \cup \\
&\quad \{((p,0), U, (p,1)) \mid p \in Q, U \subseteq \Psi\} \cup \{(q_f, a, p_f) \mid a \in \Sigma\} \cup \\
&\quad \{(p_f, U, q_f) \mid U \subseteq \Psi\}
\end{aligned}$$

□

The construction allows us to convert a path expression $\alpha \in PE_\Psi$ into an automaton $\mathcal{A}^+_\alpha$ over $\Sigma \cup 2^\Psi$ whose state space is polynomial in $|\alpha|$. Notice that the alphabet, and hence the number of transitions, of $\mathcal{A}^+_\alpha$ is exponential in $|\alpha|$.

The lemma assures us that if we complement $\mathcal{A}^+_\alpha$, we obtain an automaton that accepts precisely those Ψ-decorated sequences that do *not* model α. As we remarked earlier, this is not true of the original automaton $\mathcal{A}_\alpha$: $\overline{L(\alpha)}$ may contain sequences that can be embedded into Ψ-decorated sequences that model α. In a sense, $L^+(\mathcal{A})$ can be thought of as the semantic closure of $L(\mathcal{A})$.

Lemma 4.3 ([Saf88, EJ89]). *A Büchi automaton $\mathcal{A}$ can be effectively determinized (complemented) to get a deterministic Rabin automaton whose size is exponential in the number of states of $\mathcal{A}$ and linear in the number of transitions of $\mathcal{A}$ and whose set of accepting pairs is linear in the number of states of $\mathcal{A}$. This automaton can be constructed in time exponential in the number of states of $\mathcal{A}$ and polynomial in the number of transitions of $\mathcal{A}$.*

Lemma 4.4 follows from the previous two lemmas.

Lemma 4.4. *From a path expression $\alpha \in PE_\Psi$, we can construct a deterministic Rabin automaton $\mathcal{A}^R_\alpha$ ($\overline{\mathcal{A}}^R_\alpha$) that accepts exactly those Ψ-decorated paths that do (do not) model α. The number of states of $\mathcal{A}^R_\alpha$ ($\overline{\mathcal{A}}^R_\alpha$) is exponential in $|\alpha|$ and the number of accepting pairs of $\mathcal{A}^R_\alpha$ ($\overline{\mathcal{A}}^R_\alpha$) is polynomial in $|\alpha|$.*

Theorem 4.5. *Given a transition system TS, a state x in TS and an $\mathrm{OR_EX}$ formula φ, the model checking problem $TS, x \overset{?}{\models} \varphi$ can be solved in time $O((mk)^k)$, where m is given by $2^{O(|\varphi|)} \cdot |TS|$ and k is a polynomial in $|\varphi|$.*

Proof Sketch: We use the bottom-up labelling algorithm of CTL* and ATL*. The only interesting case is when the formula φ to be checked is of the form $E_S\alpha$ (or $A_S\alpha$). Each S-strategy f at a state x can be represented as a tree by unfolding TS starting at x, retaining all $\Sigma \setminus S$ edges in the unfolding, but discarding all S-transitions that are not chosen by the strategy. Conversely, if we unfold TS at x and prune S-transitions while ensuring that at least one S-transition is retained at each state y where $\mathit{enabled}(y) \cap \longrightarrow_S \neq \emptyset$, we obtain a representation of some S-strategy at x.

Let Ψ be the set of state formulas that appear in α. We may assume that each state of the transition system TS is labelled with the formulas from Ψ that hold at that state. In the trees that we use to represent strategies, each state is labelled by the same subset of formulas as the corresponding state in TS.

The trees that represent strategies carry labels on both the states and the edges. We can convert these to state-labelled trees by moving the label on each edge to the destination state of the edge.

Given a transition system TS, we can construct a Büchi tree automaton $\mathcal{T}_{TS}$ that accepts precisely those trees labeled by $\Sigma \cup 2^\Psi$ that arise from strategies, as described in [KV96]. The size of $\mathcal{T}_{TS}$ is linear in the size of TS.

An S-strategy f is a witness for $TS, x \models E_S\alpha$ if and only if every infinite path in the tree corresponding to f models α. Equivalently, every infinite path in this tree, when treated as a sequence over $\Sigma \cup 2^{\Psi}$, is accepted by the Rabin automaton $\mathcal{A}_\alpha^R$ constructed from $\mathcal{A}_\alpha^+$ as described in Lemma 4.4. Since $\mathcal{A}_\alpha^R$ is deterministic, we can convert it into a Rabin tree automaton $\mathcal{T}_\alpha$ that runs $\mathcal{A}_\alpha^R$ along all paths and accepts exactly those trees in which all infinite paths model α. The automaton $\mathcal{T}_\alpha$ has the same size as $\mathcal{A}_\alpha^R$—that is, the number of states is exponential in $|\alpha|$ and the number of accepting pairs is polynomial in $|\alpha|$.

Finally, we construct the product of $\mathcal{T}_{TS}$ and $\mathcal{T}_\alpha$ to obtain a tree automaton that accepts a tree if and only if it corresponds to a strategy f such that all infinite paths in TS consistent with f satisfy α. We can then use the algorithm of Emerson and Jutla [EJ88] to check whether this product automaton accepts a nonempty set of trees. The Emerson-Jutla algorithm solves the emptiness problem for a tree automaton with i states and j accepting pairs in time $O((ij)^{3j})$. From this, we can derive that the emptiness of the product tree automaton we have constructed can be checked in time $O((mk)^k)$, where m is given by $2^{O(|\alpha|)} \cdot |TS|$ while k is a polynomial in $|\alpha|$.

The case $A_S\alpha$ is similar to $E_S\alpha$ with one modification—use the construction due to Emerson and Jutla [EJ89] instead of Safra's construction to obtain a deterministic Rabin automaton $\overline{\mathcal{A}}_\alpha^R$ that accepts exactly those sequences over $\Sigma \cup 2^{\Psi}$ that do *not* model α.

In the constructions above, we have not taken into account the case when we are interested only in (weakly or strongly) S-fair computations. Fairness constraints can be handled by adding a simple Rabin condition to the automaton $\mathcal{T}_{TS}$ that accepts all trees corresponding to S-strategies. This extra structure does not materially affect the overall complexity of the procedure.

□

5 Related Work

In [AHK97], the logic ATL* is interpreted over two kinds of transition systems: synchronous structures and asynchronous structures. In each case, the notion of *agent* plays a prominent role in the definition. In a *synchronous structure*, each state is associated with a unique agent. An important subclass of synchronous structures is the one with two agents, one representing the system and the other the environment. Such systems have been studied by Ramadge and Wonham [RW89] in the context of designing controllers for discrete-event systems.

In an *asynchronous structure*, each transition is associated with an agent. A synchronous structure can be thought of as a special kind of asynchronous structure where all transitions out of a state are associated with a single agent. Our Σ-labelled transition systems correspond to asynchronous structures, where the set of agents corresponds to the alphabet Σ.

The notion of strategy described in [AHK97] is slightly different from the one we use here. In their framework, each agent has a strategy and an S-strategy at a state allows any move permitted by all the individual strategies of the agents in S. Our definition of global strategy is more generous. We believe that decompositions of strategies where each agent has a view of the global state do not arise naturally and have therefore chosen not to incorporate this into our logic. Nevertheless, we can easily simulate the framework of [AHK97] in our setup without changing in the complexity of the construction.

In the Introduction, we mentioned that $\textsc{Or}_{\textsc{E}}\textsc{x}$ subsumes ATL* in expressive power since ω-regular expressions are more powerful than LTL. We also note that in the context of open systems, it is natural to refer to properties of *both* states and actions in a transition system. If we use LTL to express properties of paths, as is the case with ATL*, we have to encode actions in the states of the system since LTL can only talk about properties of states. On the other hand $\textsc{Or}_{\textsc{E}}\textsc{x}$ provides a seamless way of interleaving assertions about actions and states along a path. This makes for more natural models of open systems—for instance, compare our example of the train gate controller with explicit action labels with the version in [AHK97] where the action labels are encoded as propositions.

There has been earlier work on game logics that mention actions. In [Par83], Parikh defines a propositional logic of games that extends propositional dynamic logics [Har84]. The logic studied in [Par83] is shown to be decidable. A complete axiomatization is provided, but the model-checking problem is not studied.

6 Conclusions

As Theorem 4.5 indicates, the model-checking complexity of $\textsc{Or}_{\textsc{E}}\textsc{x}$ is at least one exponential better than that of ATL*, though $\textsc{Or}_{\textsc{E}}\textsc{x}$ subsumes ATL* in expressive power. If we restrict the quantifiers of $\textsc{Or}_{\textsc{E}}\textsc{x}$ to the normal branching-time interpretation A and E, we obtain a system called $\textsc{Br}_{\textsc{E}}\textsc{x}$ that corresponds to the branching-time logic ECTL*. The model-checking algorithm for $\textsc{Br}_{\textsc{E}}\textsc{x}$ is exponential, which is comparable to the complexity of model-checking CTL*.

Why does going from CTL* to ATL* add an exponential to the complexity of model-checking, unlike the transition from $\textsc{Br}_{\textsc{E}}\textsc{x}$ to $\textsc{Or}_{\textsc{E}}\textsc{x}$? The construction in [VW86] can be used to build nondeterministic Büchi automata for LTL formulas α and $\neg\alpha$ with exponential blow-up. Thus the complexity of model-checking CTL* stays exponential. When we translate $\textsc{Br}_{\textsc{E}}\textsc{x}$ path formulas into automata we only incur a polynomial blowup in size, but we may have to complement the automaton for α to get an automaton for $\neg\alpha$. Complementation is, in general, exponential and, consequently, model-checking for $\textsc{Br}_{\textsc{E}}\textsc{x}$ is also exponential.

When going from CTL* to ATL*, we need to determinize the Büchi automaton constructed from the LTL formula α so that we can interpret the automaton over trees. This blows up the automaton by another exponential, resulting in a model-checking algorithm that is doubly exponential in the size of the input formula. In $\textsc{Or}_{\textsc{E}}\textsc{x}$ the automaton we construct for path formulas is already determinized, so we avoid the second exponential blow-up.

Finally, we note that if we restrict our syntax to only permit the existential path quantifier $E\alpha$, we can model-check $\textsc{Br}_{\textsc{E}}\textsc{x}$ in polynomial time — essentially, it suffices to construct a nondeterministic automaton for α. On the other hand, the model-checking problem for $\textsc{Or}_{\textsc{E}}\textsc{x}$ remains exponential even with this restriction because we have to determinize the automaton for α. Since ω-regular languages are closed under complementation, we can reduce any formula in $\textsc{Br}_{\textsc{E}}\textsc{x}$

or $\mathrm{OR_EX}$ to one with just existential path quantifiers. However, in the process, we have to complement ω-regular expressions. Since this is, in general, an exponential operation, the cost of this translation is exponential (perhaps nonelementary, since nested negations introduce nested exponentials). Thus, the full language that we have presented here permits more succinct specifications *and* more efficient verification than the reduced language with just existential quantifiers.

References

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th IEEE Symposium on the Foundations of Computer Science*. IEEE Press, 1997.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CGH$^+$95] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Future+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[EH86] E. A. Emerson and J. Y. Halpern. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A.Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of the IEEE FOCS*, 1988.

[EJ89] E.A.Emerson and C. Jutla. On simultaneously determinising and complementing ω-automata. In *Proceedings of the IEEE FOCS*, 1989.

[Har84] David Harel. Dynamic logic. In *Handbook of Philosophical Logic, Volume II.* Reidel, 1984.

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

[KV96] Orna Kupferman and Moshe Vardi. Module checking. In *Proceedings of CAV'96, LNCS 1102.* Springer-Verlag, 1996.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[Par83] R. Parikh. Propositional game logic. In *Proceedings of 24th IEEE FOCS*, pages 195–200, 1983.

[Pnu76] A. Pnueli. The temporal logic of programs. In *Proceedings of the IEEE Symposium on the Foundations of Computing Science.* IEEE Press, 1976.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of discrete-event systems. *IEEE Trans. on Control Theory*, 77:81–98, 1989.

[Saf88] S. Safra. On complexity of ω-automata. In *In the Proceedings of the 29th FOCS*, 1988.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B.* Elsevier Science Publishers, 1990.

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–344, Cambridge, Massachusetts, June 1986. Computer Society Press.

An Argumentation Approach to Semantics of Declarative Programs with Defeasible Inheritance

Ekawit Nantajeewarawat[1] and Vilas Wuwongse[2]

[1] Information Technology Program,
Sirindhorn International Institute of Technology, Thammasat University,
P.O. Box 22, Thammasat-Rangsit Post Office, Pathumthani 12121, Thailand
ekawit@siit.tu.ac.th

[2] Computer Science and Information Management Program,
School of Advanced Technologies, Asian Institute of Technology,
P.O. Box 4, Klongluang, Pathumthani 12120, Thailand
vw@cs.ait.ac.th

Abstract. Inheritance is a characteristic reasoning mechanism in systems with taxonomic information. In rule-based deductive systems with inclusion polymorphism, inheritance can be captured in a natural way by means of typed substitution. However, with method overriding and multiple inheritance, it is well-known that inheritance is nonmonotonic and the semantics of inheritance becomes problematical. We present a general framework, based on Dung's abstract theory of argumentation, for developing a natural semantics for declarative programs with dynamic defeasible inheritance. We investigate the relationship between the presented semantics and Dobbie and Topor's perfect model (with overriding) semantics, and show that for inheritance-stratified programs, the two semantics coincide. The proposed semantics, nevertheless, still provides the correct skeptical meanings for non-inheritance-stratified programs, while the perfect model semantics fails to yield sensible meanings for them.

1 Introduction

One of the most salient features associated with generalization taxonomy is inheritance. In logic-based deduction systems which support inclusion polymorphism (or subtyping), inheritance can be captured in an intuitive way by means of typed substitutions. To illustrate this, suppose that tom is an individual of type $\mathsf{student}$. Then, given a program clause:

$$C1: \quad \mathsf{X}\colon\mathsf{student}[\mathsf{residence} \rightarrow \mathsf{east\text{-}dorm}] \quad \leftarrow \quad \mathsf{X}[\mathsf{lives\text{-}in} \rightarrow \mathsf{rangsit\text{-}campus}], \; \mathsf{X}[\mathsf{sex} \rightarrow \mathsf{male}],$$

which is intended to state that for any student X, if X lives in $\mathsf{rangsit\text{-}campus}$ and X is male, then X's residence place is $\mathsf{east\text{-}dorm}$; one can obtain by the application of the typed substitution $\{\mathsf{X}\colon\mathsf{student}/\mathsf{tom}\}$ to $C1$ the ground clause:

$$G1: \ \mathsf{tom[residence \rightarrow east\text{-}dorm]} \ \leftarrow \ \mathsf{tom[lives\text{-}in \rightarrow rangsit\text{-}campus]}, \ \mathsf{tom[sex \rightarrow male]}.$$

The clause $C1$ can naturally be considered as a conditional definition of the method residence associated with the type (class[1]) student and the clause $G1$ as a definition of the same method for tom inherited from the type student.

However, when a method is supposed to return a unique value for an object, definitions of a method inherited from different types, tend to conflict. For example, suppose that tom is also an individual of type employee and a clause:

$$C2: \ \mathsf{X\!:employee[residence \rightarrow west\text{-}flats]} \ \leftarrow \ \mathsf{X[lives\text{-}in \rightarrow rangsit\text{-}campus]}, \ \mathsf{X[marital\text{-}status \rightarrow married]},$$

defining the method residence for an employee is also given. Then, the definition of residence for tom obtained from $C2$, *i.e.*,

$$G2: \ \mathsf{tom[residence \rightarrow west\text{-}flats]} \ \leftarrow \ \mathsf{tom[lives\text{-}in \rightarrow rangsit\text{-}campus]}, \ \mathsf{tom[marital\text{-}status \rightarrow married]},$$

conflicts with the previously inherited definition $G1$ when they are both applicable. In the presence of such conflicting definitions, the usual semantics of definite programs, *e.g.*, the minimal model semantics, does not provide satisfactory meanings for programs; for example, if a program has both $G1$ and $G2$ above as its ground instances, then, whenever its minimal model entails each atom in the antecedents of $G1$ and $G2$, it will entail the conflicting information that tom's residence place is east-dorm and is west-flats.

In order to provide appropriate meanings for programs with such conflicting inherited definitions, a different semantics that allows some ground clauses whose antecedents are satisfied to be inactive is needed. This paper applies Dung's theory of argumentation [6] to the development of such a semantics. To resolve inheritance conflicts, the proposed approach requires a binary relation on program ground clauses, called the *domination relation*, which determines among possibly conflicting definitions whether one is intended to defeat another. For example, with additional information that students who are also employees usually prefer the accommodation provided for employees, $G2$ is supposed to defeat $G1$. With such a domination relation, a program will be transformed into an argumentation framework, which captures the logical interaction between the intended deduction and domination; and, then, the meaning of the program will be defined based on the grounded extension of this argumentation framework.

Using this approach, conflict resolution is performed *dynamically* with respect to the applicability of method definitions. That is, the domination of one method definition over another is effective only if the antecedent of the dominating definition succeeds. The appropriateness of dynamic method resolution in the context of deductive rule-based systems, where method definitions are often conditional and may be inapplicable to certain objects, is advocated by [1]. In particular, with the possibility of overriding, when the definitions in the most

[1] In this paper, the terms "type" and "class" are used interchangeably.

specific type are inapplicable, it is reasonable to try to apply those in a more general type.

In order to argue for the correctness and the generality of the proposed semantics in the presence of method overriding, its relationship to the perfect model (with overriding) semantics proposed by Dobbie and Topor [5] is investigated. The investigation reveals that these two semantics coincide for inheritance-stratified programs. Moreover, while the perfect model semantics fails to provide sensible meanings for programs which are not inheritance-stratified, the presented semantics still yields their correct skeptical meanings.

For the sake of simplicity and generality, this paper uses Akama's axiomatic theory of logic programs [4], called DP theory (the theory of declarative programs), as its primary logical basis. The rest of this paper is organized as follows. Section 2 recalls some basic definitions and results from Dung's argumentation-theoretic foundation and DP theory. Section 3 describes the proposed semantics. Section 4 establishes the relationship between the proposed semantics and the perfect model (with overriding) semantics. Section 5 discusses other related works and summarizes the paper.

2 Preliminaries

2.1 Argumentation Framework

Based on the basic idea that a statement is believable if some argument supporting it can be defended successfully against attacking arguments, Dung has developed an abstract theory of argumentation [6] and demonstrated that many approaches to nonmonotonic reasoning in AI are special forms of argumentation. In this subsection, the basic concepts and results from this theory are recalled.

Definition 1. An *argumentation framework* is a pair $(AR, attacks)$, where AR is a set and *attacks* is a binary relation on AR. □

In the sequel, let $AF = (AR, attacks)$ be an argumentation framework. The elements of AR are called *arguments*. An argument $a \in AR$ is said to *attack* an argument $b \in AR$, iff $(a, b) \in attacks$. Let $B \subseteq AR$. B is said to *attack* an argument $b \in AR$, iff some argument in B attacks b. An argument $a \in AR$ is said to be *acceptable* with respect to B, iff, for each $b \in AR$, if b attacks a, then B attacks b. B is said to be *conflict-free*, iff there do not exist arguments $a, b \in B$ such that a attacks b. B is said to be *admissible*, iff B is conflict-free and every argument in B is acceptable with respect to B.

The credulous semantics and the stable semantics of AF are defined by the notions of preferred extension and stable extension, respectively:

Definition 2. A *preferred extension* of AF is a maximal (with respect to set inclusion) admissible subset of AR. A set $A \subseteq AR$ is called a *stable extension* of AF, iff A is conflict-free and A attacks every argument in $AR - A$. □

To define the grounded (skeptical) semantics of AF (Definition 3), the function F_{AF} on 2^{AR}, called the *characteristic function* of AF, is defined by:

$$F_{AF}(X) = \{a \mid a \text{ is acceptable with respect to } X\}.$$

Clearly, F_{AF} is monotonic (with respect to $\subseteq$), and, thus, has the least fixpoint.

Definition 3. The *grounded extension* of AF is the least fixpoint of F_{AF}. □

The next example illustrates the three kinds of extensions.

Example 1. Let $AF = (AR, attacks)$, where $AR = \{a, b, c, d, e\}$ and $attacks = \{(a, b), (b, c), (d, e), (e, d)\}$. Then, AF has two preferred extensions, *i.e.*, $\{a, c, d\}$ and $\{a, c, e\}$, which are also stable extensions. As $F_{AF}(\emptyset) = \{a\}$ and $F^2_{AF}(\emptyset) = \{a, c\} = F^3_{AF}(\emptyset)$, the grounded extension of AF is $\{a, c\}$. □

Well-foundedness of an argumentation framework, recalled next, is a sufficient condition for the coincidence between the three kinds of extensions.

Definition 4. AF is *well-founded*, iff there exists no infinite sequence of arguments $a_0, a_1, \ldots, a_n, \ldots$ such that for each $i \geq 0$, a_{i+1} attacks a_i. □

Theorem 1. *If AF is well-founded, then it has exactly one preferred extension and one stable extension, each of which is equal to its grounded extension.* □

2.2 DP Theory

DP theory [4] is an axiomatic theory which purports to generalize the concept of conventional logic programs to cover a wider variety of data domains. As an introduction to DP theory, the notion of a specialization system is reviewed first. It is followed by the concepts of declarative programs and their minimal model semantics on a specialization system.

Definition 5. A *specialization system* is a 4-tuple $(\mathcal{A}, \mathcal{G}, \mathcal{S}, \mu)$ of three sets $\mathcal{A}, \mathcal{G}$ and $\mathcal{S}$, and a mapping μ from $\mathcal{S}$ to *partial_map*$(\mathcal{A})$ (*i.e.*, the set of all partial mappings on $\mathcal{A}$), that satisfies the conditions:

1. $(\forall s, s' \in \mathcal{S})(\exists s'' \in \mathcal{S}) : \mu s'' = (\mu s') \circ (\mu s)$,
2. $(\exists s \in \mathcal{S})(\forall a \in \mathcal{A}) : (\mu s)a = a$,
3. $\mathcal{G} \subseteq \mathcal{A}$. □

In the rest of this subsection, let $\Gamma = (\mathcal{A}, \mathcal{G}, \mathcal{S}, \mu)$ be a specialization system. The elements of $\mathcal{A}$ are called *atoms*; the set $\mathcal{G}$ is called the *interpretation domain*; the elements of $\mathcal{S}$ are called *specialization parameters* or simply *specializations*; and the mapping μ is called the *specialization operator*. A specialization $s \in \mathcal{S}$ is said to be *applicable* to $a \in \mathcal{A}$, iff $a \in dom(\mu s)$. By formulating a suitable specialization operator together with a suitable set of specialization parameters, the typed-substitution operation can be regarded as a special form of specialization operation.

Let X be a subset of $\mathcal{A}$. A *definite clause* C *on* X is a formula of the form $(a \leftarrow b_1, \ldots, b_n)$, where $n \geq 0$ and $a, b_1, \ldots, b_n$ are atoms in X. The atom a is denoted by $head(C)$ and the set $\{b_1, \ldots, b_n\}$ by $Body(C)$. When $n = 0$, C is called a *unit clause.* A definite clause C' is an *instance* of C, iff there exists $s \in \mathcal{S}$ such that s is applicable to $a, b_1, \ldots, b_n$ and $C' = ((\mu s)a \leftarrow (\mu s)b_1, \ldots, (\mu s)b_n)$. A definite clause on $\mathcal{G}$ is called a *ground clause.* A *declarative program* on Γ is a set of definite clauses on $\mathcal{A}$. Given a declarative program P on Γ, let $Gclause(P)$ denote the set of all ground instances of clauses in P. Conventional (definite) logic programs as well as typed logic programs can be viewed as declarative programs on some specialization systems.

An *interpretation* is defined as a subset of $\mathcal{G}$. Let I be an interpretation. If C is a definite clause on $\mathcal{G}$, then I is said to *satisfy* C iff $(head(C) \in I)$ or $(Body(C) \nsubseteq I)$. If C is a definite clause on $\mathcal{A}$, then I is said to *satisfy* C iff for every ground instance C' of C, I satisfies C'. I is a *model* of a declarative program P on Γ, iff I satisfies every definite clause in P. The meaning of P is defined as the *minimum model* of P, which is the intersection of all models of P.

3 The Proposed Semantics

In the sequel, let $\Gamma = (\mathcal{A}, \mathcal{G}, \mathcal{S}, \mu)$ be a specialization system and P a declarative program on Γ. Let *dominates* be a binary relation on $Gclause(P)$. A ground clause C of P is said to *dominate* another ground clause C' of P, iff $(C, C') \in$ *dominates*. It will be assumed henceforth that the relation *dominates* prioritizes the ground clauses of P; more precisely, for any ground clauses C, C' of P, C dominates C', iff C is preferable to C' and whenever $Body(C)$ is satisfied, C' will be inactive. It should be emphasized that the domination of a ground clause C over another ground clause C' is intended to be *dynamically* operative with respect to the applicability of C, *i.e.*, the domination is effective only if the condition part of C is satisfied. The relation *dominates* will also be referred to as the *domination relation* of P.

3.1 Derivation Trees

The notion of a derivation tree of a program will be introduced first. A derivation tree of P represents a derivation of one conclusion from P. It will be considered as an argument that supports its derived conclusion. Every conclusion in the minimum model of P is supported by at least one derivation tree of P.

Definition 6. A *derivation tree* of P is defined inductively as follows:

1. If C is a unit clause in $Gclause(P)$, then the tree of which the root is C and the height is 0 is a *derivation tree* of P.

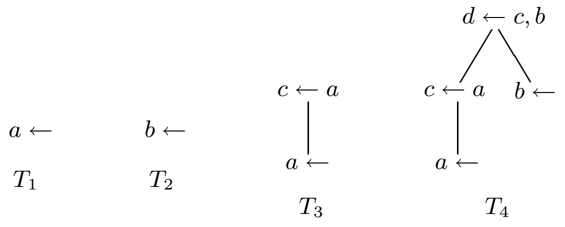


Fig. 1. The derivation trees of the program P_1.

2. If $C = (a \leftarrow b_1, \ldots, b_n)$ is a clause in $Gclause(P)$ such that $n > 0$ and $T_1, \ldots, T_n$ are derivation trees of P with roots $C_1, \ldots, C_n$, respectively, such that $head(C_i) = b_i$, for each $i \in \{1, \ldots, n\}$, then the tree of which the root is C and the immediate subtrees are exactly $T_1, \ldots, T_n$ is a *derivation tree* of P.
3. Nothing else is a derivation tree of P. □

Example 2. Let P_1 be a declarative program comprising the five ground clauses:

$$a \leftarrow \qquad b \leftarrow \qquad c \leftarrow a \qquad d \leftarrow c, b \qquad f \leftarrow e$$

Then, P_1 has exactly four derivation trees, which are shown by Figure 1. Note that the derivation trees T_1, T_2, T_3 and T_4 in the figure depict the derivation of the conclusions a, b, c and d, respectively. □

In the sequel, the root of a derivation tree T will be denoted by $root(T)$. A derivation tree T will be regarded as an argument that supports the activation of the ground clause $root(T)$ (and, thus, supports the conclusion $head(root(T))$).

3.2 Grounded-Extension-Based Semantics

In order to define the meaning of P with respect to the domination relation, the program P will be transformed into an argumentation framework $AF_\iota(P)$, which provides an appropriate structure for understanding the dynamic interaction of the deduction process of P and the specified domination relation. Intuitively, one argument (derivation tree) attacks another argument (derivation tree), when the ground clause supported by the former dominates some ground clause used in the construction of the latter.

Definition 7. The argumentation framework $AF_\iota(P) = (AR, attacks)$ is defined as follows: AR is the set of all derivation trees of P, and for any $T, T' \in AR$, T attacks T', iff $root(T)$ dominates some node of T'. □

Example 3. Referring to the program P_1 of Example 2, suppose that the ground clause $a \leftarrow$ dominates the ground clause $b \leftarrow$, and for any other two ground clauses in P_1, one does not dominate the other. Then $AF_\iota(P_1) = (AR_{P_1}, attacks)$, where AR_{P_1} consists of the four derivation trees in Figure 1 and $attacks = \{(T_1, T_2), (T_1, T_4)\}$. (Note that T_1 attacks T_4 as the root of T_1 dominates the right leaf of T_4.) □

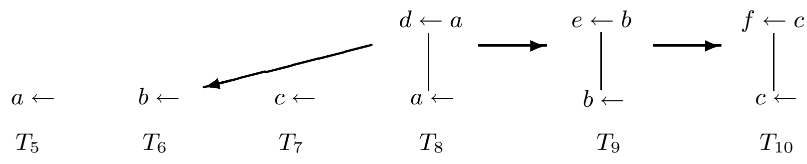


Fig. 2. The argumentation framework for the program P_2.

The meaning of P is now defined as the set of all conclusions which are supported by some arguments in the grounded extension of $AF_\iota(P)$.

Definition 8. The *grounded-extension-based meaning* of P, denoted by $\mathcal{M}_P^{\text{GE}}$, is defined as the set $\{head(root(T)) \mid T \in GE\}$, where GE is the grounded extension of $AF_\iota(P)$. □

Four examples illustrating the proposed semantics are given below.

Example 4. Consider $AF_\iota(P_1)$ of Example 3. Let F be the characteristic function of $AF_\iota(P_1)$. Clearly, $F(\emptyset) = \{T_1, T_3\} = F(F(\emptyset))$. Thus $F(\emptyset)$ is the grounded extension of $AF_\iota(P_1)$, and, then, $\mathcal{M}_{P_1}^{\text{GE}} = \{a, c\}$. □

Example 5. Let a declarative program P_2 comprise the six ground clauses:

$$a \leftarrow \qquad b \leftarrow \qquad c \leftarrow \qquad d \leftarrow a \qquad e \leftarrow b \qquad f \leftarrow c$$

Let $d \leftarrow a$ dominate $b \leftarrow$ and $e \leftarrow b$ dominate $f \leftarrow c$, and assume that for any other two ground clauses in P_2, one does not dominate the other. Then $AF_\iota(P_2) = (AR_{P_2}, attacks)$, where AR_{P_2} consists of the six derivation trees shown in Figure 2 and $attacks = \{(T_8, T_6,), (T_8, T_9), (T_9, T_{10})\}$ as depicted by the darker arrows between the derivation trees in the figure. Let F be the characteristic function of $AF_\iota(P_2)$. Then $F(\emptyset) = \{T_5, T_7, T_8\}$, and $F^2(\emptyset) = \{T_5, T_7, T_8, T_{10}\} = F^3(\emptyset)$. So $\mathcal{M}_{P_2}^{\text{GE}} = \{a, c, d, f\}$. This example also illustrates dynamic conflict resolution, *i.e.*, the domination of the ground clause $e \leftarrow b$ over the ground clause $f \leftarrow c$ does not always prevent the activation of the latter. □

Example 6. Refer to the clauses $C1, C2, G1$ and $G2$ given at the beginning of Section 1. Let tom belong both to type student and to type employee. Consider a program P_3 comprising $C1, C2$ and the following three clauses:

$C3$: tom[lives-in $\rightarrow$ rangsit-campus] $\leftarrow$
$C4$: tom[sex $\rightarrow$ male] $\leftarrow$
$C5$: tom[marital-status $\rightarrow$ married] $\leftarrow$

Assume, for simplicity, that $C1$ and $C2$ have $G1$ and $G2$, respectively, as their only ground instances. Suppose that students who are also employees prefer the accommodation provided for employees, and, then, that $G2$ dominates $G1$. Then, it is simple to see that $\mathcal{M}^{\mathrm{GE}}_{P_3}$ contains $\mathsf{tom}[\mathsf{residence} \rightarrow \mathsf{west\text{-}flats}]$ but does not contain $\mathsf{tom}[\mathsf{residence} \rightarrow \mathsf{east\text{-}dorm}]$, and yields the desired meaning of P_3.

To demonstrate dynamic conflict resolution, suppose next that the clause $C5$ is removed from P_3. Then, instead of containing $\mathsf{tom}[\mathsf{residence} \rightarrow \mathsf{west\text{-}flats}]$, $\mathcal{M}^{\mathrm{GE}}_{P_3}$ contains $\mathsf{tom}[\mathsf{residence} \rightarrow \mathsf{east\text{-}dorm}]$; and, it still provides the correct meaning of P_3 in this case. □

Example 7. This example illustrates method overriding. Let ait be an instance of type int(ernational)-school and int-school be a subtype of school. Let a program P_4 comprise the following three clauses:

$$\begin{array}{lcl}
\mathsf{X}\!:\mathsf{school}[\mathsf{medium\text{-}of\text{-}teaching} \rightarrow \mathsf{thai}] & \leftarrow & \mathsf{X}[\mathsf{located\text{-}in} \rightarrow \mathsf{thailand}] \\
\mathsf{X}\!:\mathsf{int\text{-}school}[\mathsf{medium\text{-}of\text{-}teaching} \rightarrow \mathsf{english}] & \leftarrow & \\
\mathsf{ait}[\mathsf{located\text{-}in} \rightarrow \mathsf{thailand}] \quad \leftarrow & &
\end{array}$$

For the sake of simplicity, assume that P_4 has only three ground clauses:

$$\begin{array}{lllcl}
G3: & \mathsf{ait}[\mathsf{medium\text{-}of\text{-}teaching} \rightarrow \mathsf{thai}] & \leftarrow & \mathsf{ait}[\mathsf{located\text{-}in} \rightarrow \mathsf{thailand}] \\
G4: & \mathsf{ait}[\mathsf{medium\text{-}of\text{-}teaching} \rightarrow \mathsf{english}] & \leftarrow & \\
G5: & \mathsf{ait}[\mathsf{located\text{-}in} \rightarrow \mathsf{thailand}] \quad \leftarrow & &
\end{array}$$

Since int-school is more specific than school, $G4$ is supposed to override $G3$; therefore, let $G4$ dominate $G3$. It is readily seen that $\mathcal{M}^{\mathrm{GE}}_{P_4}$ is the set consisting of the two atoms $\mathsf{ait}[\mathsf{located\text{-}in} \rightarrow \mathsf{thailand}]$ and $\mathsf{ait}[\mathsf{medium\text{-}of\text{-}teaching} \rightarrow \mathsf{english}]$, which is the expected meaning of P_4. □

4 Perfect Model (with Overriding) Semantics

Dobbie and Topor defined a deductive object-oriented language called Gulog [5], in which inheritance is realized through typed substitutions, and studied the interaction of deduction, inheritance and *overriding* in the context of this language. The declarative semantics for Gulog programs is based on Przymusinski's perfect model semantics for logic programs [11], but using the possibility of overriding instead of negation in defining a priority relationship between ground atoms. The perfect model (with overriding) semantics provides the correct meanings for inheritance-stratified programs. In order to investigate the relationship between this semantics and the grounded-extension-based semantics, the notions of inheritance stratification and perfect model will be reformulated in the framework of DP theory in Subsection 4.1. The relationship between the two kinds of semantics will then be discussed in Subsection 4.2.

4.1 Inheritance-Stratified Programs and Perfect Models

According to [5], a program is inheritance-stratified if there is no cycle in any definition of a method, *i.e.*, a definition of a method does not depend on an inherited definition of the same method. More precisely:

Definition 9. A declarative program P on Γ is said to be *inheritance-stratified*, iff it is possible to decompose the interpretation domain $\mathcal{G}$ into disjoint sets, called *strata*, $G_0, G_1, \ldots, G_\gamma, \ldots$, where $\gamma < \delta$ and δ is a countable ordinal, such that the following conditions are all satisfied.

1. For each $C \in \mathit{Gclause}(P)$, if $\mathit{head}(C) \in G_\alpha$, then
   (a) for each $b \in \mathit{Body}(C)$, $b \in \bigcup_{\beta \leq \alpha} G_\beta$,
   (b) for each $C' \in \mathit{Gclause}(P)$ such that C' dominates C,
      i. $\mathit{head}(C') \in \bigcup_{\beta \leq \alpha} G_\beta$,
      ii. for each $b' \in \mathit{Body}(C')$, $b' \in \bigcup_{\beta < \alpha} G_\beta$.
2. There exists no infinite sequence $C_0, C_1, \ldots, C_n, \ldots$ of clauses in $\mathit{Gclause}(P)$ such that for each $i \geq 0$, C_{i+1} dominates C_i.

Any decomposition $\{G_0, G_1, \ldots, G_\gamma, \ldots\}$ of $\mathcal{G}$ satisfying the above conditions is called an *inheritance stratification* of P. □

An example of non-inheritance-stratified programs will be given in Subsection 4.2 (Example 8). The next theorem illuminates the coincidence between the grounded extension, preferred extension and stable extension of the argumentation framework for an inheritance-stratified program (see Theorem 1 in Subsection 2.1). Its proof can be found in the full version of this paper [10].

Theorem 2. *If P is inheritance-stratified, then $AF_\iota(P)$ is well-founded.* □

With overriding, not every ground clause of a program is expected to be satisfied by a *reasonable* model of that program. More precisely, a ground clause need not be satisfied if it is overridden by some ground clause whose premise is satisfied. This leads to the following notion of a model with overriding:

Definition 10. An interpretation I is a *model with overriding* (for short, *o-model*) of P, iff for each $C \in \mathit{Gclause}(P)$, either I satisfies C or there exists $C' \in \mathit{Gclause}(P)$ such that C' dominates C and $\mathit{Body}(C') \subseteq I$. □

A program may have more than one *o*-model. Following [5], priority relations between ground atoms are defined based on the possibility of overriding.

Definition 11. Priority relations $<_p$ and $\leq_p$ on $\mathcal{G}$ are defined as follows:

1. If $C \in \mathit{Gclause}(P)$, then
   (a) for each $b \in \mathit{Body}(C)$, $\mathit{head}(C) \leq_p b$,
   (b) for each $C' \in \mathit{Gclause}(P)$, if C' dominates C, then
      i. $\mathit{head}(C) \leq_p \mathit{head}(C')$,
      ii. for each $b' \in \mathit{Body}(C')$, $\mathit{head}(C) <_p b'$,
2. If $a \leq_p b$ and $b \leq_p c$, then $a \leq_p c$,
3. If $a \leq_p b$ and $b <_p c$ (respectively, $d <_p a$), then $a <_p c$ (respectively, $d <_p b$),
4. If $a <_p b$, then $a \leq_p b$,
5. Nothing else satisfies $<_p$ or $\leq_p$. □

A preference relationship among *o*-models will then be defined based on the priority relation $<_p$.

Definition 12. Let M and N be o-models of P. M is said to be *preferable* to N, in symbols, $M \ll N$, iff $M \neq N$ and for each $a \in M - N$, there exists $b \in N - M$ such that $a <_p b$. M is said to be a *perfect o-model* of P, iff there exists no o-model of P preferable to M. □

Every inheritance-stratified program P has exactly one perfect o-model,[2] denoted by $\mathcal{M}_P^{\text{Perf}}$, which provides the correct meaning of P with respect to method overriding.

4.2 Relationship between the Proposed Semantics and Perfect Model (with Overriding) Semantics

It is shown in the full version of this paper [10] that:

Theorem 3. *If P is inheritance-stratified and the domination relation is transitive, then $\mathcal{M}_P^{GE} = \mathcal{M}_P^{Perf}$.* □

It is important to note that since the domination due to method overriding is typically transitive, the transitivity requirement does not weaken Theorem 3.

For programs that are not inheritance-stratified, the perfect model semantics fails to provide their sensible meanings, while the proposed semantics still yields their correct skeptical meanings. (The skeptical approach to method resolution discards all conflicting definitions.) This is demonstrated by the next example.

Example 8. Let tom be an instance of type gr(aduate)-student and gr-student is a subtype of student. Consider the declarative program P_5 comprising the following five clauses:

$C6$:	X: student[math-ability → good]	←	X[math-grade → b]
$C7$:	X: student[major → math]	←	X[math-ability → good], X[favourite-subject → math]
$C8$:	X: gr-student[math-ability → average]	←	X[major → math], X[math-grade → b]
$C9$:	tom[math-grade → b] ←		
$C10$:	tom[favourite-subject → math] ←		

Without loss of generality, suppose for simplicity that $C6, C7$ and $C8$ have as their ground instances only the clauses $G6, G7$ and $G8$, given below, respectively:

$G6$:	tom[math-ability → good]	←	tom[math-grade → b]
$G7$:	tom[major → math]	←	tom[math-ability → good], tom[favourite-subject → math]
$G8$:	tom[math-ability → average]	←	tom[major → math], tom[math-grade → b]

[2] This result is analogous to and inspired by the corresponding result for inheritance-stratified Gulog programs [5]. Its proof is given completely in [9].

The ground clauses $G6$ and $G8$ are considered as definitions of the method math-ability inherited from the types student and gr-student, respectively. As gr-student is more specific than student, $G8$ is supposed to dominate $G6$. Then, every inheritance stratification of P_5 requires that the ground atom tom[major $\rightarrow$ math] must be in a stratum which is lower than the stratum containing it, which is a contradiction. Hence P_5 is not inheritance-stratified.

Observe that $G8$ dominates $G6$, but $G8$ also depends on $G6$; that is, the activation of $G6$ results in the activation of $G8$, which is supposed to override $G6$. Therefore, it is not reasonable to use any of them. As a consequence, none of the conclusions of $G6$, $G7$ and $G8$ should be derived. However, it can be shown that each o-model of P_5 contains both tom[major $\rightarrow$ math] and tom[math-ability $\rightarrow$ average]. So every o-model of P_5 does not serve as its reasonable meaning.

Now consider the proposed semantics. It is simple to see that $\mathcal{M}_{P_5}^{\mathrm{GE}}$ is the set {tom[math-grade $\rightarrow$ b], tom[favourite-subject $\rightarrow$ math]}, which is the correct skeptical meaning of P_5 (*i.e.*, the meaning obtained in the usual way after discarding the conflicting clauses $G6$ and $G8$). □

5 Related Works and Conclusions

Defeasible inheritance has been intensively studied in the context of inheritance networks [7,12,13]. Although the process of drawing conclusions from a set of defeasible hypotheses in inheritance networks is quite different from the process of deduction (as pointed out in [7]) and these works do not discuss dynamic method resolution, they do provide the presented approach with a foundation for determining the domination relation among ground clauses. A type hierarchy and a membership relation can be represented as a network, and the domination relation can then be determined based on the topological structure of the network. For example, if there exists a path from an object o through a type t to a type t' in the network, then it is natural to suppose that the ground method definitions for o inherited from t dominate those inherited from t'.

Besides [5], distinguished proposals that incorporate inheritance in the context of logic-based deduction systems include [1,2,3,8]. However, in [1] and [8], inheritance is realized by other means than typed substitution; *i.e.*, [1] captures inheritance by transforming subclass relationships into rules of the form $class(X) \leftarrow subclass(X)$, and [8] models inheritance as implicit implication on interpretation domains (called H-structures). [2] and [3] incorporate inheritance into unification algorithms but do not discuss nonmonotonic inheritance.

This paper studies the interaction of inheritance, realized by means of typed substitution, and deduction, and proposes a framework for discussing a declarative semantics for definite declarative programs with nonmonotonic inheritance. The framework uses a domination relation on program ground clauses, specifying their priority, as additional information for resolving conflicting method definitions. With a specified domination relation, a program is transformed into an argumentation framework which provides an appropriate structure for analyzing the interrelation between the intended deduction and domination. The meaning of the program is defined based on the grounded extension of this argumentation framework. Method resolution in the framework is dynamic with respect to the applicability of methods. The paper not only shows that the proposed semantics and Dobbie and Topor's perfect model (with overriding) semantics [5] coincide for inheritance-stratified programs (Theorem 3), but also claims that the proposed semantics provides correct skeptical meanings for non-inheritance-stratified programs.

Acknowledgement

The proposed semantics is inspired by Phan Minh Dung. This work is supported in part by the Thailand Research Fund.

References

1. Abiteboul, S., Lausen, G., Uphoff, H., Waller, E.: Methods and Rules. In: Proceedings of the 1993 ACM SIGMOD International Conference on the Management of Data. ACM Press (1993) 32–41
2. Aït-Kaci, H., Nasr, R.: LOGIN: A Logic Programming Language with Built-in Inheritance. The Journal of Logic Programming **3** (1986) 185–215
3. Aït-Kaci, H., Podelski, A.: Towards a Meaning of Life. The Journal of Logic Programming **16** (1993) 195–234
4. Akama, K.: Declarative Semantics of Logic Programs on Parameterized Representation Systems. Advances in Software Science and Technology **5** (1993) 45–63
5. Dobbie, G., Topor, R.: On the Declarative and Procedural Semantics of Deductive Object-Oriented Systems. Journal of Intelligent Information Systems **4** (1995) 193–219
6. Dung, P.M.: On the Acceptability of Arguments and Its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-Person Games. Artificial Intelligence **77** (1995) 321–357
7. Horty, J.F., Thomason, R.H., Touretzky, D.S.: A Skeptical Theory of Inheritance in Nonmonotonic Semantic Networks. Artificial Intelligence **42** (1990) 311–348
8. Kifer, M., Lausen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the Association for Computing Machinery **42** (1995) 741–843
9. Nantajeewarawat, E.: An Axiomatic Framework for Deductive Object-Oriented Representation Systems Based-on Declarative Program Theory. PhD thesis, CS-97-7, Asian Institute of Technology, Bangkok, Thailand (1997)
10. Nantajeewarawat, E., Wuwongse, V.: Defeasible Inheritance Through Specialization. Technical Report, Computer Science and Information Management Program, Asian Institute of Technology, Bangkok, Thailand (1999)
11. Przymusinski, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. In: Minker, J. (ed.): Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1988) 193–216
12. Stein, L. A.: Resolving Ambiguity in Nonmonotonic Inheritance Hierarchies. Artificial Intelligence **55** (1992) 259–310
13. Touretzky, D.S.: The Mathematics of Inheritance. Morgan Kaufmann (1986)

Entailment of Non-structural Subtype Constraints

Joachim Niehren and Tim Priesnitz

Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany
www.ps.uni-sb.de/∼{niehren,tim}

Abstract. Entailment of subtype constraints was introduced for constraint simplification in subtype inference systems. Designing an efficient algorithm for subtype entailment turned out to be surprisingly difficult. The situation was clarified by Rehof and Henglein who proved entailment of structural subtype constraints to be coNP-complete for simple types and PSPACE-complete for recursive types. For entailment of non-structural subtype constraints of both simple and recursive types they proved PSPACE-hardness and conjectured PSPACE-completeness but failed in finding a complete algorithm. In this paper, we investigate the source of complications and isolate a natural subproblem of non-structural subtype entailment that we prove PSPACE-complete. We conjecture (but this is left open) that the presented approach can be extended to the general case.

Keywords: subtyping, constraints, entailment, automata.

1 Introduction

Subtyping is a natural concept in programming. This observation has motivated the design of programming languages featuring a system for subtype inference [8, 11, 2, 6, 18]. Simplification of typings turned out to be the key issue in what concerns the complexity of subtype inference systems [7, 19, 17]. Several authors proposed to simplify typings based on algorithms for *subtype entailment*, i.e. entailment of *subtype constraints* [22, 16]. First approaches towards subtype entailment seem to presuppose [22, 16] that the problem could be solved efficiently. But finding an efficient algorithm for subtype entailment turned out to surprisingly difficult [9, 20, 10, 18]. And in fact, it still remains open whether subtype entailment is decidable, even if restricted to an inexpressive type languages. The most prominent open problem is the decidability of entailment between non-structural subtype constraints.

Types. A simple type is finite *tree* built from a signature Σ of function symbols (i.e. a ground term over Σ). A recursive type is an infinite tree over Σ. Most typically, Σ contains the constants *int* and *real* and the binary function symbol $\times$ for pairing. The type of a pair of integers, for instance, is the finite tree *int*$\times$*int*. The signature Σ typically also provides constants $\bot$ and $\top$ for the *least type* and the *greatest type*.

Many further types are of interest for programming: contra-variant function types $\tau{\rightarrow}\tau'$, record types $\{f_1{:}\tau_1, \ldots, f_n{:}\tau_n\}$, intersection and union types, and polymorphic types. These types fall out of the scope of the present paper. In order to keep subtype entailment simple, we restrict ourself to types that are finite or infinite trees built from a signature $\Sigma \subseteq \{int, real, \times, \bot, \top\}$.

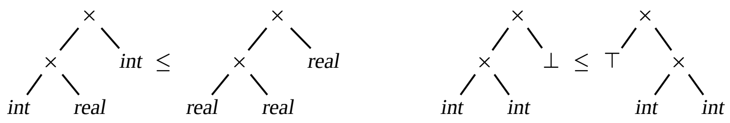


Fig. 1. Structural versus non-structural subtyping

Subtyping. When considering types as trees, subtyping becomes a partial order on trees. A typical subtype relationship is *int* $\leq$ *real* which states that every integer can be used as a real (its relevance is discussed in depth in [12]). The former subtype relationship induces *int*$\times$*int* $\leq$ *real*$\times$*real* which means that every pair of integers can be used as a pair of reals. Both relationships are *structural* in that they relate trees of the same shape. Subtyping becomes *non-structural* in the presence of a least type $\bot$ or greatest type $\top$, since $\bot\leq\tau$ and $\tau\leq\top$ hold for all types τ. The difference between structural and non-structural subtyping is illustrated in Figure 1.

Subtype Entailment. A subtype constraint is a logical description of types whose interpretation is based on the subtype relation. We assume a set of *type variables* ranged over by x, y, z. A *subtype constraint* ψ is a conjunction of ordering constraints $t\leq t'$ between terms t, t' built from variables and function symbols in Σ. *Subtype entailment* is the problem to decide whether an implication $\psi \rightarrow x\leq y$ is valid in the structure of trees, i.e. whether $\psi \models x\leq y$ holds. Four cases are to be distinguished: either we interpret over finite trees (simple types) or else over possibly infinite trees (recursive types); either we consider *non-structural subtyping* where $\bot, \top \in \Sigma$ or else *structural subtyping* where $\bot, \top \notin \Sigma$. The differences between these cases can be illustrated at the following example.

$$x\leq y\times y \wedge x\times x\leq y \models x\leq y$$

First, we consider structural subtyping with the signature $\Sigma = \{$*int*, *real*, $\times\}$. For finite trees, the left hand side is unsatisfiable and thus entailment holds. For infinite trees, there exists a unique solution where both x and y are mapped to the complete binary tree $\mu z.z\times z$; thus entailment holds again. Second, we consider non-structural subtyping with the signature $\Sigma = \{\top, \bot,$ *int*, *real*, $\times\}$. There are many more solutions than for structural subtyping. For instance, the variable assignment mapping x to $\bot\times(\top\times\bot)$ and y to $\top\times(\bot\times\top)$ is a solution of $x\leq y\times y \wedge x\times x\leq y$ which contradicts entailment of $x\leq y$ for both finite and infinite trees.

Open Problem. Early algorithms for subtype entailment were incomplete [22, 16, 18]. The situation was clarified by Henglein and Rehof who determined the complexity of structural subtype entailment: for simple types, it is coNP-complete [9] and for recursive types it is PSPACE-complete [10]. However, the complexity of non-structural subtype entailment remains open; it is at least PSPACE-hard, both, for finite and infinite trees [20, 10]. It is even unknown whether non-structural subtype entailment is decidable. Nevertheless, Rehof conjectures PSPACE-completeness (see Conjecture 9.4.5 of [20]).

Contribution. In this paper, we investigate the source of complications underlying non-structural subtype entailment. To this purpose, we introduce an extension of finite automata that we call *P-automata* and illustrate the relevance of P-automata to non-structural subtype entailment at a sequence of examples. P-automata can recognize nonregular and even non-context-free languages, as we show. This fact yields new insights into the expressiveness of non-structural subtype entailment.

Based on the insight gained by P-automata, we isolate a fragment of non-structural subtype constraints for which we prove decidability of entailment. We consider the signature $\{\bot, \top, \times\}$ and both cases, finite and possibly infinite trees respectively. The only restriction we require is that $\bot$ and $\top$ are not supported syntactically, i.e. that constraints such as $z{\times}\top{\leq}z$ are *not* cannot be written.

The algorithm we present is based on a polynomial time reduction to the universality problem of finite automata (which is PSPACE-complete). The idea is that more general P-automata are not needed for entailment of the restricted language. Our algorithm solves an entailment problem in PSPACE that was proved PSPACE-hard by Rehof and Henglein [10]. Its correctness proof is technically involved; it shows why nonregular sets of words – as recognized by P-automata – can be safely ignored.

Related Entailment Problems. Several entailment problems for constraint languages describing trees are considered in the literature. Two of them were shown PSPACE-complete in [14, 15]. The common property of these PSPACE-complete entailment problems is that entailment depends on properties of regular sets of words in the constraint graph. In contrast, nonregular sets have to be taken into account for non-structural subtype entailment.

In feature logics, several languages for describing feature trees (i.e. records types) have been investigated for entailment. Entailment for equality constraints over feature trees can be decided in quasi linear time [1, 21]. Ordering constraints over feature trees [5, 4] can be considered as record subtype constraints. Entailment of ordering constraints over feature trees can be solved in cubic time [13]. However, entailment with existential quantification is PSPACE-complete again [14].

Entailment has also been considered for set constraints (i.e. constraints for union and intersection types). Entailment of set constraints with intersection is proved DEXPTIME-complete in [3] for an infinite signature. Entailment of atomic set constraints [15] is proved PSPACE-complete in case of an infinite signature and DEXPTIME-hard for a finite signature.

2 Non-structural Subtype Constraints

We assume a signature Σ which provides function symbols denoted by f each of which has a fixed arity $\mathsf{ar}(f) \geq 0$. We require that Σ contains the constants $\bot$ and $\top$, i.e. $\mathsf{ar}(\bot) = \mathsf{ar}(\top) = 0$. We also assume an infinite set of variables ranged over by x, y, z, u, v, w.

Paths and Trees. A *path* is a word of natural numbers $n \geq 1$ that we denote by π, o, ϱ, or σ. The *empty path* is denoted by ε and the free-monoid *concatenation* of paths π and π' by juxtaposition $\pi\pi'$, with the property that $\varepsilon\pi = \pi\varepsilon = \pi$. A *prefix* of a path π is a path π' for which there exists a path π'' such that $\pi = \pi'\pi''$. A *proper prefix* of π is a prefix of π but not π itself. If π' is a prefix of π then we write $\pi' \leq \pi$ and if π' is a proper prefix of π then we write $\pi' < \pi$. The *prefix closure* of a set of path Π is denoted as *pr*(Π), i.e. *pr*$(\Pi) = \{\pi \mid \text{exists } \pi' \in \Pi : \pi \leq \pi'\}$ and its *proper prefix closure* with *pr*$_{\neq}(\Pi)$, i.e. *pr*$_{\neq}(\Pi) = \{\pi \mid \text{exists } \pi' \in \Pi : \pi < \pi'\}$.

A *tree* τ is a pair (D, L) where D is a tree domain, i.e. a non-empty prefixed-closed set of paths, and $L : D \to \Sigma$ a (total) function determining the labels of τ. We denote the tree domain of a tree τ by D_τ and its labeling function with L_τ. We require that trees τ are *arity consistent*: for all paths $\pi \in D_\tau$ and natural numbers i: $1 \leq i \leq \mathsf{ar}(L_\tau(\pi))$ iff $\pi i \in \mathcal{D}_\tau$. A tree is *finite* if its tree domain is finite and *infinite* otherwise. We denote the set of all finite trees with *Tree*$_\Sigma^{fin}$ and the set of all trees with *Tree*$_\Sigma$.

Non-Structural Subtyping. Let $\leq_L$ be the least (reflexive) partial order on function symbols of Σ which satisfies for all $f \in \Sigma$:

$$\bot \ \leq_L \ f \ \leq_L \ \top$$

We define *non-structural subtyping* as a partial order $\leq$ on trees such that $\tau_1 \leq \tau_2$ holds for trees τ_1, τ_2 iff for all paths $\pi \in D_{\tau_1} \cap D_{\tau_2}$ it holds that $L_{\tau_1}(\pi) \leq_L L_{\tau_2}(\pi)$.

Let *NS*$_\Sigma$ be the structure with signature $\Sigma \cup \{\leq\}$ whose domain is the set *Tree*$_\Sigma$. Function symbols in Σ are interpreted as tree constructors and the relation symbol $\leq$ as non-structural subtyping (which we also denote by $\leq$). The structure *NS*$_\Sigma^{fin}$ is the restriction of *NS*$_\Sigma$ to the domain of finite trees *Tree*$_\Sigma^{fin}$.

A *term* t is either a variable or a construction $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, $f \in \Sigma$, and $n = \mathsf{ar}(f)$. Of course, $\bot$ and $\top$ are terms since they are constants in Σ. A *non-structural subtype constraint* over Σ is a conjunction of ordering constraints $t_1 \leq t_2$. We consider two cases for their interpretation, either the structure *NS*$_\Sigma$ or the structure *NS*$_\Sigma^{fin}$. We mostly use flattened constraints ψ of the following form:

$$\psi \ ::= \ x{=}f(x_1, \ldots, x_n) \mid x{\leq}y \mid \psi \wedge \psi' \qquad (f \in \Sigma,\ \mathsf{ar}(f) = n)$$

The omission of nested terms does not restrict the expressiveness of entailment. Terms on the left hand side of an entailment judgment can be flattened by introducing new variables for all subterms. Furthermore, $\psi \models t_1{\leq}t_2$ is equivalent to $\psi \wedge t_1{\leq}x \wedge y{\leq}t_2 \models x{\leq}y$ where x, y are fresh variables.

Satisfiability and Entailment. Let Φ denote a first-order formula built from ordering constraints with the usual first-order connectives and let $\mathcal{V}(\Phi)$ be the set of *free variables* in Φ. We write Φ' *in* Φ if there exists Φ'' such that $\Phi = \Phi' \wedge \Phi''$ up to associativity and commutativity of conjunction. Suppose that $\mathcal{A}$ is a structure with signature $\Sigma \cup \{\leq\}$. A *solution of* Φ in $\mathcal{A}$ is a variable assignment α into the domain of $\mathcal{A}$ such that Φ evaluates to true under $\mathcal{A}$ and α. We call Φ *satisfiable in* $\mathcal{A}$ if there exists a solution for Φ in $\mathcal{A}$. A formula Φ is *valid in* $\mathcal{A}$ if all variable assignments into the domain of $\mathcal{A}$ are solutions of Φ. A formula Φ *entails* Φ' in $\mathcal{A}$, written $\Phi \models_{\mathcal{A}} \Phi'$ if $\Phi \to \Phi'$ is valid in $\mathcal{A}$.

Restricted Language. Let Σ_2 be the signature $\{\bot, \top, \times\}$ where $\times$ is a binary function symbol. A *restricted subtype constraints* φ has the form:

$$\varphi \ ::= \ u{=}u_1{\times}u_2 \mid u_1{\leq}u_2 \mid \varphi_1 \wedge \varphi_2$$

The following restrictions are crucial for entailment as we will discuss later on: 1) The constraints $x{=}\bot$ and $x{=}\top$ are excluded. 2) The signature Σ_2 does not contain a unary

S0	if $x \in \mathcal{V}(\psi)$ then $x{\le}x$ *in* ψ
S1	if $x{\le}y$ *in* ψ and $y{\le}z$ *in* ψ then $x{\le}z$ *in* ψ
S2	if $x{=}f(x_1,\dots,x_n) \wedge x{\le}y \wedge y{=}f(y_1,\dots,y_n)$ *in* ψ then $\bigwedge_{i=1}^n x_i{\le}y_i$ *in* ψ
S3	not $x{=}f_1(x_1,\dots,x_n)$ *in* ψ, $x{\le}y$ *in* ψ, $y{=}f_2(y_1,\dots,y_n)$ *in* ψ, and $f_1 \not\le_L f_2$
S4	not $\bigwedge_{i=1}^n x_i{=}f(\dots,y_{i+1},\dots) \wedge y_{i+1}{=}x_{i+1}$ *in* ψ where $n \ge 1$ and $x_1 = x_{n+1}$

Table 1. Closure and Clash-freeness Properties: *S0-S3* for *NS*$_\Sigma$ and *S0-S4* for *NS*$_\Sigma^{fin}$

function symbol. Nevertheless, the restricted entailment problem is not trivial. It is not difficult to see and proved by Rehof and Henglein [10] that entailment of the restricted language can express universality of non-deterministic finite automata; thus:

Proposition 1 (Hardness). *Non-structural subtype entailment for the restricted constraint language is PSPACE hard for both structures NS*$_{\Sigma_2}$ *and NS*$_{\Sigma_2}^{fin}$*.*

We next recall a closure algorithm from [10] which decides the satisfiability of (unrestricted) non-structural subtype constraints ψ over an arbitrary signature Σ. In Table 1, a set of properties *S0-S4* is given. The properties for *NS*$_\Sigma^{fin}$ and *NS*$_\Sigma$ differ only in an additional occurs check for the case of finite trees (*S4*). Reflexivity and transitivity of subtype are required by (*S0*) and (*S1*). The decomposition property of subtyping is stated in (*S2*), and clash-freeness for labeling in (*S3*).

We call a (flattened) constraint ψ *closed* it it satisfies *S0-S2*. Properties *S0-S2* can also be considered as a saturation algorithm which computes the *closure* of a (flattened) constraint ψ in cubic time. A constraint ψ is *clash-free* for *NS*$_\Sigma$ it it satisfies *S3* and for *NS*$_\Sigma^{fin}$ if it satisfies *S3-S4*.

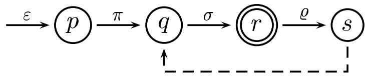
Proposition 2 (Satisfiability). *A constraint is satisfiable in NS*$_\Sigma$ *(resp. NS*$_\Sigma^{fin}$*) if its closure is clash-free for NS*$_\Sigma$ *(resp. NS*$_\Sigma^{fin}$*).*

3 P-Automata

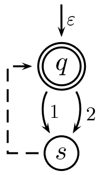
We now present the notion of a *P-automaton* on which we will base our analysis of subtype entailment. A P-automaton is an extension of a finite automaton with a new kind of edges. Let $\mathcal{A} = (A, Q, I, F, \Delta)$ be a finite automaton with *alphabet* A, *states* Q, *initial states* I, *final states* F, and *transition edges* Δ. We write $\mathcal{A} \vdash p \xrightarrow{\pi} q$ for states $p, q \in Q$ and $\pi \in A^*$ if the automaton $\mathcal{A}$ permits a transition from p to q while reading π. Thus, $\mathcal{A}$ recognizes the language $\mathcal{L}(\mathcal{A}) = \{\pi \in A^* \mid \mathcal{A} \vdash p \xrightarrow{\pi} q,\ p \in I, q \in F\}$.

Definition 3. *A* P-automaton $\mathcal{P}$ *is a pair* $(\mathcal{A}, P)$ *consisting of a finite automaton* $\mathcal{A} = (A, Q, I, F, \Delta)$ *and a set of* P-edges $P \subseteq Q \times Q$ *between the states of* $\mathcal{A}$*. The P-automaton* $\mathcal{P}$ *recognizes the language* $\mathcal{L}(\mathcal{P}) \subseteq A^*$ *given by:*

$$\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{A}) \cup \bigcup \{\pi(\sigma\varrho)^*\sigma \mid \mathcal{A} \vdash p \xrightarrow{\pi} q \xrightarrow{\sigma} r \xrightarrow{\varrho} s,\ (s,q){\in}P,\ p{\in}I,\ r{\in}F\}$$



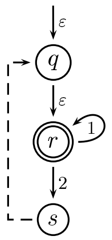
A P-automaton recognizes all words in the language of the underlying finite automaton. In addition, it is permitted to use P-edges as ε-edges, except that the first usage of a P-edge determines the period of the remaining word to be read (the period is $\sigma\varrho$ in Definition 3). We draw P-edges as dashed lines.



Example 4. Consider the P-automaton with alphabet $\{1, 2\}$, states $\{q, s\}$, initial and final state q, edges $q \xrightarrow{1} s$ and $q \xrightarrow{2} s$ and a single P-edge (s, q). The automaton can loop by using its P-edge multiply, but the first usage determines a period (the word 1 or 2) of the remaining word. Thus, the language recognized is $1^* \cup 2^*$ rather than $(1 \cup 2)^*$.

The length of a period fixed by a first usage of a P-edge needs not to be bounded by the number of states of the P-automaton. This fact raises the following problem.

Lemma 5 (Failure of context-freeness). *There exists a P-automaton whose language is not context-free (and thus not regular).*



Proof. We consider the P-automaton with alphabet $\{1, 2\}$, states $\{q, r, s\}$, initial states $\{q\}$, final states $\{r\}$, transition edges $q \xrightarrow{\varepsilon} r$, $r \xrightarrow{1} r$ and $r \xrightarrow{2} s$ and a single P-edge (r, q). This P-automaton is depicted to the right. It recognizes the language $\bigcup\{pr(\pi^*) \mid \pi \in 1^*2\}$. which is not context-free. Otherwise, the intersection with the regular language $1^*21^*21^*2$ would also be context-free. But this intersection is the language $\{1^n21^n21^n2 \mid n \geq 0\}$ which is clearly not context-free.

4 Path Constraints

We now introduce path constraints which express properties of subtrees at a given path. Path constraints are fundamental in understanding entailment for many languages of ordering constraints [9, 10, 20, 14, 13, 15].

If τ is a tree and $\pi \in D_\tau$ then we write $\tau.\pi$ for the *subtree of* τ *at* π, i.e. $D_{\tau.\pi} = \{\pi' \mid \pi\pi' \in D_\tau\}$ and $L_{\tau.\pi}(\pi') = L_\tau(\pi\pi')$ for all $\pi' \in D_{\tau.\pi}$. A *subtree constraint* $x.\pi{=}y$ requires that the domain of the denotation of x contains π and that its subtree at path π is the denotation of y.

Conditional path constraints of the first kind we use are of the form $x?\sigma \leq_L y?\pi$. The question mark indicates conditionality depending on the existence of a subtree. A path constraint $x?\sigma \leq_L y?\pi$ is solved by a variable assignment α if $\pi_1 \in D_{\alpha(x)}$ and $\pi_2 \in D_{\alpha(y)}$ implies $L_{\alpha(x)}(\pi_1) \leq_L L_{\alpha(y)}(\pi_2)$. We freely omit the conditionality $?\varepsilon$ since it ε path does always exist. We also write $x?\sigma \leq_L f$ instead of $\exists y(x?\sigma \leq_L y \wedge y{\leq}f(\top, \ldots, \top))$, and, symmetrically, $f \leq_L x?\sigma$ instead of $\exists y(f(\bot, \ldots, \bot){\leq}y \wedge y \leq_L x?\sigma)$.

Proposition 6 (Characterization of Entailment). *For all u, v the following equivalence is valid in NS_{Σ_n}:*

$$u{\leq}v \;\leftrightarrow\; \bigwedge\{u?\pi \leq_L v?\pi \mid \pi \textit{ a path}\}$$

We call a path π a *contradiction path for* $\psi \models x{\leq}y$ if and only if ψ does not entail $x?\pi \ \leq_L \ y?\pi$. In this terminology, Proposition 6 states that an entailment judgment $\psi \models x{\leq}y$ holds if and only if there exists no contradiction path for it.

We need further conditional path constraints of the form $x?\pi{\leq}^{pr}y$, $x{\leq}^{pr}y?o$, and $x?\pi{\leq}^{pr}y?o$ which do not only restrict x and y at the paths π and o but also at their prefixes. The semantics of these constraints is defined in Table 2. Note that the path con-

$$\begin{array}{rcl} x?\pi{\leq}^{pr}y & \leftrightarrow & x.\pi{\leq}y \vee \bigvee_{\pi'<\pi} x.\pi'{\leq}\bot \\ x{\leq}^{pr}y?o & \leftrightarrow & x{\leq}y.o \vee \bigvee_{o'<o} \top{\leq}y.o' \\ x?\pi{\leq}^{pr}y?o & \leftrightarrow & \exists u(x?\pi{\leq}^{pr}u \wedge u{\leq}^{pr}y?o) \end{array}$$

Table 2. Semantics of conditional path constraints

straint $x?\pi{\leq}^{pr}y$ entails $\exists z(x.\pi{=}z \rightarrow z{\leq}y)$ but not vice versa. The reason is $x?\pi{\leq}^{pr}y$ constrains x even if $x.\pi$ is not defined. For instance, the constraint $x{\leq}f(y)$ entails $x?1{\leq}^{pr}y$ which – if $x.1$ is not defined – requires $x{\leq}\bot$.

For a restricted signature, the semantics of conditional path constraints is much less ad hoc than it might seem at first sight. This is shown by Lemma 8 for the signature $\Sigma_n = \{\bot, \top, g\}$ where g is a function symbol with $\mathsf{ar}(g) = n$.

Lemma 7. *For* $n \geq 1$*, signature* $\Sigma_n = \{\bot, \top, g\}$*, paths* $\pi \in \{1, \ldots, n\}^*$ *and* $\pi' < \pi$*:* $u?\pi{\leq}^{pr}v \rightarrow u?\pi' \leq_L g$ *and* $u{\leq}^{pr}v?\pi \rightarrow g \leq_L v?\pi'$ *are valid in NS*$_{\Sigma_n}$*.*

Lemma 8 (Subtree versus conditional path constraints). *For* $n \geq 1$*, paths* $\pi \in \{1, \ldots, n\}^*$ *and variables* x, y *the following equivalences hold in the structure NS*$_{\Sigma_n}$*:*

$$\begin{array}{ll} x?\pi{\leq}^{pr}y \leftrightarrow \exists z(x{\leq}z \wedge z.\pi{=}y), & x{\leq}^{pr}y?\pi \leftrightarrow \exists z(z{\leq}y \wedge z.\pi{=}x) \\ u.\pi{=}v \leftrightarrow u?\pi{\leq}^{pr}v \wedge v{\leq}^{pr}u?\pi & \end{array}$$

Proof. We only prove the implication from the right to the left in the third equivalence. Assume that $u?\pi{\leq}^{pr}v \wedge v{\leq}^{pr}u?\pi$. For arbitrary $\pi' < \pi$, Lemma 7 proves the validity of $u?\pi' \leq_L g$ and $g \leq_L u?\pi'$. Thus, $u.\pi$ must be defined, and hence, $u.\pi = v$.

Lemma 9 (Strange loops raising P-edges). *For all variable* u, v*, all paths* $\sigma < \pi$*, and* $k \geq 0$ *the following implication is valid in NS*$_{\Sigma_n}$ *for all* $k \geq 0$*:*

$$u?\pi{\leq}^{pr}v \wedge u{\leq}^{pr}v?\pi \quad \rightarrow \quad u?\pi^k\sigma \leq_L v?\pi^k\sigma$$

Proof. By induction on k. Let $k = 0$. Since $\sigma < \pi$, Lemma 7 proves that $u?\pi{\leq}^{pr}v \wedge u{\leq}^{pr}v?\pi$ entails $u?\sigma \leq_L g \wedge g \leq_L v?\sigma$ which in turn validates $u?\sigma \leq_L v?\sigma$. Suppose $k > 0$ and that $u?\pi{\leq}^{pr}v \wedge u{\leq}^{pr}v?\pi$ is valid. By definition, $u?\pi{\leq}^{pr}v \leftrightarrow u.\pi{\leq}v \vee \bigvee_{\varrho<\pi} u.\varrho{\leq}\bot$ and $v{\leq}^{pr}v?\pi \leftrightarrow u{\leq}v.\pi \vee \bigvee_{\varrho<\pi} \top{\leq}u.\varrho$ hold. If there exists $\varrho < \pi$ such that $u.\varrho{\leq}\bot$ or $\top{\leq}u.\varrho$ then $u.\varrho{\leq}v.\varrho$ is entailed for some prefix of π. In this case, $u?\pi^k\sigma \leq_L v?\pi^k\sigma$ follows from Proposition 6. Otherwise, $u.\pi{\leq}v \wedge u{\leq}v.\pi$ is valid. Let u', v' be such that $u' = u.\pi$ and $v' = u.\pi$. Thus, $u'{\leq}v \wedge v.\pi{=}v'$ holds and entails $u'?\pi{\leq}^{pr}v'$ by Lemma 8. Symmetrically, $u'{\leq}^{pr}v'?\pi$ is entailed, too. The induction hypothesis yields $u'?\pi^{k-1}\sigma \leq_L v'?\pi^{k-1}\sigma$ and thus $u?\pi^k\sigma \leq_L v?\pi^k\sigma$.

Signature	$\Sigma_n = \{\bot, \top, g\}$	$\mathsf{ar}(g) = n$
Alphabet	$A_n = \{1, \ldots, n\}$	
States	$Q_\psi = \{(u,v) \mid u, v \in \mathcal{V}(\psi)\}$	
Intial States	$I_{xy} = \{(x,y)\}$	
Increase	$(u,v) \xrightarrow{\varepsilon} (u',v) \in \Delta_\psi$	if $u{\le}u'$ *in* ψ
Decrease	$(u,v) \xrightarrow{\varepsilon} (u,v') \in \Delta_\psi$	if $v'{\le}v$ *in* ψ
Decomposition	$\left.\begin{array}{l}(u,v) \xrightarrow{i} (u_i,v_i) \in \Delta_\psi \\ (u,v) \in F_\psi\end{array}\right\}$	if $\left\{\begin{array}{l}u{=}g(u_1,\ldots,u_n) \textit{ in } \psi, \\ v{=}g(v_1,\ldots,v_n) \textit{ in } \psi, \\ \text{and } i \in A_n\end{array}\right.$
Equality	$\left.\begin{array}{l}(u,u) \xrightarrow{i} (u,u) \in \Delta_\psi \\ (u,u) \in F_\psi\end{array}\right\}$	if $i \in A_n$
P−Edges	$((u,v),(v,u)) \in P_\psi$	if $u, v \in \mathcal{V}(\psi)$

Table 3. The finite automaton $\mathcal{A}_\psi = (A_n, Q_\psi, I_{xy}, F_\psi, \Delta_\psi)$ and P-automaton $(\mathcal{A}_\psi, P_\psi)$ for $\psi \models x{\le}y$

5 Entailment and P-Automata

We continue with the signature $\Sigma_n = \{\bot, \top, g\}$ where $\mathsf{ar}(g) = n$. We fix two variables x, y globally and consider a constraint ψ with $x, y \in \mathcal{V}(\psi)$. In Table 3, we define a finite automaton $\mathcal{A}_\psi$ and a P-automaton $\mathcal{P}_\psi = (\mathcal{A}_\psi, P_\psi)$ for the judgment $\psi \models x{\le}y$. Note that $\mathcal{A}_\psi$ and thus $\mathcal{P}_\psi$ depend on our global variables x and y.

The idea is that the P-automaton $\mathcal{P}_\psi$ recognizes all *safe* paths, i.e those paths that are not contradiction paths for $\psi \models x{\le}y$. In fact, the definition of $\mathcal{P}_\psi$ does not always achieve this goal. This is not a problem for the purpose of this paper since our theory will be based exclusively on the regular approximation of $\mathcal{P}_\psi$ provided by the finite automaton $\mathcal{A}_\psi$. Even though the construction rules given in Table 3 apply without further restriction to ψ, an automaton $\mathcal{P}_\psi$ may well be useless if ψ is not closed and clash-free, or contains $\bot$ and $\top$.

Given a constraint ψ over the signature Σ_n, the automata $\mathcal{P}_\psi$ and $\mathcal{A}_\psi$ constructed in Table 3 recognize words over the alphabet $\{1, \ldots, n\}$. Its states are pairs of variables (u,v) in $\mathcal{V}(\psi)$. The initial state is (x,y), i.e. the pair of variables for which entailment is tested. Ordering constraints $u{\le}v$ correspond to ε-transitions in the rules **Increase** and **Decrease**. The **Decomposition** rule permits transitions that read a natural number $i \in A_n$ and descend to the i-th child, in parallel for both variables in a state. States to which decomposition applies are final. The **Equality** rule requires that states (u,u) are final and permitted to loop into itself. The automaton $\mathcal{P}_\psi$ features **P-Edges** for switching the two variables in a state.

Proposition 10 (Soundness). *Given a constraint ψ with $x, y \in \mathcal{V}(\psi)$ and signature Σ_n where $n \ge 1$, no word recognized by the P-automaton $\mathcal{P}_\psi$ is a contradiction path for $\psi \models x{\le}y$.*

Proof. We first show that $\pi \in \mathcal{L}(\mathcal{A}_\psi)$ implies entailment $\psi \models x?\pi \le_L y?\pi$ to hold. Clearly, if $\mathcal{A}_\psi \vdash (u,v) \xrightarrow{\pi} (u',v')$ then ψ entails $u?\pi {\le}^{pr} u'$ and $v' {\le}^{pr} v?\pi$. If $\pi \in$

$x{\leq}y{\times}y \wedge x{\times}x{\leq}y \;\not\models x{\leq}y$

Entailment can be contradicted at paths 12, 21, . . .

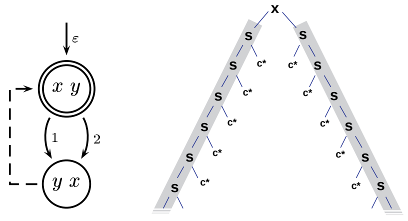


Fig. 2. The finite automaton and P-automaton for Example 11 and their languages

$\mathcal{L}(\mathcal{A}_\psi)$ due to a transition $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\pi} (u, v) \in F_\psi$ which ends in a final state (u, v) created by the **Decomposition** rule, then $\pi i \in \mathcal{L}(\mathcal{A}_\psi)$ for all $i \in A_n$. Thus, ψ entails $x?\pi i{\leq}^{pr}u$ and $v{\leq}^{pr}y?\pi i$ for some variables u, v which in turn entails $x?\pi \leq_L y?\pi$ (Lemma 7). If $\pi \in \mathcal{L}(\mathcal{A}_\psi)$ because a transition $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\pi} (u, u) \in F_\psi$ ends in a final state (u, u) contributed the **Equality** rule then ψ entails $x?\pi{\leq}^{pr}y?\pi$ and thus $x?\pi \leq_L y?\pi$.

It remains to verify that P-edges cannot contribute a contradiction path. If a path is contributed by a P-edge to $\mathcal{L}(\mathcal{P}_\psi)$ then it has the form $\pi(\sigma\varrho)^k\sigma$ such that $\mathcal{A}_\psi \vdash (x, y) \xrightarrow{\pi} (u, v) \xrightarrow{\sigma\tau} (v, u)$ for some $u, v \in \mathcal{V}(\psi)$ (see Definition 3 and the **P-Edges** rule in Table 3). From $\mathcal{A}_\psi \vdash (u, v) \xrightarrow{\sigma\tau} (v, u)$ it follows that ψ entails $u?\sigma\varrho{\leq}^{pr}v \wedge u{\leq}^{pr}v?\sigma\varrho$. Thus, Lemma 9 on strange loops implies that ψ entails $u?(\sigma\varrho)^k\sigma \leq_L v?(\sigma\varrho)^k\sigma$. Since $\mathcal{A}_\psi \vdash (x, y) \xrightarrow{\pi} (u, v)$ it follows that ψ entails $x?\pi(\sigma\varrho)^k\sigma \leq_L y?\pi(\sigma\varrho)^k\sigma$, too.

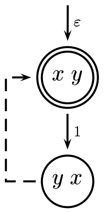
Example 11. For the signature Σ_2 the judgment φ_2: $x{\leq}y{\times}y \wedge x{\times}x{\leq}y \;\models x{\leq}y$ does not hold if x and y are distinct variables. Entailment is contradicted by the solution of φ_2 which maps x to $\bot{\times}(\top{\times}\bot)$ and y to $\top{\times}(\bot{\times}\top)$. The P-automaton $\mathcal{P}_{\varphi_2}$ illustrated in Figure 2 explains what happens. The finite automaton $\mathcal{A}_{\varphi_2}$ recognizes the language $\{\varepsilon\}$ only but $\mathcal{P}_{\varphi_2}$ has an additional P-edge from (y, x) to (x, y) by which it can also recognize the words in $1^+ \cup 2^+$. Since P-edges are not normal ε-edges, the P-automaton does not recognize the words 12 nor 21 which are in fact contradiction paths.

In Figures 2 and 3, we depict the language recognized by an P-automaton over the alphabet $\{1, \ldots, n\}$ as an n-ary tree: a word recognized by the underlying finite automaton corresponds to a node labeled by **x**, a word recognized by the additional P-edges only is indicated by a node labeled with **s** (for strange loop). All other words correspond to a node labeled with **c** (for contradiction).

Example 12. For the signature $\Sigma_1 = \{\bot, \top, g\}$ with $\mathsf{ar}(g) = 1$ the entailment judgment $\varphi_1 : x{\leq}g(y) \wedge g(x){\leq}y \;\models x{\leq}y$ holds. This might seem surprising since the only difference to Example 11 seems to be the choice of a unary versus a binary function symbol. The situation is again clarified when considering the P-automaton. The automaton $\mathcal{P}_{\varphi_1}$ is given in Figure 3. In contrast to $\mathcal{P}_{\varphi_2}$ in Figure 2, the alphabet of $\mathcal{P}_{\varphi_1}$ is the singleton $\{1\}$. Thus, its language $\mathcal{L}(\mathcal{P}_{\varphi_1}) = \{1\}^*$ is universal. Hence, there cannot be any contradiction path for $\varphi_1 \models x{\leq}y$, i.e. entailment holds.

$x \leq g(y) \wedge g(x) \leq y \models x \leq y$

Entailment holds.



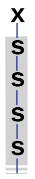
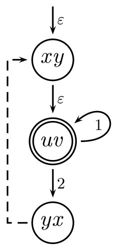


Fig. 3. The finite automaton and P-automaton for Example 12 and their languages

Examples 11 and 12 illustrate that P-edges have less effect on entailment in absence of unary function symbols. In fact, we show in this paper that P-edges do not have any effect on entailment for the restricted language. Even more importantly, this property depends on the restriction that constraints $u=\bot$ or $u=\top$ are not supported.

The context freeness failure for languages of P-automata has a counterpart for non-structural subtype entailment, even for the restricted language. This is illustrated by the judgment: $\varphi_3 : x \leq u \wedge v \leq y \wedge u = u \times y \wedge v = v \times x \models x \leq y$. The language $\mathcal{L}(\mathcal{P}_{\varphi_3})$ is not context-free since $\mathcal{P}_{\varphi_3}$ is exactly the P-automaton considered in the proof of Lemma 5. On the other hand side, the non-context free part of $\mathcal{L}(\mathcal{P}_{\varphi_3})$ does not force entailment to hold.



6 Deciding Entailment in PSPACE

We now show how to decide entailment for the restricted entailment problem with signature Σ_2. Our algorithm requires polynomial space and applies to both structures NS_{Σ_2} or $NS^{fin}_{\Sigma_2}$ respectively. The only difference is hidden in the satisfiability test used. Let *NS* be either of the two structures.

Proposition 13 (Characterization). *Let φ be a closed (restricted) constraint with $x, y \in \mathcal{V}(\varphi)$ which is clash-free with respect to NS. Then the entailment judgment $\varphi \models x \leq y$ holds in NS if and only if the set $\mathcal{L}(\mathcal{A}_\varphi)$ is universal, i.e. $\mathcal{L}(\mathcal{A}_\varphi) = \{1,2\}^*$.*

Proof. If $\mathcal{L}(\mathcal{A}_\varphi) = \{1,2\}^*$ then no contradiction path for $\varphi \models x \leq y$ exists (Proposition 10) and hence $\varphi \models x \leq y$ holds (Proposition 6). Proving the converse (completeness) is much more involved. This proof is sketched in Section 8.

Theorem 14 (Decidability and Complexity). *Non-structural subtype entailment in the restricted language is PSPACE-complete for both structures NS_{Σ_2} and $NS^{fin}_{\Sigma_2}$.*

Proof. Proposition 1 claims that entailment is PSPACE-hard. For deciding $\varphi \models x \leq y$, we compute the closure of φ in polynomial time and check whether it is clash-free with respect to NS_{Σ_2} or $NS^{fin}_{\Sigma_2}$ respectively (Proposition 2). For closed and clash-free φ, entailment holds if and only if $\mathcal{L}(\mathcal{A}_\varphi)$ is universal (Proposition 13). This can be checked in PSPACE since $\mathcal{A}_\varphi$ is a finite automaton which can be constructed from φ in (deterministic) polynomial time.

7 Completeness Proof

We prove the completeness of the characterization of entailment in Proposition 13. For a constraint φ of the restricted language, the idea is that we can freely extend the P-automaton $(\mathcal{A}_\varphi, P_\varphi)$ with additional P-edges without affecting universality. This motives to consideration of a language *Trace*$_\varphi$ which is recognized by the P-automaton $(\mathcal{A}_\varphi, Q_\varphi \times Q_\varphi)$ where Q_φ is the set of all states of $\mathcal{A}_\varphi$.

Definition 15. *We define the set Base*$_\varphi$ *of* bases *and Trace*$_\varphi$ *of* traces *of* $\varphi \models x \leq y$ *by:*

$$\mathit{Base}_\varphi = \{\pi \mid \exists u, v : \mathcal{A}_\varphi \vdash (x,y) \xrightarrow{\pi} (u,v)\}$$
$$\mathit{Trace}_\varphi = \bigcup \{\mathit{pr}(o\pi^*) \mid o\pi \in \mathit{Base}_\varphi\}$$

Lemma 16. *The set* $\mathit{pr}_{\neq}(\mathit{Base}_\varphi)$ *is equal to the set* $\mathcal{L}(\mathcal{A}_\varphi)$.

Proof. Showing that $\varepsilon \in \mathcal{L}(\mathcal{A}_\varphi)$ implies $\varepsilon \in \mathit{pr}_{\neq}(\mathit{Base}_\varphi)$ is left to the reader. If $\pi i \in \mathcal{L}(\mathcal{A}_\varphi)$ then $\mathcal{A}_\varphi \vdash (x,y) \xrightarrow{\pi i} (u,v)$ for some (final) state (u,v). Thus, $\pi i \in \mathit{Base}$, i.e. $\pi \in \mathit{pr}_{\neq}(\mathit{Base}_\varphi)$. For the converse, assume $\pi \in \mathit{pr}_{\neq}(\mathit{Base}_\varphi)$. Hence, $\pi i \in \mathit{Base}_\varphi$ for some i. There exists transitions $\mathcal{A}_\varphi \vdash (x,y) \xrightarrow{\pi} (u,v) \xrightarrow{i} (u',v')$ with a final step done by the **Decomposition** rule in Table 3. Thus, $(u,v) \in F_\varphi$, i.e. $\pi \in \mathcal{L}(\mathcal{A}_\varphi)$.

Lemma 16 implies that $\mathcal{L}(\mathcal{P}_\varphi) \subseteq \mathit{Trace}_\varphi$. The next proposition states that if $\mathcal{L}(\mathcal{A}_\varphi)$ is not universal then neither *Trace*$_\varphi$ nor $\mathcal{L}(\mathcal{P}_\varphi)$ are universal.

Proposition 17 (Escape). *If* $\sigma \notin \mathcal{L}(\mathcal{A}_\varphi)$ *then there is a path* $\varrho \notin \mathit{Trace}_\varphi$ *with* $\sigma \leq \varrho$.

Proof. We assume $\sigma \notin \mathcal{L}(\mathcal{A}_\varphi)$ and define $\varrho := \sigma 1^{|\sigma|} 2$ where $|\sigma|$ denotes the length of σ and 1^n a word which consists of exactly n letters 1. We prove $\varrho \notin \mathit{Trace}_\varphi$ by contradiction. Suppose that $\varrho \in \mathit{Trace}_\varphi$. By definition there exists paths o, π such that $o\pi \in \mathit{Base}_\varphi$ and $\varrho \in \mathit{pr}(o\pi^*)$. Hence $\sigma \in \mathit{pr}(o\pi^*)$ such that either $\sigma < o\pi$ or $o\pi \leq \sigma$. It it not possible that $\sigma < o\pi$ since otherwise, $\sigma \in \mathit{pr}_{\neq}(o\pi) \subseteq \mathit{pr}_{\neq}(\mathit{Base}_\varphi)$ which by Lemma 16 contradicts $\sigma \notin \mathcal{L}(\mathcal{A}_\varphi)$. Hence $o\pi \leq \sigma$ such that $\sigma = o\pi\sigma_0$ for some path σ_0. In combination with $\varrho = \sigma 1^{|\sigma|} 2 \in \mathit{pr}(o\pi^*)$ this yields $\sigma_0 1^{|\sigma|} 2 \in \mathit{pr}(\pi^*)$. Furthermore, $|\pi| \leq |o\pi| \leq |\sigma|$. The key point comes now: $\sigma_0 1^{|\sigma|} 2 \in \mathit{pr}(\pi^*)$ and $|\pi| \leq |\sigma|$ imply $\pi \in 1^*$ which is impossible since π must contain the letter 2. Hence, $\varrho \notin \mathit{Trace}_\varphi$.

Lemma 18 (Contradiction). *Let* φ *be closed and clash-free,* $o \notin \mathcal{L}(\mathcal{A}_\varphi)$*, and* $\varrho \notin \mathit{Trace}_\varphi$*: if* $o \leq \varrho$ *then* ϱ *is a contradiction path for* $\varphi \models x \leq y$ *in NS.*

Proof of Proposition 13 continued (Completeness). If $\mathcal{L}(\mathcal{A}_\varphi)$ is not universal then there exists a path $o \leq \varrho$ such that $o \notin \mathcal{L}(\mathcal{A}_\varphi)$ and $\varrho \notin \mathit{Trace}_\varphi$ according to the Escape Proposition 17. By Lemma 18, there exists a contradiction path which proves that entailment $\varphi \models x \leq y$ cannot hold.

$\psi \vdash x?\varepsilon {\leq^{pr}} y$	if $x{\leq}y$ *in* ψ
$\psi \vdash x?\pi i {\leq^{pr}} y$	if $\psi \vdash x?\pi {\leq^{pr}} z$ and $z{=}f(z_1, \dots z_i \dots, z_n),\ z_i{\leq}y$ *in* ψ
$\psi \vdash x {\leq^{pr}} y?\varepsilon$	if $x{\leq}y$ *in* ψ
$\psi \vdash x {\leq^{pr}} y?\pi i$	if $\psi \vdash z {\leq^{pr}} y?\pi$ and $z{=}f(z_1, \dots z_i \dots, z_n), x{\leq}z_i$ *in* ψ
$\psi \vdash x?\pi {\leq^{pr}} y?\pi'$	if $\exists z : \psi \vdash x?\pi {\leq^{pr}} z$ and $\psi \vdash z {\leq^{pr}} y?\pi'$

Table 4. Syntactic Support

8 Proof of the Contradiction Lemma

In a first step, we refine the contradiction Lemma 18 into Lemma 21. This requires a notion of *syntactic support* that is given in Table 4. If μ is a path constraint then the judgment $\varphi \vdash \mu$ reads as 'φ supports μ syntactically'. Syntactic support for φ refines judgments performed by the finite automaton $\mathcal{A}_\varphi$. For instance, it holds for a closed and clash-free constraint φ that $\varphi \vdash x?\pi {\leq^{pr}} y?\pi$ iff $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\pi} (u, u)$. Judgments like $\varphi \vdash x?\pi {\leq^{pr}} y$ or $\varphi \vdash x?\pi {\leq^{pr}} y?\pi'$ cannot be expressed by $\mathcal{A}_\varphi$.

Lemma 19. *For all path constraints μ if $\varphi \vdash \mu$ then $\varphi \models \mu$ holds.*

Definition 20. *We define two functions l_φ and r_φ for the judgment $\varphi \models x{\leq}y$.*

$$\begin{array}{ll} l_\varphi(\sigma) = \max\{\pi \mid \pi{\leq}\sigma \wedge \exists u.\varphi \vdash x?\pi 1 {\leq^{pr}} u\} & \textit{(left)} \\ r_\varphi(\sigma) = \max\{\pi \mid \pi{\leq}\sigma \wedge \exists v.\varphi \vdash v {\leq^{pr}} y?\pi 1\} & \textit{(right)} \end{array}$$

Note that if $l_\varphi(\sigma) \leq r_\varphi(\sigma)$ then $l_\varphi(\sigma)$ is the maximal prefix of σ in $\mathcal{L}(\mathcal{A}_\varphi)$. Symmetrically, if $r_\varphi(\sigma) \leq l_\varphi(\sigma)$ then $r_\varphi(\sigma)$ is the maximal prefix of σ in $\mathcal{L}(\mathcal{A}_\varphi)$.

Lemma 21 (Contradiction refined). *Let φ be a closed and clash-free constraint and $o{\leq}\varrho$ paths such that $o \notin \mathcal{L}(\mathcal{A}_\varphi)$ and $\varrho \notin Trace_\varphi$.*

1. *if $l_\varphi(\varrho) \leq r_\varphi(\varrho)$ then $\varphi \wedge x.\varrho{=}\top \wedge y.\varrho{=}_L\times$ is satisfiable.*
2. *if $l_\varphi(\varrho) > r_\varphi(\varrho)$ then $\varphi \wedge x.\varrho{=}\times \wedge y.\varrho{=}_L\bot$ is satisfiable.*

Trivially, Lemma 21 subsumes the contradiction Lemma 18. The proof of Lemma 21 captures the rest of this section. Since both of its cases are symmetric we restrict ourself to the first one. We assume that φ is closed and clash-free and satisfies $x, y \in \mathcal{V}(\varphi)$. Given a fresh variable u we define a constraint $s(\varphi, \varrho)$ that is satisfaction equivalent to $\varphi \wedge x.\varrho{=}u \wedge y.\varrho{=}_L\times$ and in addition closed and clash-free.

Definition 22. *We call a set $D \subseteq \{1, 2\}^*$* domain closed *if D is prefixed-closed and satisfies the following property for all $\pi \in \{1, 2\}^*$: $\pi 1 \in D$ iff $\pi 2 \in D$. The* domain closure *$dc(D)$ is the least domain closed set containing D.*

Definition 23 (Saturation). *Let φ be a constraint, $x, y \in \mathcal{V}(\varphi)$, and $\varrho \in \{1,2\}^*$. For every $z \in \{x, y\}$ and $\pi \in dc(\{\varrho 1, \varrho 2\})$ let q^z_π be a fresh variable and $W(\varphi, \varrho)$ the collection of these fresh variables. The saturation $s(\varphi, \varrho)$ of φ at path ϱ is the constraint of minimal size satisfying properties a-f:*

a. φ in $s(\varphi, \varrho)$
b. *for all* $q^z_\pi \in W(\varphi, \varrho)$: $q^z_\pi{=}q^z_{\pi 1}{\times}q^z_{\pi 2}$ in $s(\varphi, \varrho)$ *if* $\pi < \varrho$
c. $q^y_\varrho{=}q^y_{\varrho 1}{\times}q^y_{\varrho 2}$ in $s(\varphi, \varrho)$
d. *for all* $q^z_\pi \in W(\varphi, \varrho), u \in \mathcal{V}(\varphi)$: $q^z_\pi{\le}u$ in $s(\varphi, \varrho)$ *if* $\varphi \vdash z?\pi{\le}^{pr}u$
e. *for all* $q^z_\pi \in W(\varphi, \varrho), u \in \mathcal{V}(\varphi)$: $u{\le}q^z_\pi$ in $s(\varphi, \varrho)$ *if* $\varphi \vdash u{\le}^{pr}z?\pi$
f. *for all* $q^z_{o\pi}, q^{z'}_{o'\pi} \in W(\varphi, \varrho)$: $q^z_{o\pi}{\le}q^{z'}_{o'\pi}$ in $s(\varphi, \varrho)$ *if* $\varphi \vdash z?o{\le}^{pr}z'?o'$

Lemma 24. *If φ is closed and clash-free then $s(\varphi, \varrho)$ is also closed and clash-free.*

Lemma 24 would go wrong for unrestricted constraints containing $\bot$ or $\top$. Its proof is not difficult but tedious since it requires a lot of case distinctions. We omit it for lack of space. Instead we note the following lemma which despite of its simplicity will turn out to be essential.

Lemma 25. *Let σ and o be paths with $\sigma{\le}o\sigma$. If $o{\ne}\varepsilon$ then $\sigma \in pr(o^*)$.*

The proof of Lemma 25 is simple and thus omitted. We can now approach the final step in which we show that $s(\varphi, \varrho) \wedge q^x_\varrho{=}\top$ is also closed and clash-free. Closedness follows trivially from Lemma 24 but clash-freeness requires work. The only clash rule which might possibly apply is S3. Since $\bot$ does not occur in $s(\varphi, \varrho)$, S3 can only be applied with $q^x_\varrho{=}\top$ and $\top{\not\le}_L{\times}$, i.e. if there are $w, w_1, w_2 \in \mathcal{V}(s(\varphi, \varrho))$ such that:

$$w{=}w_1{\times}w_2 \text{ in } s(\varphi, \varrho) \text{ and } q^x_\varrho{\le}w \text{ in } s(\varphi, \varrho)$$

We have to distinguish all possible choices of $w \in \mathcal{V}(s(\varphi, \varrho))$ but restrict ourself to the more interesting cases where $w \in W(\varphi, \varrho)$. In this case, $q^x_\varrho{\le}w$ was added to $s(\varphi, \varrho)$ by rule f in Definition 23. Since $w{=}w_1{\times}w_2$ *in* $s(\varphi, \varrho)$ and $w \in W(\varphi, \varrho)$ it follows that $w = q^y_\varrho$, or $w = q^z_\pi$ for some $\pi < \varrho$ and $z \in \{x, y\}$.

1. **Case $w = q^y_\varrho$:** Rule f requires $\varphi \vdash x?\sigma{\le}^{pr}y?\sigma$ for some prefix $\sigma{\le}\varrho$. This is equivalent to $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\sigma} (u, u)$ for some u. The **Equality** rule in the automaton construction yields $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\varrho} (u, u)$. Thus, $\varrho \in \mathcal{L}(\mathcal{A}_\varphi)$ which contradicts $\varrho \notin Trace_\varphi$.
2. **Case $w = q^z_\pi$ where $\pi < \varrho$ and $z \in \{x, y\}$:** Rule f requires the existence of σ, ϱ', π' such that $\varphi \vdash x?\varrho'{\le}^{pr}z?\pi'$ where $\varrho = \varrho'\sigma$ and $\pi = \pi'\sigma$. From $\pi < \varrho$ it follows that $\pi'\sigma < \varrho'\sigma$ and thus $\pi' < \varrho'$. Let $o \ne \varepsilon$ be such that $\pi'o = \varrho'$. Thus, $\pi'\sigma < \pi'o\sigma$ which in turn yields $\sigma < o\sigma$. The key point comes now. We can apply Lemma 25 in order to deduce $\sigma \in pr(o^*)$. Hence $\varrho = \varrho'\sigma = \pi'o\sigma \in \pi'opr(o^*) \subseteq pr(\pi'o^*)$. Since $\varphi \vdash x?\varrho'{\le}^{pr}z?\pi'$ there exists u such that $\varphi \vdash x?\varrho'{\le}^{pr}u$; together with our assumption $l_\varphi(\varrho) \le r_\varphi(\varrho)$ it follows that $\mathcal{A}_\varphi \vdash (x, y) \xrightarrow{\varrho'} (u, v)$ for some u, v. Hence, $\varrho' \in Base_\varphi$, i.e. $\pi'o \in Base_\varphi$. Combined with $\varrho \in pr(\pi'o^*)$, we obtain $\varrho \in Trace_\varphi$ in contradiction to our assumption.

$$\left.\begin{array}{l} x{\leq}y{\times}y \wedge \\ z{\times}y{\leq}y \;\wedge \\ z{\times}\top{\leq}z \end{array}\right\} \models x{\leq}y$$

Entailment holds.

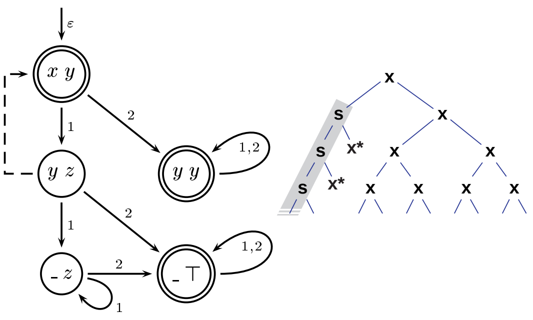


Fig. 4. An example for the general case

9 Conclusion and Future Work

We have solved the problem of non-structural subtype entailment over the signature $\{\bot, \top, \times\}$ for the restricted language where $\bot$ and $\top$ are not supported syntactically. We have proved PSPACE-completeness both for simple and recursive types. We have presented the notion of a P-automaton and illustrated its importance for understanding non-structural subtype entailment. Because of its P-edges a P-automaton can recognize non context-free languages. In what concerns non-structural subtype entailment for the restricted language, we have proved that non regularity can be safely ignored.

We believe that our methods can be extended to the full problem of non-structural subtype entailment. However, the full problem may well turn out to be more complex then PSPACE-complete. More research is needed to answer this question finally. The main problem in the general case is that we have to take P-edges into account. This is illustrated by the following example:

$$\varphi_4\colon x{\leq}y{\times}y \wedge z{\times}y{\leq}y \;\wedge z{\times}\top{\leq}z \;\models x{\leq}y$$

Entailment holds even though the language of finite automaton for φ_4 given in Figure 4 is not universal. The construction rules for this automaton are more involved than in Table 3 since $\top$ has to be accounted for. A P-edge from (x, y) to (y, z) has to be added even though only one of the two variables is switched.

References

[1] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, Jan. 1994.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[3] W. Charatonik and A. Podelski. Set constraints with intersection. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pages 352–361, Warsaw, Poland, 1997.

[4] J. Dörre. Feature logics with weak subsumption constraints. In *Annual Meeting of the ACL (Association of Computational Logics)*, pages 256–263, 1991.

[5] J. Dörre and W. C. Rounds. On subsumption and semiunification in feature algebras. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 300–310, 1990.

[6] J. Eifrig, S. Smith, and V. Trifonow. Sound polymorphic type inference for objects. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1995.

[7] J. Eifrig, S. Smith, and V. Trifonow. Type inference for recursively constrained types and its application to object-oriented programming. *Elec. Notes in Theoretical Computer Science*, 1, 1995.

[8] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73, 1990.

[9] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pages 362–372, Warsaw, Poland, 1997.

[10] F. Henglein and J. Rehof. Constraint automata and the complexity of recursive subtype entailment. In *Proceedings of the 25th Int. Conf. on Automata, Languages, and Programming*, LNCS, 1998.

[11] J. C. Mitchell. Type inference with simple subtypes. *The Journal of Functional Programming*, 1(3):245–285, July 1991.

[12] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, MA, 1996.

[13] M. Müller, J. Niehren, and A. Podelski. Ordering constraints over feature trees. *Constraints, an International Journal, Special Issue on CP'97*, 5(1–2), Jan. 2000. To appear.

[14] M. Müller, J. Niehren, and R. Treinen. The first-order theory of ordering constraints over feature trees. In *IEEE Symposium on Logic in Computer Science*, pages 432–443, 21-24 June 1998.

[15] J. Niehren, M. Müller, and J.-M. Talbot. Entailment of atomic set constraints is PSPACE-complete. In *IEEE Symposium on Logic in Computer Sience*, 2–5, July 1999. to appear.

[16] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, New York, May 1996.

[17] F. Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 228–238, Sept. 1998.

[18] F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Institut de Recherche d'Informatique et d'Automatique, 1998.

[19] J. Rehof. Minimal typings in atomic subtyping. In *ACM Symposium on Principles of Programming Languages*. ACM Press, 1997.

[20] J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, University of Copenhagen, 1998.

[21] G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, Apr. 1994.

[22] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365, Aachen, 1996.

A CPS-Transform of Constructive Classical Logic

Ichiro Ogata

Electrotechnical Laboratory
1-1-4 Umezono Tsukuba 305-8568 JAPAN
`ogata@etl.go.jp`, `http://www.etl.go.jp/~ogata`

Abstract. We show that the cut-elimination for LKT, as presented in Danos et al.(1993), simulates the normalization for classical natural deduction(CND). Particularly, the denotation for CND inherits the one for LKT. Moreover the transform from CND proof (i.e., Parigot's $\lambda\mu$-term) to LKT proof can be considered as a classical extension to call-by-name (CBN) CPS-transform.

1 Introduction

What is LKT?: One constructive classical logic we consider is **LKT** presented by Danos, Joinet and Schellinx(DJS) [1]. It has long been thought that classical logic cannot be put to use for computational purposes. It is because, in general, the normalization process for the the proof of classical logic has a lot of critical pairs. The normalization is the cut-elimination in case of the the Gentzen's sequent-style classical logic: **LK**. It is the Gentzen's theorem that **LK** has a Strongly Normalizing (SN) cut-elimination procedure. However it is not Church-Rosser(CR). **LKT** is a variant of **LK** which is equiped with SN and CR cut-elimination procedure. We say **LK** is *constructive* in this sense. The SN and CR cut-elimination procedure is called as **tq-protocol**. The CR property of tq-protocol is recovered by adding some restrictions on logical rules to **LK**. Despite of the restrictions, soundness and completeness w.r.t. classical provability is still retained in **LKT**. **LKT** is "classical logic" in this sence.

What is Classical Natural Deduction?: The other constructive classical logic is classical natural deduction (**CND**) presented by Parigot [9]. Church's λ-calculus is widely accepted as the logical basis of functional programming. It is also well known that typed λ-calculus has Curry-Howard correspondence with intuitionistic natural deduction. Parigot extends this idea to a classical logic: **CND**. Its computational interpretation is a natural extension of call-by-name (CBN) λ-calculus, called $\lambda\mu$-calculus. The $\lambda\mu$-calculus, equiped with so called "structural reduction" rule, is known to be SN and CR [9]. This exactly means the normalization procedure (in the sence of Parigot) of **CND** is SN and CR. Therefore **CND** can also be considered as a constructive classical logic. Hereafter we refer to Parigot's $\lambda\mu$-calculus by $\lambda\mu_\mathrm{n}$ in order to put stress on the fact that it is CBN.

Our Work: In this paper, we show how **CND** and **LKT** are related. More precisely, the normalization for **CND** can be simulated by tq-protocol for **LKT** (simulation theorem). Therefore the SN and CR property of $\lambda\mu_{\mathrm{n}}$ is shown to be a consequence of that of tq-protocol. Moreover we show the transform from **CND** proof to **LKT** proof can be considered as an essence of CPS-transform which is of computer science interest.

This paper consists of two part. In first part, we present a new term calculus for **LKT**. Hereafter we refer to this term calculus as $\lambda\mu_{\mathrm{t}}$-calculus. By this method, **LKT** which is a purely proof-theoretical artifact is set in correspondence with a simple term rewriting system. In second part, we show how $\lambda\mu_{\mathrm{n}}$ can be simulated by $\lambda\mu_{\mathrm{t}}$. Our investigation tool is a transform: it inductively transform the $\lambda\mu_{\mathrm{n}}$-term into the $\lambda\mu_{\mathrm{t}}$-term. As we mentioned before, $\lambda\mu_{\mathrm{n}}$ can be considered as a classical extension to CBN λ-calculus. Also, we already revealed that **LKT** is the target language of the CBN CPS-transform[7]. In sum, this transform can be considered as a classical extension to Plotkin's CBN CPS-transform[10]. Note that Plotkin's result is on the untyped case. However, we restrict ourself to well-typed case. This is because our investigation is purely proof theoretical.

As far as the denotational semantics is concerned, the advantage of our method is clear. Our main result shows that $\lambda\mu_{\mathrm{n}}$ can be simulated by tq-protocol. Tq-protocol can be simulated further by λ-calculus[8]. Hence $\lambda\mu_{\mathrm{n}}$ are shown to have the denotation in cartesian closed category. This observation confronts the categorical semantics approach such as "control category" of Selinger[11] and "continuation category" of Hofmann and Streicher[6]. Moreover, by the DJS's simulation theorem for linear decoration, our $\lambda\mu_{\mathrm{n}}$ also inherits the denotation in linear logic's coherent space semantics. This fact seems to be overlooked in the community of categorical semantics.

Related Works: In the community of proof theory, our simulation theorem are considered informally by some people. Herbelin[5] published a detailed work on the term calculus of **LJT**, which is an intuitionistic fragment of **LKT**. He interpreted **LJT** proofs as programs and cut-elimination steps as reduction steps, just as we do here. DJS also mentioned to the relation between **CND** and **LKT** in [2](p.795). They refer to a manuscript for full details there, however, the manuscript is not open to the public yet. Hence it should be considered as an open problem. Moreover what is new to our work is how it is proved. We use the technique of CPS-transform. That is, we discover the relation between the constructive classical logic and CPS-transform. Needless to say, they are both important concept in each community.

In the community of the theory of programming languages, De Groote describes the CPS-transform of the $\lambda\mu_{\mathrm{n}}$ in [3]. However his method was merely syntactic; he dose not mention to the relation to the proof theory. That is, the second part of our work can be seen as a proof theoretical recast of his work. In addition, our simulation theorem establishes more clear relation. We describe how De Groote's work can be recovered within our framework at the end.

Table 1. Original Derivation Rules for **LKT**

$$\dfrac{}{A\,;\ \Rightarrow A}\,\text{Ax} \qquad \dfrac{A\,;\ \varGamma \Rightarrow \varDelta}{;\ \varGamma, A \Rightarrow \varDelta}\,\text{D}$$

$$\dfrac{\varPi\,;\ \varGamma \Rightarrow \varDelta}{\varPi\,;\ \varGamma, A \Rightarrow \varDelta}\,\text{LW} \quad \dfrac{\varPi\,;\ \varGamma, A, A \Rightarrow \varDelta}{\varPi\,;\ \varGamma, A \Rightarrow \varDelta}\,\text{LC} \quad \dfrac{\varPi\,;\ \varGamma \Rightarrow \varDelta}{\varPi\,;\ \varGamma \Rightarrow \varDelta, A}\,\text{RW} \quad \dfrac{\varPi\,;\ \varGamma \Rightarrow \varDelta, A, A}{\varPi\,;\ \varGamma \Rightarrow \varDelta, A}\,\text{RC}$$

$$\dfrac{\varPi\,;\ \varGamma_0 \Rightarrow \varDelta_0, A \qquad A\,;\ \varGamma_1 \Rightarrow \varDelta_1}{\varPi\,;\ \varGamma_0, \varGamma_1 \Rightarrow \varDelta_0, \varDelta_1}\,\text{h-cut} \quad \dfrac{;\ \varGamma_0 \Rightarrow \varDelta_0, A \qquad \varPi\,;\ \varGamma_1, A \Rightarrow \varDelta_1}{\varPi\,;\ \varGamma_0, \varGamma_1 \Rightarrow \varDelta_0, \varDelta_1}\,\text{m-cut}$$

$$\dfrac{B\,;\ \varGamma_0 \Rightarrow \varDelta_0 \qquad ;\ \varGamma_1 \Rightarrow \varDelta_1, A}{A \to B\,;\ \varGamma_0, \varGamma_1 \Rightarrow \varDelta_0, \varDelta_1}\,\text{L}\!\to \quad \dfrac{\varPi\,;\ \varGamma, A \Rightarrow \varDelta, B}{\varPi\,;\ \varGamma \Rightarrow \varDelta, A \to B}\,\text{R}\!\to$$

2 $\lambda\mu_{\mathrm{t}}$ for LKT

In the following, we use the word **derivation**, instead of *proof*, for a tree of derivation rules.

First of all, we quote the original derivation rules for **LKT** from [1] in Table 1. $\Rightarrow$ is the **entailment sign** of the calculus. We use **rhs** and **lhs** for the right-hand-side and left-hand-side of the entailment sign, respectively. The letter L/R stands for **Left** and **Right** introduction, D for **Dereliction**, C for **Contraction** and W for **Weakening**. Notice also that we use different names for cut rules. We use **m-cut** instead of “mid” and **h-cut** instead of “head”. This is needed to avoid confusion. Notice that, in their original definition, contexts are interpreted as multisets, and structural rules are explicit.

2.1 Indexed Logical Systems

Now we will explain our formulation of logical systems. We use **indexed formula** method which is firstly developed by Zucker to relate a Gentzen's sequent-style derivation and a λ-terms[12]. Parigot's follows his method to relate a $\lambda\mu_{\mathrm{n}}$-term and a **CND** derivation.

In order to relate a term and a derivation, we need some way to specify formulas. For this, we change the notion of context. We interpret the context as the set of the **indexed formulas**. Accordingly, by contrast to the original, **structural rules** are implicit. **Formulas** are that of first order propositional logic constructed from $\to$. We use same implication symbol between logical systems. **Indexed formula** is an ordered pair of a formula and an index. We assume there are denumerably many **λ-indices** (resp. **μ-indices**) ranged over $x, y, z \ldots$ (resp. $\alpha, \beta, \gamma, \ldots$). We write an indexed formula (A, x) as A^x and (A, α) as A^α. **Sequents** of each logical systems are of the form as follows:

$$\begin{aligned} \textbf{LK}:&\ \varGamma \Rightarrow \varDelta \\ \textbf{LKT}:&\ \varPi\,;\ \varGamma \Rightarrow \varDelta \\ \textbf{CND}:&\ \varGamma \Rightarrow \varDelta \end{aligned}$$

where $\varGamma$ is a **λ-context** which is a set of λ-indexed formulas. Similarly, $\varDelta$ is a **μ-context** which is a set of μ-indexed formulas. Comma means taking union as sets. Thus, the set $\varGamma_0 \cup \varGamma_1$ is denoted by "$\varGamma_0, \varGamma_1$" and $\{A^x\} \cup \varGamma$ by "$A^x, \varGamma$". $\varPi$ denotes at most one unindexed formula.

We only handle **multiplicative** rules in every logical system. That is, λ-contexts (μ-contexts) in the conclusion is the union of λ-contexts (μ-contexts respectively) in the premises. For example, in L$\rightarrow$ of **LK**:

$$\frac{\varGamma_0 \Rightarrow \varDelta_0, A^\alpha \qquad \varGamma_1, B^y \Rightarrow \varDelta_1}{(A \rightarrow B)^x, \varGamma_0, \varGamma_1 \Rightarrow \varDelta_0, \varDelta_1}\,\mathrm{L}\!\rightarrow$$

Hereafter, for readability, we only write **active** and **main** formulas, and omit contexts as follows:

$$\frac{\Rightarrow A^\alpha \qquad B^y \Rightarrow}{(A \rightarrow B)^x \Rightarrow}\,\mathrm{L}\!\rightarrow$$

In the above, A and B are active formulas, while $A \rightarrow B$ is main formula. As we interpret contexts as sets, occurrences of formulas with the same index are automatically contracted. One can interpret this that binary rules are always followed by appropriate explicit **contractions** which *rename* the indices to the same name. We also interpret axiom rules contain appropriate **weakenings** as contexts. Therefore, we say, **structural rules** are implicit. Notice that in **LKT**, application of structural rules are restricted within $\varGamma$ and $\varDelta$. Specifically $\varPi$ can not be introduced by weakening.

We use π for the derivation. **Initial index** is an index which appears for the first time in whole derivation. We assume all initial indices are distinct unless they are truly related (i.e., subject of further implicit contraction). This is possible, by introducing the "concatenation" to indices on every binary rules. See Zucker.

2.2 Raw $\lambda\mu_{\mathrm{t}}$-Terms

We introduce a new notion for $\lambda\mu_{\mathrm{t}}$-terms. In a word, it has a "bi-directional" form of application. They correspond to the two orientation of cut in Gentzen's sequent-style classical logic. We explain this in the next subsection. Accordingly we have two kind of β-contractions.

The raw $\lambda\mu_{\mathrm{t}}$-terms, ranged over s, t, u, etc., are defined as follows:

$s, t, u := x$	**λ-variable**
$\mid\ \lambda x^A.t$	**λ-abstraction**
$\mid\ \mu\alpha^A.s$	**μ-abstraction**
$\mid\ s\,t$	**t-application**
$\mid\ \langle\!\langle s\rangle\!\rangle t$	**q-application**
$\mid\ [\alpha]s$	**named-term**
$\mid\ (x, \beta).\,t$	**R-term**
$\mid\ h\langle s, u\rangle$	**L-term**.

In the above definition, t-application corresponds to the usual application (i.e., application in λ-calculus). q-application is the new form of application in *reverse direction.* We often omit the type of λ-variable (μ-name) in λ-abstraction(μ-abstraction), in case it is clear from context.

2.3 The Notion of Orientation of Cut

To begin with, let us introduce briefly the notion of **orientation of cut** in tq-protocol[2]. This can be explained through the consideration "why **LK** has critical pairs?". The most typical example of critical pairs are as follows:

$$\dfrac{\dfrac{\Rightarrow}{\Rightarrow A}\,\mathrm{RW} \qquad \dfrac{\Rightarrow}{A \Rightarrow}\,\mathrm{LW}}{\Rightarrow}\,\mathrm{cut}$$

This simple example shows the source of inconfluency. Obviously, we have to choose whether the subderivation of the *left* premise of the cut or that of the *right* premise is going to be erased. The premise is going to be erased because it is introduced by weakening. We get two inconfluent result according to the choice of the direction. This choice exactly is the **orientation** of cut. In **LKT**, the orientation of cut is fixed beforehand; two cut rules (h-cut or m-cut) represents the two orientation. Roughly speaking, this is how **LKT** recovers CR property.

Now, our intention is to express cut-elimination by β-reduction. Therefore we introduce new notation to express two orientation of cut as follows:

$$\dfrac{\boxed{s:\ \Rightarrow A^{\alpha}} \qquad t:\ A^{x} \Rightarrow}{(\lambda x^{A}.t)\ (\mu\alpha^{A}.s):\ \Rightarrow}\,\mathrm{t} \qquad\qquad \dfrac{s:\ \Rightarrow A^{\alpha} \qquad \boxed{t:\ A^{x} \Rightarrow}}{\langle\!\langle\lambda x^{A}.t\rangle\!\rangle\,(\mu\alpha^{A}.s):\ \Rightarrow}\,\mathrm{q}$$

The sequent in the box should be duplicated/erased. Now we adapt this idea to tq-protocol on **LKT**. We split β redex rule into two as follows:

$$\dfrac{}{(\lambda x^{A}.t)\ (\mu\alpha^{A}.s) \to t\,[x := \mu\alpha^{A}.s]}\,\beta^{t} \qquad\qquad \dfrac{}{\langle\!\langle\lambda h^{A}.t\rangle\!\rangle\,(\mu\alpha^{A}.s) \to s\,[\alpha := \lambda h^{A}.t]}\,\beta^{q}$$

where $\to$ denotes one step β-reduction. In the β^{q} rule above, by using the jargon in [2], the cut-formula A^{h} is *attractive* and *main* in logical rule. Hence the attracting subderivation represented by t is the transporting one. When this transportation passing an instance of contraction/weakening, it should be duplicated/erased. This process corresponds to structural step in tq-protocol. We will come back to this point later.

The $\lambda\mu_{\mathfrak{t}}$-term of the form $(\lambda x^{A}.t)\ (\mu\alpha^{A}.s)$ is called β^{t}**-redex**. Similarly, $\langle\!\langle\lambda x^{A}.t\rangle\!\rangle\,(\mu\alpha^{A}.s)$ is called β^{q}-redex. t is **normal** iff no subterm of t is neither β^{t}-redex nor β^{q}-redex. The result of β-reduction on β-redex is called β**-contractum**.

2.4 Substitution

$t\,[x^{A} := s]$ means the standard substitution as meta-operation. In addition to this, we use μ-name substitution. The result of μ-name substitution: $u\,[\alpha^{A} :=$

Table 2. $\lambda\mu_{\mathrm{t}}$-Term Assignment for **LKT**

$$\dfrac{}{[\alpha]h:\ A^h\ ;\ \Rightarrow A^\alpha}\,\text{Ax} \qquad \dfrac{t:\ A^h\ ;\ \Rightarrow}{\langle\!\langle\lambda h.t\rangle\!\rangle x:\ \ ;\ A^x \Rightarrow}\,\text{D}$$

$$\dfrac{s:\ \ ;\ \Rightarrow A^\alpha \qquad t:\ A^h\ ;\ \Rightarrow}{\langle\!\langle\lambda h.t\rangle\!\rangle\,(\mu\alpha.s):\ \ ;\ \Rightarrow}\,\text{h-cut} \qquad \dfrac{s:\ \ ;\ \Rightarrow A^\alpha \qquad t:\ \ ;\ A^x \Rightarrow}{(\lambda x.t)\,(\mu\alpha.s):\ \ ;\ \Rightarrow}\,\text{m-cut}$$

$$\dfrac{u:\ B^h\ ;\ \Rightarrow \qquad s:\ \ ;\ \Rightarrow A^\alpha}{h'\,\langle\mu\alpha.s, \lambda h.u\rangle:\ (A \to B)^{h'}\ ;\ \Rightarrow}\,\text{L}\!\to \qquad \dfrac{t:\ \ ;\ A^x \Rightarrow B^\beta}{[\gamma](x,\beta).t:\ \ ;\ \Rightarrow (A \to B)^\gamma}\,\text{R}\!\to$$

$\lambda h.s]$ is obtained from u by recursively replacing every named subterm of u of the shape $[\alpha]v$ by $s\,[h := v\,[\alpha := \lambda h.s]]$. Formally it is defined as follows:

$$\begin{aligned}([\alpha]v)\,[\alpha := \lambda h.s] &\quad=\quad s\,[h := v\,[\alpha := \lambda h.s]]\\ ([\beta]v)\,[\alpha := \lambda h.s] &\quad=\quad [\beta](v\,[\alpha := \lambda h.s]), \quad \text{if} \quad \alpha \neq \beta\end{aligned}$$

We only show the base case. This definition says that named-term substitution is made of two substitution. First named-term is interpreted as a t-application. That is, the result of replacing α by $\lambda h.s$ is $(\lambda h.s)(v\,[\alpha := \lambda h.s])$. Then this newly created redex is immediately reduced. Hence we get $s\,[h := v\,[\alpha := \lambda h.s]]$. Notice that h occurs exactly once in s. Since second substitution $[h := v\,[\alpha := \lambda h.s]]$ dose not duplicate/erase the $v\,[\alpha := \lambda h.s]$.

2.5 Definitions for $\boldsymbol{\lambda\mu_{\mathrm{t}}}$-Calculus

Our $\lambda\mu$-term assignment for **LKT** is displayed in Table 2.

Definition 1 ($\lambda\mu_{\mathrm{t}}$ as a reduction system). *The* **reduction relation** $\longrightarrow_{\lambda\mu_t}$ *of $\lambda\mu_{\mathrm{t}}$, viewed as a rewrite system, is defined to be the compatible closure of the notion of reduction defined by two redex rules, namely, β^t and β^q.*

The $\longrightarrow_{\lambda\mu_t}$ defines the full **LKT-reduction** on $\lambda\mu_{\mathrm{t}}$-terms. We write reflexive and transitive closure of $\longrightarrow_{\lambda\mu_t}$ as $\Longrightarrow_{\lambda\mu_t}$. Hereafter we refer to the reduction system as $\lambda\mu_{\mathrm{t}}$-calculus.

Term assignment judgment is an ordered pair of $\lambda\mu_{\mathrm{t}}$-term and indexed sequent. We write a judgment $(s, \Pi\ ;\ \Gamma \Rightarrow \Delta)$ as $s:\ \Pi\ ;\ \Gamma \Rightarrow \Delta$. **Derivation rules** define the term assignment judgment. **Derivation** is a tree of derivation rules of which leaves are axioms, of which nodes are derivation rules other than axiom. We use π for the derivation. We call the term s as an assigned term to the derivation π, and refer to it by the notion of TermOf(π). Two derivations are said to be **equal** if they differ only up to indices of the formulas. Notice that this equivalence is identical with the *weak equivalence* of DJS([2]) thanks to the indexed formula/implicit structural rules in our formulation. We say that the derivation is **cut-free** if it contains neither m-cut nor h-cut. Let π be the derivation of which last inference rule is m-cut. We call the TermOf(π), which is a β-redex, as **m-cut redex**. **h-cut redex** is defined as the same way.

2.6 Some Properties of $\lambda\mu_{\mathrm{t}}$-Calculus

Our term assignment faithfully reflects the structure of sequent calculus. Thus, inductive definition of substitution on terms, agrees with the induction on the length of derivation. We can state this formally as follows:

Proposition 1 (subterm property). *In every derivation rule, all terms of premises are included as subterms in the term of the derivation rule.*

Proof. Mechanical checking of term assignment for each derivation rules.

Proposition 2 (cut-freeness and normality). *The derivation π is cut-free iff TermOf(π) is normal.*

Proof. ($\Rightarrow$) By induction on the length of derivation. E.g. for L$\rightarrow$: By induction hypothesis s and t are normal, hence $\langle\!\langle \lambda y.t \rangle\!\rangle (x\ s)$ is normal. ($\Leftarrow$) Obvious, as term of of h-cut is of the form of β^t-redex, and term of m-cut is β^q-redex.

Now we are ready to prove that our term assignment and bi-directional β-redex simulates tq-protocol.

Theorem 1 (tq-protocol simulated by normalization). *If an* **LKT** *derivation π reduces to π' by one tq-protocol step, then TermOf(π) $\rightarrow$ TermOf(π') by reducing the m-cut(h-cut) redex associated to the m-cut(h-cut).*

Proof (sketch). Naturally, the $\lambda\mu_{\mathrm{t}}$-terms are assigned to be compatible with tq-protocol. The rule Ax says that we identify the head-variable with the head-index. The rule D says that we identify the set of λ-variables with the set of λ-indices. It also says that we identify μ-names and μ-indices. Namely, in the term assignment judgment, each λ-variable (resp. μ-name) that occurs free in $\lambda\mu_{\mathrm{t}}$-term is identified with the λ-index (resp. μ-index) of the same name.

Structural Step (or S-step) of tq-protocol: S-step consists of two phases, namely S1-step and S2-step.

Elimination of h-cut corresponds to S2-step. In case of h-cut, *attractive* cut-formula is *main* in a logical rule (i.e., introduced by L$\rightarrow$). Hence we *transport* attracting subderivation: t. Transporting t exactly correspond to the μ-name substitution: $s\,[\alpha := \lambda h.t]$.(The definition of μ-name substitution will be displayed later in this section.) In the process of μ-name substitution, $\lambda h.t$ is duplicated (erased, resp.) whenever passing an instance of (implicit) contraction (weakening, resp.). That is, elimination of h-cut corresponds to S2-step in tq-protocol.

Similarly, elimination of m-cut corresponds to S1-step. That is, S1-step is simulated by λ-variable substitution. When the λ-variable substitution reaches to an instance of D rule, it turns into h-cut. That is, S1-step will change into S2-step at the point of the occurrence of D rule.

Logical Step (or L-step) of tq-protocol: L-step decompose the implication ($\to$). One may think of R-term as a kind of abstraction, and L-term as a kind of application. Moreover, an L-step can be considered as a communication. L-term: $h\langle s,u\rangle$ *sends* the two subderivations s and u to R-term. R-term: $(x,\beta).\,t$ receives the two subderivations by x and β. Under these intuitions, we display how L-term and R-term works in simulating of **L-step** of tq-protocol:

$$\dfrac{\dfrac{u\colon\ B^h\,;\ \Rightarrow \qquad s\colon\ ;\ \Rightarrow A^\alpha}{h'\langle\mu\alpha.s,\lambda h.u\rangle\colon\ (A\to B)^{h'}\ ;\ \Rightarrow}\,\mathrm{L}{\to} \qquad \dfrac{t\colon\ ;\ A^x \Rightarrow B^\beta}{[\gamma](x,\beta).\,t\colon\ ;\ \Rightarrow (A\to B)^\gamma}\,\mathrm{R}{\to}}{\langle\!\langle \lambda h'.h'\langle\mu\alpha.s,\lambda h.u\rangle\rangle\!\rangle\,(\mu\gamma.[\gamma](x,\beta).\,t)\colon\ ;\ \Rightarrow}\,\text{h-cut}$$

reduces to:

$$\dfrac{\dfrac{s\colon\ ;\ \Rightarrow A^\alpha \qquad t\colon\ ;\ A^x \Rightarrow B^\beta}{(\lambda x.t)\ (\mu\alpha.s)\colon\ ;\ \Rightarrow B^\beta}\,\text{m-cut} \qquad u\colon\ B^h\,;\ \Rightarrow}{\langle\!\langle\lambda h.u\rangle\!\rangle\,(\mu\beta.(\lambda x.t)\ (\mu\alpha.s))\colon\ ;\ \Rightarrow}\,\text{h-cut}$$

Obviously, $\langle\!\langle\lambda h'.h'\langle\mu\alpha.s,\lambda h.u\rangle\rangle\!\rangle\,(\mu\gamma.[\gamma](x,\beta).\,t)$ reduces to $((x,\beta).\,t)\,\langle\mu\alpha.s,\lambda h.u\rangle$ by μ-name substitution. However this is not equal to the series of application of m-cut and h-cut. Hence we need syntactic trick to mimic this. We define the syntactical equality as follows:

$$((x,\beta).\,t)\,\langle\mu\alpha.s,\lambda h.u\rangle \overset{\text{def}}{=} \langle\!\langle\lambda h.u\rangle\!\rangle\,(\mu\beta.(\lambda x.t)\ (\mu\alpha.s))$$

$((x,\beta).\,t)\,\langle\mu\alpha.s,\lambda h.u\rangle$ can be regarded as representing simultaneous application of m-cut and h-cut. In fact, two cuts are commutative; we can change the order of m-cut and h-cut for free. This is the essence of CR property of tq-protocol. Let us say that this syntactical equality is needed to fill the gap of term-calculus and sequent calculus.

At last we mention to **immediate** reduction of axiom cuts which is required by tq-protocol. It is easy to see that our substitution semantics agrees with immediate reduction of axiom cuts. To be more precise,

$$\begin{aligned}(\lambda x.t)\ (\mu\alpha.[\alpha]h) &= \lambda x.t\\ (\lambda x.\langle\!\langle\alpha'\rangle\!\rangle x)\ (\mu\alpha.s) &= \mu\alpha.s\ [\alpha := \alpha']\\ \langle\!\langle\lambda h.t\rangle\!\rangle\,(\mu\alpha.[\alpha]h) &= \lambda h.t\\ \langle\!\langle\lambda h.[\alpha']h\rangle\!\rangle\,(\mu\alpha.s) &= \mu\alpha'.s\ [\alpha := \alpha']\end{aligned}$$

These μ-name substitution stands for the syntactical replacement of α by α'.

3 CBN CPS-Transform of $\lambda\mu_{\mathrm{n}}$-Calculus

In this section, we define the transform from $\lambda\mu_{\mathrm{n}}$ into $\lambda\mu_{\mathrm{t}}$. Our goal is to prove that the SN and CR property of $\lambda\mu_{\mathrm{n}}$ is a consequence of that of $\lambda\mu_{\mathrm{t}}$.

Table 3. $\lambda\mu_{\mathrm{n}}$-Term Assignment for **CND**

$$\frac{}{x:\ A^x \Rightarrow A}\,\mathrm{Ax} \qquad \frac{L:\ \Gamma \Rightarrow B, A^\alpha, \Delta}{\mu^*\alpha.[\beta]L:\ \Gamma \Rightarrow A, B^\beta, \Delta}\,\mathrm{rename}$$

$$\frac{L:\ \Gamma, A^x \Rightarrow B, \Delta}{\lambda x.L:\ \Gamma \Rightarrow A \to B, \Delta} \to\mathcal{I} \qquad \frac{M:\ \Gamma_0 \Rightarrow A \to B, \Delta_0 \qquad N:\ \Gamma_1 \Rightarrow A, \Delta_1}{(M\,N):\ \Gamma_0, \Gamma_1 \Rightarrow B, \Delta_0, \Delta_1} \to\mathcal{E}$$

3.1 Parigot's $\boldsymbol{\lambda\mu_{\mathrm{n}}}$-Calculus

To begin with, we adopt $\lambda\mu_{\mathrm{n}}$-calculus of Parigot from [9]. It is defined on a natural deduction-style logic, called Classical Natural Deduction (**CND**). Sequents of **CND** is similar to that of **LK**. The only difference is, it has exactly one unindexed formula in rhs. The term assignment for **CND** is displayed in Table 3. We refer to the term as $\lambda\mu_{\mathrm{n}}$**-terms**, ranged over L, M, N, etc. There are three redex rules, namely β,ζ and $muetaCBN$:

$$\frac{}{(\lambda x.M)\ N \to M\ [x := M]}\ \beta$$

$$\frac{}{(\mu^*\gamma.[\delta]M)\ N \to \mu^*\beta.[\delta]M\ [\gamma := \lambda x.[\beta](x\ N)]}\ \zeta$$

$$\frac{}{\mu^*\delta.[\alpha'](\mu^*\alpha.[\beta]L) \to \mu^*\delta.([\beta]L)\ [\alpha := \alpha']}\ \mu\text{-}\eta$$

In ζ redex rule, we use one more new notion called μ-name substitution for $\lambda\mu_{\mathrm{n}}$-calculus.

Definition 2 (μ-name substitution for $\lambda\mu_{\mathrm{n}}$). *The μ-name substitution is of the form $M\,[\gamma := \lambda x.[\beta](x\,N)]$. The result of it is obtained from M by recursively replacing every named subterm of M of the shape $[\gamma]L$ by $[\beta](L\,N)$.*

That is, if we interpret named term $[\gamma]L$ as t-application, i.e., as $(\gamma\ L)$, μ-name substitution could be understood as a standard variable substitution. Note that ζ is essentially identical with Parigot's **structural reduction**.

Definition 3 ($\lambda\mu_{\mathrm{n}}$ as a reduction system). *The* **reduction relation** $\longrightarrow_{\lambda\mu_n}$ *of $\lambda\mu_n$, viewed as a rewrite system, is defined to be the compatible closure of the notion of reduction defined by three redex rules, namely, β and ζ and μ-η.*

The $\longrightarrow_{\lambda\mu_n}$ defines the **full** $\lambda\mu$**-reduction** relation, written $\longrightarrow_{\lambda\mu_n}$; its reflexive and transitive closure is $\Longrightarrow_{\lambda\mu_n}$. We say M is **intuitionistic** if it is of the form of x,$\lambda x.M$ or $(M\ N)$.

3.2 The “Naive” CPS-Transform

The naive transform a $\lambda\mu_{\mathrm{t}}$-term $\underline{M}$ of a $\lambda\mu_{\mathrm{n}}$-term M is based on the naive, inductive transform of each **CND** derivation rule into a sequence of **LKT** derivation.

$$\begin{aligned}
\underline{x} &= \mu\alpha.\langle\!\langle\alpha\rangle\!\rangle x \\
\underline{\lambda x.L} &= \mu\gamma.[\gamma](x,\beta).\,(\langle\!\langle\beta\rangle\!\rangle\,\underline{L}) \\
\underline{(M\;N)} &= \mu\beta.\langle\!\langle\lambda h.h\langle\underline{N},\beta\rangle\rangle\!\rangle\,\underline{M} \\
\underline{\mu^*\alpha.[\beta]L} &= \mu\alpha.\langle\!\langle\beta\rangle\!\rangle\,\underline{L}
\end{aligned}$$

In the above, we use abbreviation $\langle\!\langle\beta\rangle\!\rangle t$ for $\langle\!\langle\lambda y.[\beta]y\rangle\!\rangle t$. We show how $\to\mathcal{E}$ is transformed into an **LKT** derivation:

$$\cfrac{u :\ ;\ \Rightarrow (A\to B)^{\gamma} \qquad \cfrac{s :\ ;\ \Rightarrow A^{\alpha} \qquad [\beta]h :\ B^{h}\,;\ \Rightarrow B^{\beta}}{h\langle\mu\alpha.s,\lambda h.[\beta]h\rangle :\ (A\to B)^{h}\,;\ \Rightarrow B^{\beta}}\ \mathrm{L}\!\to}{\langle\!\langle\lambda h.h\langle\mu\alpha.s,\lambda h.[\beta]h\rangle\rangle\!\rangle\,(\mu\gamma.u) :\ ;\ \Rightarrow B^{\beta}}\ \text{h-cut}$$

3.3 The “Modified” CPS-Transform

Then, we proceed to the “modified” transform. The intention of this transform is, to remove *disturbing* m-cut during transform. Specifically, we transform the *k-successive* application of the form $(\ldots(MN_1)\ldots N_k)$ into k-successive L$\to$ in **LKT**. In fact, as we show later, this transform maps normal $\lambda\mu_{\mathrm{n}}$-terms to normal $\lambda\mu_{\mathrm{t}}$-terms. These m-cut are part of what Plotkin calls **administrative redexes**[10]. Similar observation is also reported by Herbelin[5]. His solution was to introduce special term, called **argument list**.

Let r be the **continuations** which is generated from the grammar: $r\ := \alpha \mid \lambda h.h\langle\mu\alpha.s,r\rangle$. Now we introduce modified transform $\underline{\underline{M}}$ of M as follows:

$$\begin{aligned}
\underline{\underline{L}} &= \mu\beta.(L : \beta) \\
\underline{\underline{\mu^*\alpha.[\beta]L}} &= \mu\alpha.(L : \beta)
\end{aligned}$$

In the first line, L is intuitionistic and β is a fresh μ-name. Then the infix operator “colon (:)” : $\lambda\mu_{\mathrm{n}}$-term $\times$ $\lambda\mu_{\mathrm{t}}$-term $\to$ $\lambda\mu_{\mathrm{t}}$-term is defined as follows:

$$\begin{aligned}
x : r &= \langle\!\langle r\rangle\!\rangle x \\
(\lambda x.L) : \gamma &= [\gamma](x,\beta).\,(L : \beta) \\
(\lambda x.L) : \lambda h.h\langle\underline{\underline{N}},r\rangle &= ((x,\beta).\,(L : \beta))\langle\underline{\underline{N}},r\rangle \\
(M\;N) : r &= M : \lambda h.h\langle\underline{\underline{N}},r\rangle \\
(\mu^*\alpha.[\beta]L) : r &= \langle\!\langle r\rangle\!\rangle(\mu\alpha.(L : \beta))
\end{aligned}$$

In the second and third line, we assume L is intuitionistic and β is a fresh name. Otherwise, the it is defined as follows:

$$\begin{aligned} \lambda x.(\mu^{*}\beta.[\delta]L) : \gamma &= [\gamma](x,\beta).\,(L : \delta) \\ \lambda x.(\mu^{*}\beta.[\delta]L) : \lambda h.h\langle \underline{\underline{N}}, r\rangle &= ((x,\beta).\,(L : \delta))\,\langle \underline{\underline{N}}, r\rangle \end{aligned}$$

This transform is very similar to Plotkin's "colon" translation. The difference is, we ask the "argument part" N of $(M\ N)$ to be transformed by this modified transform again. This is because we consider *full* reduction, instead of CBN reduction strategy. In the following, we considerably omit the proof for the lack of space.

Lemma 1. *if M is normal, then $\underline{\underline{M}}$ is normal.*

Proof. By induction on the structure of M.

Lemma 2. *If γ does not occur in M, then $(M : r)\,[\gamma := r'] = M : (r\,[\gamma := r'])$.*

Proof. By induction on the structure of M. For example,

$$\begin{aligned} (\lambda x.L : \gamma)\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, r\rangle] &= ([\gamma](x,\beta).\,(L : \beta))\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, r\rangle] \\ &= (x,\beta).\,(L : \beta)\,\langle \underline{\underline{N}}, r\rangle \\ &= \lambda x.L : \lambda h.h\langle \underline{\underline{N}}, r\rangle \end{aligned}$$

Lemma 3. $\langle\!\langle r\rangle\!\rangle \underline{\underline{N}} \longrightarrow_{\lambda\mu_t} \underline{\underline{N}} : r$

Proof. By cases whether N is intuitionistic or not.

Lemma 4. $\underline{L} \Longrightarrow_{\lambda\mu_t} \underline{\underline{L}}$

Proof. By induction on the structure of L. For example, suppose L is of the form $(M\ N)$, where $M = (M_1\ M_2)$.

$$\begin{array}{rcll} \underline{(M\ N)} & = & \mu\beta.\langle\!\langle \lambda h.h\langle \underline{\underline{N}}, \beta\rangle\rangle\!\rangle \underline{M} & \\ & \Longrightarrow_{\lambda\mu_t} & \mu\beta.\langle\!\langle \lambda h.h\langle \underline{\underline{N}}, \beta\rangle\rangle\!\rangle \underline{\underline{M}} & \text{i.h.} \\ & \longrightarrow_{\lambda\mu_t} & \mu\beta.\langle\!\langle \lambda h.h\langle \underline{\underline{N}}, \beta\rangle\rangle\!\rangle\,(\mu\gamma.(M : \gamma)) & \\ & \longrightarrow_{\lambda\mu_t} & \mu\beta.(M : \lambda h.h\langle \underline{\underline{N}}, \beta\rangle) & \\ & = & \mu\beta.((M\ N) : \beta) & \\ & = & \underline{\underline{(M\ N)}} & \end{array}$$

3.4 $\lambda\mu_\mathrm{t}$ Simulates $\lambda\mu_\mathrm{n}$

In this subsection, we establish the simulation of the normalization for **CND** by tq-protocol for **LKT**.

Lemma 5. $\underline{\underline{M}}\,[x := \underline{\underline{N}}] \Longrightarrow_{\lambda\mu_t} \underline{\underline{M\,[x := N]}}$

Proof. By induction on the structure of M. For the base case,

$$\begin{array}{rcl} \underline{\underline{x}}\,[x := \underline{\underline{N}}] & = & (\mu\alpha.\langle\!\langle\alpha\rangle\!\rangle\, x)\,[x := \underline{\underline{N}}] \\ & = & \mu\alpha.\langle\!\langle\alpha\rangle\!\rangle\,\underline{\underline{N}} \\ & \longrightarrow_{\lambda\mu_t} & \underline{\underline{N}} \\ & = & \underline{\underline{x\,[x := N]}} \end{array}$$

Lemma 6. $(M : r)\,[x := \underline{\underline{N}}] \Longrightarrow_{\lambda\mu_t} (M\,[x := N]) : r$

Proof. By induction on the structure of M. For the base case,

$$(x : r)\,[x := \underline{\underline{N}}] \quad = \quad \langle\!\langle r\rangle\!\rangle\,\underline{\underline{N}} \quad \longrightarrow_{\lambda\mu_t} \quad \underline{\underline{N}} : r$$

Lemma 7. $\underline{\underline{((\lambda x.L)\;N)}} \Longrightarrow_{\lambda\mu_t} \underline{\underline{L\,[x := N]}}$

Proof. Suppose M is intuitionistic. Another case is similar.

$$\begin{array}{rcl} \underline{\underline{((\lambda x.L)\;N)}} & = & \mu\beta'.((x,\beta).\,(L : \beta))\,\langle \underline{\underline{N}}, \beta'\rangle \\ & \Longrightarrow_{\lambda\mu_t} & \mu\beta'.(L : \beta)\,[x := \underline{\underline{N}}]\,[\beta := \beta'] \\ & \Longrightarrow_{\lambda\mu_t} & \mu\beta'.(L\,[x := N]) : \beta' \\ & = & \underline{\underline{L\,[x := N]}} \end{array}$$

Lemma 8.

$$\begin{array}{rcl} \underline{\underline{M}}\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] & \Longrightarrow_{\lambda\mu_t} & \underline{\underline{M\,[\gamma := \lambda x.[\beta](x\;N)]}} \\ (M : r)\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] & \Longrightarrow_{\lambda\mu_t} & (M\,[r := \lambda x.[\beta](x\;N)]) : (r\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle]) \end{array}$$

Proof. By induction on the structure of M. Suppose $M = (M_1\;M_2)$.

$$\begin{array}{rl} \underline{\underline{M}}\,[\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] & = \mu\delta.(M : \gamma)[\ldots] \\ & = \mu\delta.(M\,[\gamma := \lambda x.[\beta](x\;N)]) : \lambda h.h\langle \underline{\underline{N}}, \beta\rangle \\ & = \mu\delta.((M[\ldots]\;N) : \beta) \\ & = \underline{\underline{\mu^*\delta.[\beta](M[\ldots]\;N)}} \\ & = \underline{\underline{M\,[\gamma := \lambda x.[\beta](x\;N)]}} \end{array}$$

From second to third line, we use secondary induction as follows:

$$\begin{aligned}((M_1\ M_2):r)\ [\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] &= (M_1 : \lambda h.h\langle \underline{\underline{M_2}}, r\rangle)\ [\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] \\ &= (M_1[\ldots]) : (\lambda h.h\langle \underline{\underline{M_2}}, r\rangle)\ [\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] \\ &= M_1[\ldots] : (\lambda h.h\langle \underline{\underline{M_2\ [\gamma := \lambda x.[\beta](x\ N)]}}, r[\ldots]\rangle) \\ &= ((M_1\ M_2)\ [\gamma := \lambda x.[\beta](x\ N)]) : r[\ldots]\end{aligned}$$

Lemma 9. $\underline{\underline{((\mu^*\gamma.[\delta]M)\ N)}} \Longrightarrow_{\lambda\mu_t} \underline{\underline{\mu^*\beta.[\delta]M\ [\gamma := \lambda x.[\beta](x\ N)]}}$

Proof. By induction on the structure of M. The only interesting case is as follows:

$$\begin{aligned}\underline{\underline{((\mu^*\gamma.[\delta]M)\ N)}} &= \mu\beta.(((\mu^*\gamma.[\delta]M)\ N) : \beta) \\ &= \mu\beta.((\mu^*\gamma.[\delta]M) : \lambda h.h\langle \underline{\underline{N}}, \beta\rangle) \\ &= \mu\beta.\langle\!\langle \lambda h.h\langle \underline{\underline{N}}, \beta\rangle\rangle\!\rangle (\mu\gamma.M : \delta) \\ &\longrightarrow_{\lambda\mu_t} \mu\beta.(M : \delta)\ [\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] \\ &= \underline{\underline{\mu^*\beta.[\delta]M}}\ [\gamma := \lambda h.h\langle \underline{\underline{N}}, \beta\rangle] \\ &\Longrightarrow_{\lambda\mu_t} \underline{\underline{\mu^*\beta.[\delta]M\ [\gamma := \lambda x.[\beta](x\ N)]}}\end{aligned}$$

Lemma 10. $\underline{\underline{\mu^*\alpha.[\beta](\mu^*\beta'.[\delta]M)}} \longrightarrow_{\lambda\mu_t} \underline{\underline{(\mu^*\alpha.[\delta]M)\ [\beta' := \beta]}}$

Proof. easy.

Theorem 2 (Simulation). *if $M \longrightarrow_{\lambda\mu_n} N$, then $\underline{\underline{M}} \Longrightarrow_{\lambda\mu_t} \underline{\underline{N}}$.*

Proof. Lemma 7,Lemma 9 and Lemma 10 shows that β,ζ and μ-η redex rules are correctly simulated by $\Longrightarrow_{\lambda\mu_t}$.

Corollary 1. *SN and CR property of $\Longrightarrow_{\lambda\mu_n}$ is a corollary of that of $\Longrightarrow_{\lambda\mu_t}$.*

3.5 Relation to De Groote's Work

First, we briefly quote our result described in [7, 8]. We have revealed that CBN CPS-calculi are exactly the term calculi on the (intuitionistic decoration of) **LKT**; CBN CPS-calculi simulates tq-protocol for **LKT**.

Theorem 3 (Simulation). *$\lambda\mu_{\mathfrak{t}}$-calculus can be simulated by λ-calculus(i.e., Plotkin-style CBN* CPS*-calculus).*

The transform of $\lambda\mu_{\mathfrak{t}}$-term s into Plotkin-style CBN CPS-terms s^* are fairly simple. It can be described briefly as follows: $([\alpha]s)^* = (k\ s^*)$, $(\langle\!\langle\alpha\rangle\!\rangle s)^* = (s^*\ k)$, $(\mu\alpha.s)^* = \lambda k.s^*$ and $(\langle\!\langle t\rangle\!\rangle s)^* = (s^*\ t^*)$. A given mapping from μ-names into continuation variables such as $(A^\alpha)^* = (\neg A^t)^k$, $(B^\beta)^* = (\neg B^t)^{k'}$, $((A \to B)^\gamma)^* =$

$(\neg(A \to B)^t)^{k''}$ are assumed. For object variable, $(A^x)^* = (\neg\neg A^t)^x$ is also assumed. With this further simulation of $\lambda\mu_{\mathrm{t}}$, our "naive" CPS-transform recovers De Groote's CPS-transform of $\lambda\mu_{\mathrm{n}}$ as follows:

$$\begin{aligned}
\underline{x} &= \lambda k.x\, k \\
\underline{\lambda x.L} &= \lambda k''.k''\, (\lambda(x, k').\, (\underline{L}\, k')) \\
\underline{(M\, N)} &= \lambda k'.\underline{M}\, (\lambda h.h\langle \underline{N}, k'\rangle) \\
\underline{\mu^*\alpha.[\beta]L} &= \lambda k.\underline{L}\, k'
\end{aligned}$$

Note that this transform raise "double negation translation" on types. Hence we can see that Plotkin's CPS-transform can be divided into two phases. One is the simulation of **CND** by **LKT**, and the other is the simulation of **LKT** by its intuitionistic decoration in **LJ**. Recall that the intuitionistic decoration is the homomorphic transform. Thus we know that the simulation of **CND** by **LKT** is the essential part of the CPS-transform. This fact connects the theory of constructive classical logic and CPS-transform.

4 Conclusions and Further Directions

Naturally, the CBV system needs further study. One can easily adapt the term assignment method (i.e., first part) to the **LKQ** which is the dual of the **LKT**. From its duality, $\lambda\mu_{\mathrm{q}}$-calculus seems to be the candidate to simulate CBV version of $\lambda\mu$-calculus.

Our transform allows $\lambda\mu_{\mathrm{n}}$-calculus to be interpreted in **LKT**, hence its semantics. It should be more understood how these syntactical characterization of CBN relate to their (maybe more intrinsic) characterizations using domains, categories and more recently games. We should not be satisfied with the embedding into the coherent space.

It is known that $\mathcal{A} \to \mathcal{B}$, the set of stable morphisms in stable domain theory, is the same with $(!\,\mathcal{A}) \multimap \mathcal{B}$, the set of linear morphisms that linearize the source space. The discovery of this splitting of $\to$ into $!$ and $\multimap$ was a very important step towards the Girard's discovery of linear logic. Our investigation seems to go the other way round. It strongly implies the existence of "classical" CBN stable domain theory. That is, $(!?\,\mathcal{A}) \multimap (?\,\mathcal{B})$, represents CBN stable function.

References

[1] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. Sequent calculi for second order logic. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 211–224. Cambridge University Press, 1995.

[2] Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: linear logic. *Journal of Symbolic Logic*, 62(3), September 1997.

[3] Phillipe de Groote. A cps-translation of the $\lambda\mu$-calculus. In *Proc. CAAP'94*, pages 85–99. Springer-Verlag LNCS 787, April 1994.

[4] G. Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935.

[5] Hugo Herbelin. A λ-calculus structure ismorphic to gentzen-style sequent calculus structure. In *Proc. CSL '94*, Kazimierz, Poland, September 1994. Springer Verlag, LNCS 933.

[6] Martin Hofmann and Thomas Streicher. Continuation models are universal for $\lambda\mu$-calculus. In *Proc. LICS '97*, june 1997.

[7] Ichiro Ogata. Cut elimination for classical proofs as continuation passing style computation. In *Proc.Asian'98*, pages 61–78, Manila, Philippines, December 1998. Springer-Verlag LNCS 1538.

[8] Ichiro Ogata. Constructive classical logic as cps-calculus. To appear in International Journal of Foundations of computer science, 1999.

[9] Michel Parigot. Classical proofs as programs. In *3rd Kurt Gödel Colloquium*, pages 263–276. Springer-Verlag LNCS 713, 1993.

[10] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[11] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. Presented at MFPS '98, London, 1999.

[12] J. I. Zucker. Correspondence between cut-elimination and normalization, part i and ii. *Annals of Mathematical Logic*, 7:1–156, 1974.

Formal Verification of the MCS List-Based Queuing Lock

Kazuhiro Ogata and Kokichi Futatsugi

JAIST
{ogata, kokichi}@jaist.ac.jp

Abstract. We have formally verified the MCS list-based queuing lock algorithm (MCS) with CafeOBJ and UNITY. What we have shown is that it has the two properties that more than one process can never enter their critical section simultaneously and a process wanting to enter a critical section eventually enters there. First a simple queuing lock algorithm (MCS0) has been specified in CafeOBJ by adopting UNITY computational model, and verified with UNITY logic. Secondly a queuing lock algorithm (MCS1) specified in the same way as MCS0 has been verified by showing the existence of a simulation relation from MCS1 to MCS0 with the help of CafeOBJ. Lastly MCS has been derived from a slightly modified MCS1.

1 Introduction

We have formally verified the MCS list-based queuing lock algorithm (MCS) [6] with CafeOBJ [2] and UNITY [1]. MCS is a scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processes (or processors) attempting to acquire the lock. We have shown MCS has two properties: one is a safety property, and the other a liveness property. The safety property is that more than one process can never enter their critical section when MCS is used to implement the mutual exclusion problem. The liveness property is that a process wanting to a critical section eventually enters there when MCS is used to implement the mutual exclusion problem.

The verification is divided into three stages. First a simple queuing lock algorithm (MCS0) has been specified in CafeOBJ by adopting the parallel computational model of UNITY, and verified w.r.t. the two properties using UNITY logic with the help of CafeOBJ rewrite engine [7]. Secondly a queuing lock algorithm (MCS1) has been specified in the same way as MCS0. We have then shown there exists a simulation relation from MCS1 to MCS0 with the help of CafeOBJ rewrite engine, which indicates MCS1 has the safety properties that MCS0 has [4]. MCS1 has also been verified w.r.t. the liveness property using the simulation relation and UNITY logic. Lastly MCS has been derived from a slightly modified MCS1.

2 Preliminary

The parallel computational model of UNITY is basically a labeled transition system. It has some initial states and finitely many transition rules. The execution starts from one initial state and goes on forever; in each step some transition rule is nondeterministically chosen and executed. The nondeterministic choice is constrained by the following *fairness* rule: every transition rule is chosen infinitely often. Execution of one transition rule may simultaneously change (possibly nothing) multiple components of the system.

UNITY has a mini programming language to represent the parallel computational model. A program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. A multiple-assignment statement corresponds to a transition rule.

UNITY also provides a proof system based on the logic that is an extension of Floyd-Hoare logic to parallel programs, with a flavor of temporal logic [5]. The logic is based on assertions of the form $\{p\}\ s\ \{q\}$, denoting that execution of statement s in any state that satisfies predicate p results in a state that satisfies predicate q, if execution of s terminates. Properties of a UNITY program are expressed using assertions of the form $\{p\}\ s\ \{q\}$, where s is universally or existentially quantified over the statements of the program. The properties are classified into a *safety* or a *liveness* property. Existential quantification over program statements is essential in stating liveness properties, whereas safety properties can be stated using only universal quantifications over statements (and using the initial condition). Although all properties of a program can be expressed directly using assertions, a few additional terms are introduced for conveniently describing properties of programs: *unless*, *stable*, *invariant*, *ensures*, and *leads-to* (or $\mapsto$). The first four terms are defined as follows: $p\ \mathit{unless}\ q \equiv \langle \forall s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{p \vee q\}\rangle$, $\mathit{stable}\ p \equiv p\ \mathit{unless}\ \mathit{false}$, $\mathit{invariant}\ p \equiv (\text{initial condition} \Rightarrow p) \wedge \mathit{stable}\ p$, and $p\ \mathit{ensures}\ q \equiv (p\ \mathit{unless}\ q) \wedge \langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\}\rangle$, where F is a program, s a statement, and p and q predicates. A given program has the property $p \mapsto q$ if and only if this property can be derived by a finite number of applications of the following inference rules: $\frac{p\ \mathit{ensures}\ q}{p\ \mapsto\ q}$, $\frac{p\ \mapsto\ q,\ q\ \mapsto\ r}{p\ \mapsto\ r}$, and $\frac{\langle \forall m : m \in W :: p(m)\ \mapsto\ q\rangle}{\langle \exists m : m \in W :: p(m)\rangle\ \mapsto q}$ for any set W.

CafeOBJ provides notational machinery to describe labeled transition systems, and the corresponding semantics, namely, *hidden algebra* [3]. In hidden algebra, a *hidden sort* represents (states of) a labeled transition system. *Action operations*, which take the state of a labeled transition system and zero or more data represented by *visible sorts*, and returns another (possibly the same) state of the system, can change the state of a labeled transition system. The state of a labeled transition system can be observed only with *observation operations* that take the state of a labeled transition system and returns the value of a data component in the system.

3 The MCS List-Based Queuing Lock

The MCS list-based queuing lock (MCS) is a scalable algorithm for spin locks that has the following properties:

– it guarantees FIFO ordering of lock acquisitions;
– it spins on locally-accessible flag variables only;
– it requires a small constant amount of space per lock; and
– it works equally well (requiring only $O(1)$ network transactions per lock acquisition) on machines with and without coherent caches.

The code in a traditional style is given below:

```
type qnode = record
    next : ^qnode
    locked : Boolean
type lock = ^qnode

procedure acquireLock( L : ^lock, I : ^qnode )
    I->next := nil
    pred : ^qnode := fetch&store( L, I )
    if pred != nil
        I->locked := true
        pred->next := I
        repeat while I->locked

procedure releaseLock( L : ^lock, I : ^qnode )
    if I->next = nil
        if comp&swap( L, I, nil )
            return
        repeat while I->next = nil
    I->next->locked := false
```

MCS uses two atomic operations *fetch&store* and *comp&swap*. *fetch&store* takes as arguments the address of a memory location and a value, and indivisibly stores the value into the memory location and returns the old value stored in the memory location. *comp&swap* takes as arguments the address of a memory location and two values, and indivisibly stores the second value into the memory location only if the first value is equal to the old value stored in the memory location and returns *true* if so and *false* otherwise.

4 Verification of MCS

We formally verify MCS w.r.t. the following two points:

ME1 more than one process can never enter their critical section simultaneously when MCS is used to implement the mutual exclusion problem; and
ME2 a process wanting to enter a critical section eventually enters there when MCS is used to implement the mutual exclusion problem.

4.1 Simple Queuing Lock

The simple queuing lock MCS0 uses two atomic operations with large granularity. The operations are *atomicPut* and *atomicGet*. *atomicPut* takes as input a queue and a queue item, and indivisibly puts the item into the queue at the end, and *atomicGet* takes as input a queue, and indivisibly deletes the top item from the queue. The code for MCS0 in a traditional style is given below:

```
type queue = record
    head : ^qnode
    tail : ^qnode

procedure acquireLock( Q : ^queue, I : ^qnode )
    I->next := nil
    atomicPut( Q, I )
    repeat while I != Q->head

procedure releaseLock( Q : ^queue, I : ^qnode )
    atomicGet( Q )
```

Processes can mutually exclusively enter a critical section with *acquireLock* and *releaseLock*. The code in a traditional style is as follows:

```
acquireLock( &queue, &item[i] )
Critical Section
releaseLock( &queue, &item[i] )
```

queue is a global variable, and *item*[*i*] is a local variable to each process *i*. Initially *queue* is empty, that is, *queue*−>*tail* is *nil*. We will later refer this code as ME-CODE. Suppose that a process that has entered the critical section eventually exits from there.

Specification We formally specify ME-CODE in CafeOBJ by adopting UNITY computational model. First ME-CODE is divided into six atomic actions: *acquireLock*, *setNext*, *atomicPut*, *checkHead*, *releaseLock*, and *atomicGet*. For example, *setNext* corresponds to (&*item*[*i*])−>*next* := *nil*, and *atomicPut* to *atomicPut*(&*queue*, &*item*[*i*]). Each atomic action corresponds to a UNITY assignment-statement. Next let each process have six states: *rem0*, *try0-1*, *try0-2*, *try0-3*, *crit0*, and *exit0-1*. For example, that a process is in *rem0* means that it is executing any code segment but ME-CODE, and a process changes its state to *try0-1* whenever it executes *acquireLock* only if its state is *rem0*. The state transition diagram is shown in Fig. 1 (a).

We specify each atomic action with an action operator of CafeOBJ's behavioral specification. In the specification, there are also four observation operators for each variable or state in ME-CODE. They are *p*, *head*, *tail*, and *next*. We can use *p* to observe the state of each process. Positive integers are used to identify processes, and also their local variables *item*'s. Let 0 mean *nil*. The main part of the signature of the specification is as follows:

```
-- Initial system state
 op init0 : -> SState0
-- Observation operators
 bop p    : NzNat SState0 -> PState0
 bops head tail : SState0 -> Nat
 bop  next : NzNat SState0 -> Nat
-- Action operators
 bops acquireLock setNext atomicPut   : NzNat SState0 -> SState0
 bops checkHead releaseLock atomicGet : NzNat SState0 -> SState0
```

SState0 is a hidden sort that represents the state of ME-CODE, and *Nat*, *NzNat*, and *PState0* are visible sorts that represent natural numbers, positive integers, and the states of processes. *init0* is any initial state of ME-CODE.

In this paper, we give only two sets of equations for the two states after a process has executed *atomicPut* and *atomicGet*, respectively. We will refer to the specification for ME-CODE as MCS0-SPEC.

```
-- 3. after execution of ‘atomicPut’
ceq p(I,atomicPut(I,S)) = try0-3 if p(I,S) == try0-2 .
ceq p(J,atomicPut(I,S)) = p(J,S) if I =/= J or p(I,S) =/= try0-2 .
ceq head(atomicPut(I,S)) = I       if p(I,S) == try0-2 and tail(S) == 0 .
ceq head(atomicPut(I,S)) = head(S) if p(I,S) =/= try0-2 or tail(S) > 0 .
ceq tail(atomicPut(I,S)) = I       if p(I,S) == try0-2 .
ceq tail(atomicPut(I,S)) = tail(S) if p(I,S) =/= try0-2 .
ceq next(J,atomicPut(I,S)) = I if p(I,S) == try0-2 and tail(S) > 0 and J == tail(S) .
ceq next(J,atomicPut(I,S)) = next(J,S)
                             if p(I,S) =/= try0-2 or tail(S) == 0 or J =/= tail(S) .

-- 6. after execution of ‘atomicGet’
ceq p(I,atomicGet(I,S)) = rem0   if p(I,S) == exit0-1 .
ceq p(J,atomicGet(I,S)) = p(J,S) if I =/= J or p(I,S) =/= exit0-1 .
ceq head(atomicGet(I,S)) = next(head(S),S)
                           if p(I,S) == exit0-1 and next(head(S),S) > 0 .
ceq head(atomicGet(I,S)) = head(S) if p(I,S) =/= exit0-1 or next(head(S),S) == 0 .
ceq tail(atomicGet(I,S)) = 0       if p(I,S) == exit0-1 and next(head(S),S) == 0 .
ceq tail(atomicGet(I,S)) = tail(S) if p(I,S) =/= exit0-1 or next(head(S),S) > 0 .
eq  next(J,atomicGet(I,S)) = next(J,S) .
```

Verification We formally verify MCS0-SPEC w.r.t. **ME1** and **ME2** with UNITY logic and CafeOBJ. First **ME1** and **ME2** are restated more formally. Let p_i be the state of a process i.

1. $\textit{invariant}\ p_i = \textit{crit0} \wedge p_j = \textit{crit0} \Rightarrow i = j$; and
2. $p_i = \textit{try0-1} \mapsto p_i = \textit{crit0}$.

Proof sketch 1: We actually prove the more powerful property that is given below:

$$\textit{invariant}\ p_i \in \{\textit{crit0}, \textit{exit0-1}\} \wedge p_j \in \{\textit{crit0}, \textit{exit0-1}\} \Rightarrow i = j\ ,$$

where $x \in \{v_1, \ldots, v_n\}$ means $x = v_1, \ldots,$ or $x = v_n$. Suppose the property ‘$\textit{invariant}\ p_i \in \{\textit{crit0}, \textit{exit0-1}\} \Rightarrow \textit{head} = i$’ holds, the desired property can be derived. Thus we prove the assumed property. In the initial state, the predicate is vacuously true. Thus it is sufficient to show that a process i changes its state to *crit0* whenever it executes *checkHead* only if its state is *try0-3* and $\textit{head} = i$, and that *head* does not change unless the process i in *crit0* or *exit0-1* changes its state to *rem0*. In this paper, only the proof score for the latter case is shown. In the proof, suppose another property ‘$\textit{invariant}\ p_i \in \{\textit{crit0}, \textit{exit0-1}\} \Rightarrow \textit{tail} \neq \textit{nil}$’ holds, which can be shown in the same way. The proof score is given below:

```
open MCS0-SPEC
  op s : -> SState0 .  ops i j1 j2 j3 j4 : -> NzNat .
  eq p(i,s) = crit0 .  eq p(j1,s) = rem0 .  eq p(j2,s) = try0-1 .
  eq p(j3,s) = try0-2 .  eq p(j4,s) = try0-3 .
  eq head(s) = i .   eq tail(s) > 0 = true .  eq j3 > 0 = true .
  red head(acquireLock(j1,s)) == i and tail(acquireLock(j1,s)) > 0 .
  red head(setNext(j2,s)) == i and tail(setNext(j2,s)) > 0 .
  red head(atomicPut(j3,s)) == i and tail(atomicPut(j3,s)) > 0 .
  red head(checkHead(j4,s)) == i and tail(checkHead(j4,s)) > 0 .
  red head(releaseLock(i,s)) == i and tail(releaseLock(i,s)) > 0 .
close
```

We can show the latter case that *head* does not change unless the process i in *crit0* or *exit0-1* changes its state to *rem0* by having CafeOBJ rewrite engine execute the proof score. □

Proof sketch 2: It is easy to show the property '$p_i = \mathit{try0\text{-}1} \mapsto p_i = \mathit{try0\text{-}3}$' holds. Thus it is sufficient to show the property '$p_i = \mathit{try0\text{-}3} \mapsto p_i = \mathit{crit0}$' holds. From these two properties, the desired one can be obtained.

First we define a partial function d that takes as arguments two queue items, and returns the distance between them in the queue if both of them are in a queue and the first item is not in the rear of the second one in the queue. It is defined as follows:

$$d(i,j) = \begin{cases} 0 & \textbf{if } i = j \\ 1 + d(i\text{-}{>}\mathit{next}, k) & \textbf{otherwise} . \end{cases}$$

To prove the property, it is sufficient to show the following one:

$$p_i = \mathit{try0\text{-}3} \wedge d(\mathit{head}, i) = k \mapsto (p_i = \mathit{try0\text{-}3} \wedge d(\mathit{head}, i) < k) \vee p_i = \mathit{crit0} .$$

By applying the induction principle for *leads-to* to this one, the desired property '$p_i = \mathit{try0\text{-}3} \mapsto p_i = \mathit{crit0}$' can be derived. This property is then proven.

If $d(\mathit{head}, i) = 0$, it is easy to show '$p_i = \mathit{try0\text{-}3} \mapsto p_i = \mathit{crit0}$' since $\mathit{head} = i$. So, suppose $d(\mathit{head}, i) > 0$. If so, there must be exact one process j such that $\mathit{head} = j$. The process j must be in *try0-3*, *crit0*, or *exit0-1*. All we have to do is to show that only the process j whose state is *exit0-1* decrements $d(\mathit{head}, i)$, but any other process does not change $d(\mathit{head}, i)$. The two cases that a process k in *try0-2* executes *atomicPut*, and the process j in *exit0-1* executes *atomicGet* are more interesting than others. Only the two cases are handled here. In the former case it is shown that neither *head* nor the *next* field of a non-tail item i in the queue changes, and in the latter case it is shown that *head* changes to the second item in the queue, but the *next* field of a non-tail item i does not change. In both cases, the process i does not change its state. By having CafeOBJ rewrite engine execute the following proof score, they are shown.

```
open MCS0-SPEC
  op s : -> SState0 .  ops i j k : -> NzNat .
  eq p(i,s) = try0-3 .  eq p(j,s) = exit0-1 .  eq p(k,s) = try0-2 .
  eq head(s) = j .  eq tail(s) > 0 = true .  eq next(j,s) > 0 = true .
  red next(i,atomicPut(k,s)) == next(i,s) and next(i,atomicGet(j,s)) == next(i,s) .
  red head(atomicPut(k,s)) == head(s) and head(atomicGet(j,s)) == next(head(s),s) .
  red p(i,atomicPut(k,s)) == try0-3 and p(i,atomicGet(j,s)) == try0-3 .
close
```

□

4.2 Queuing Lock

We prove the queuing lock MCS1 has the two properties **ME1** and **ME2** by showing there exists a simulation relation from MCS1 to MCS0. MCS1 uses the same atomic operations as MCS. The code for MCS1 in a traditional style is given as follows:

```
procedure acquireLock( Q : ^queue, I : ^qnode )
    I->next := nil
    pred : ^qnode := fetch&store( &Q->tail, I )
    if pred = nil
        Q->head := I
    else
        pred->next := I
    repeat while I != Q->head

procedure releaseLock( Q : ^queue, I : ^qnode )
    if Q->head->next = nil
        if comp&swap( &Q->tail, Q->head, nil )
            return
        repeat while Q->head->next = nil
    Q->head := Q->head->next
```

Specification We formally specify ME-CODE with MCS1 instead of MCS0 in the same way as ME-CODE with MCS0. The specification is referred as MCS1-SPEC. Each process in MCS1-SPEC has 10 states: *rem1*, *try1-1*, *try1-2*, *try1-3*, *try1-4*, *crit1*, *exit1-1*, *exit1-2*, *exit1-3*, and *exit1-4*. There are 10 atomic actions: *acquireLock*, *setNext*, *fetch&store*, *checkPred*, *checkHead*, *releaseLock*, *checkHNext1*, *comp&swap*, *checkHNext2*, and *setHead*. The state transition diagram is shown in Fig. 1 (b). The main part of the signature of MCS1-SPEC is as follows:

```
-- Initial system state
op init1 : -> SState1
-- Observation operators
bop  p    : NzNat SState1 -> PState1
bops head tail : SState1 -> Nat
bops next pred : NzNat SState1 -> Nat
-- Action operators
bops acquireLock setNext fetch&store checkPred checkHead   : NzNat SState1 -> SState1
bops releaseLock checkHNext1 comp&swap checkHNext2 setHead : NzNat SState1 -> SState1
```

In this paper, we give only two sets of equations for the two states after a process has executed *fetch&store* and *comp&swap*, respectively.

```
-- 3. after execution of ‘fetch&store’
ceq p(I,fetch&store(I,S)) = try1-3 if p(I,S) == try1-2 .
ceq p(J,fetch&store(I,S)) = p(J,S) if I =/= J or p(I,S) =/= try1-2 .
eq  head(fetch&store(I,S)) = head(S) .
ceq tail(fetch&store(I,S)) = I       if p(I,S) == try1-2 .
ceq tail(fetch&store(I,S)) = tail(S) if p(I,S) =/= try1-2 .
eq  next(J,fetch&store(I,S)) = next(J,S) .
ceq pred(I,fetch&store(I,S)) = tail(S)   if p(I,S) == try1-2 .
ceq pred(J,fetch&store(I,S)) = pred(I,S) if I =/= J or p(I,S) =/= try1-2 .

-- 8. after execution of ‘comp&swap’
ceq p(I,comp&swap(I,S)) = rem1    if p(I,S) == exit1-2 and tail(S) == head(S) .
ceq p(I,comp&swap(I,S)) = exit1-3 if p(I,S) == exit1-2 and tail(S) =/= head(S) .
ceq p(J,comp&swap(I,S)) = p(J,S)  if I =/= J or p(I,S) =/= exit1-2 .
eq  head(comp&swap(I,S)) = head(S) .
```

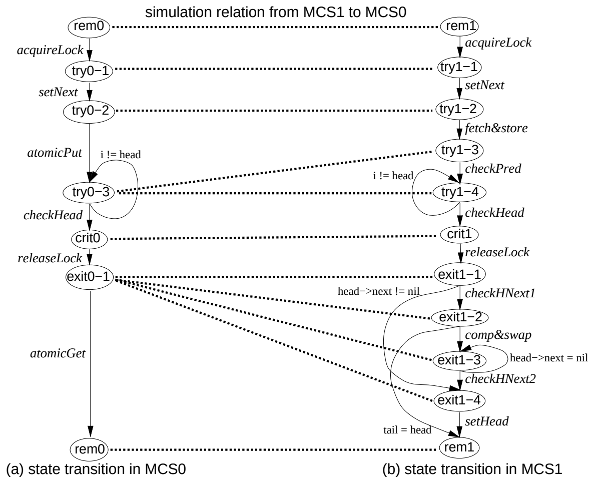


Fig. 1. Correspondence between states in MCS0 and MCS1

```
ceq tail(comp&swap(I,S)) = 0       if p(I,S) == exit1-2 and tail(S) == head(S) .
ceq tail(comp&swap(I,S)) = tail(S) if p(I,S) =/= exit1-2 or tail(S) =/= head(S) .
eq  next(J,comp&swap(I,S)) = next(J,S) .
eq  pred(J,comp&swap(I,S)) = pred(J,S) .
```

Verification We formally verify MCS1-SPEC w.r.t. **ME1** and **ME2**. First we prove MCS1-SPEC has **ME1** by showing there exists a simulation relation from MCS1-SPEC to MCS0-SPEC with the help of CafeOBJ rewrite engine. Next we prove MCS1-SPEC has **ME2** with UNITY logic and the simulation relation.

Proof sketch 1: We make a mapping from each state in MCS1-SPEC to some state in MCS0-SPEC. Given a state of MCS1-SPEC, that is, each process's state, *next* and *pred*, and global *head* and *tail*, we have some corresponding state of MCS0-SPEC, that is, each process's state and *next*, and global *head* and *tail*.

Given a state of MCS1-SPEC, each process's state in MCS0-SPEC corresponding to the MCS1-SPEC state is defined as shown in Fig. 1. The correspondence is given in terms of equations. Only two equations are shown here:

```
ceq p(I, sim(S)) = try0-3  if p(I, S) == try1-3 or p(I, S) == try1-4 .
ceq p(I, sim(S)) = crit0   if p(I, S) == crit1 .
```

Even if a process i in MCS1-SPEC executes some action, the others do no change their states. In the corresponding situation in MCS0-SPEC, all processes but i do not change their states either. Thus we also have the equations such as '`ceq p(J, sim(acquireLock(I, S))) = p(J, sim(S)) if I =/= J .`'.

Given a state of MCS1-SPEC, the corresponding *head* in MCS0-SPEC is i if there exists a process i in MCS1-SPEC such that its state is *try1-3* and its *pred* is *nil*, and otherwise it is the same as that in MCS1-SPEC. It is sufficient to have the following two equations for mapping each state in MCS1-SPEC to *head* in MCS0-SPEC:

```
ceq head(sim(S)) = I       if p(I, S) == try1-3 and pred(I, S) == 0 .
ceq head(sim(S)) = head(S) if p(I, S) =/= try1-3 or pred(I, S) > 0 .
```

However, since the equations have a variable that does not appear in the left-hand sides, the proof process becomes complicated if these equations are used. So, instead of the two equations, we have equations that indicate how *head* in MCS0-SPEC changes when a process executes an action in MCS1-SPEC. Some of the equations are given here:

```
eq  head(sim(setNext(I, S))) = head(sim(S)) .
ceq head(sim(fetch&store(I, S))) = I            if p(I, S) == try1-2 and tail(S) == 0 .
ceq head(sim(fetch&store(I, S))) = head(sim(S)) if p(I, S) =/= try1-2 or tail(S) > 0 .
```

The first equation, and the second and third equations indicate how *head* in MCS0-SPEC changes if a process in MCS1-SPEC executes *setNext* and *fetch&store*, respectively. The second equation specifies *head* in MCS0-SPEC is set to i whenever a process i in MCS1-SPEC executes *fetch&store* only if its state is *try1-2* and *tail* is *nil*, that is, the queue is empty. In this case, the corresponding action in MCS0-SPEC is *atomicPut* as shown in Fig. 1.

It is straightforward to map a state in MCS1-SPEC to *tail* in MCS0-SPEC. Given a state of MCS1-SPEC, the corresponding *tail* in MCS0-SPEC is the same as that in MCS1-SPEC.

Given a state of MCS1-SPEC, the corresponding *next* of a process j in MCS0-SPEC is i if there exists a process i in MCS1-SPEC such that its state is *try1-3* and its *pred* is equal to j, and otherwise it is the same as that in MCS1-SPEC. It is sufficient to have the following two equations for mapping each state in MCS1-SPEC to each *next* in MCS0-SPEC:

```
ceq next(J,sim(S)) = I if p(I,S) == try1-3 and pred(I,S) > 0 and J == pred(I,S) .
ceq next(J,sim(S)) = next(J,S)
                      if p(I,S) =/= try1-3 or pred(I,S) == 0 or J =/= pred(I,S) .
```

As is the case for *head*, however, the proof process becomes complicated if these equations are used. So, instead of the two equations, we have equations that indicate how each *next* in MCS0-SPEC changes when a process executes an action in MCS1-SPEC. Some of the equations are given here:

```
ceq next(I,sim(setNext(I,S))) = 0              if p(I,S) == try1-1 .
ceq next(J,sim(setNext(I,S))) = next(J,sim(S)) if I =/= J or p(I,S) =/= try1-1 .
ceq next(J,sim(fetch&store(I,S))) = I
    if p(I,S) == try1-2 and tail(S) > 0 and J == tail(S) .
ceq next(J,sim(fetch&store(I,S))) = next(J,sim(S))
    if p(I,S) =/= try1-2 or tail(S) == 0 or J =/= tail(S) .
```

The first and second equations, and the third and fourth equations indicate how each *next* in MCS0-SPEC changes if a process in MCS1-SPEC executes *setNext* and *fetch&store*, respectively. The third equation specifies the process j's *next* in MCS0-SPEC is set to i whenever a process i in MCS1-SPEC executes *fetch&store* only if its state is *try1-2*, *tail* is not *nil*, and *tail* is equal to j.

Now that we have defined the mapping *sim* from each state in MCS1-SPEC to some state in MCS0-SPEC, we give a candidate R for a simulation relation from MCS1-SPEC to MCS0-SPEC. Here the signature and equation for the candidate is shown:

```
op _R[_]_ : SState0 NzNat SState1 -> Bool
eq S0 R[I] S1 = p(I,S0) == p(I,sim(S1)) and head(S0) == head(sim(S1)) and
                tail(S0) == tail(sim(S1)) and next(I,S0) == next(I,sim(S1)) .
```

Then we prove the candidate is a simulation relation from MCS1-SPEC to MCS0-SPEC with CafeOBJ. It is easy to show *init0* and *init1* are under the candidate, i.e. *init0* $R[i]$ *init1* for any process i. Thus we show the candidate is preserved if any process executes any action in MCS1-SPEC. First we define two states, one for MCS0 and the other for MCS1, that are under the candidate as follows:

```
mod SIMREL1-PROOF1 {
  pr(SIMREL1)
  ops i j : -> NzNat  op s0 : -> SState0  op s1 : -> SState1
  eq p(i,s0) = p(i,sim(s1)) .  eq p(j,s0) = p(j,sim(s1)) .
  eq head(s0) = head(sim(s1)) .  eq tail(s0) = tail(sim(s1)) .
  eq next(i,s0) = next(i,sim(s1)) .  eq next(j,s0) = next(j,sim(s1)) .
}
```

SIMREL1 is the module in which the candidate is defined. We give some proof scores that show the candidate is preserved if a process i executes some action.

```
-- 3. after execution of ‘fetch&store’
-- 3.1 in the case that ‘tail(s1) = 0’
open SIMREL1-1-PROOF1
  eq p(i,s1) = try1-2 .  eq tail(s1) = 0 .
  red atomicPut(i,s0) R[i] fetch&store(i,s1) .
  red atomicPut(i,s0) R[j] fetch&store(i,s1) .
close
-- 3.2 in the case that ‘tail(s1) > 0’
open SIMREL1-1-PROOF1
  eq p(i,s1) = try1-2 .  eq tail(s1) > 0 = true .
  red atomicPut(i,s0) R[i] fetch&store(i,s1) .
  red atomicPut(i,s0) R[j] fetch&store(i,s1) .
close
```

The above two proof scores show the relation R between $s0$ and $s1$ is preserved after a process i in *try1-2* executes *fetch&store* in $s1$ and the corresponding process i executes *atomicPut* in $s0$. The former score is the case where $tail = nil$, and the latter the case where $tail \neq nil$. In both scores, j denotes any process but i. Even if any process whose state is not *try1-2* executes *fetch&store* in $s1$, nothing changes. Thus in that case the relation is vacuously preserved.

```
-- 8. after execution of ‘comp&swap’
-- 8.1 in the case that ‘head(s1) = tail(s1)’
open SIMREL1-1-PROOF1
  eq p(i,s1) = exit1-2 .  op k : -> NzNat .  eq tail(s1) = k .
  eq head(s1) = tail(s1) .  eq head(sim(s1)) = tail(sim(s1)) .
  eq next(k,s1) = 0 .  eq next(k,sim(s1)) = 0 .  eq next(k,s0) = next(k,sim(s1)) .
  red atomicGet(i,s0) R[i] comp&swap(i,s1) .
  red atomicGet(i,s0) R[j] comp&swap(i,s1) .
close
-- 8.2 in the case that ‘head(s1) =/= tail(s1)’
open SIMREL1-1-PROOF1
  eq p(i,s1) = exit1-2 .
```

```
    red s0 R[i] comp&swap(i,s1) .
    red s0 R[j] comp&swap(i,s1) .
  close
```

The above two proof scores show the relation R between $s0$ and $s1$ is preserved after a process i in *exit1-2* executes *comp&swap* in $s1$ and the corresponding process i in $s0$ executes *atomicGet* or does nothing depending on whether $head = tail$ in $s1$ (and also $s0$).

Since we have proven the candidate is a simulation relation from MCS1-SPEC to MCS0-SPEC as described above, we can say MCS1-SPEC also has the safety properties such as **ME1** that MCS0-SPEC has. □

Proof sketch 2: It is sufficient to show '$p_i = try1\text{-}4 \mapsto p_i = crit1$' in MCS1-SPEC in order that MCS1-SPEC has the liveness property **ME2**. For this sake, all we have to do is to prove the following property in MCS1-SPEC:

$$p_i = try1\text{-}4 \wedge d(head, i) = k \mapsto (p_i = try1\text{-}4 \wedge d(head, i) < k) \vee p_i = crit1 \ .$$

Since we have shown there is a simulation relation from MCS1-SPEC to MCS0-SPEC, mapped *crit1* to *crit0* and each of *exit0-1* through *exit1-4* to *exit0-1*, and proven there is at most one process in *crit0* and *exit0-1* in MCS0-SPEC, there must be at most one process in *crit1*, *exit1-1*, *exit1-2*, *exit1-3* and *exit1-4*.

We show a process in one of *crit1* through *exit1-4* eventually reaches *exit1-4* if there is at least one process in *try1-4*. We here handle only the case that a process is in *exit1-3*. A process in *crit1* or *exit1-1* has to go through *exit1-2* in order to reach *exit1-3*. A process in *exit1-1* (or *exit1-2*) changes its state to *exit1-2* (or *exit1-3*) whenever it executes *checkHNext1* (or *comp&swap*) only if *head−>next* = *nil*, i.e. *head* = *tail* (or *head* ≠ *tail*, that is, another process k has executed *fetch&store* before the process executes *comp&swap*). In the latter case, the k's *pred* is equal to *head*, and *head−>next* eventually becomes k that is non-*nil*. Thus a process in *exit1-3* eventually reaches *exit1-4*.

Since there is at most one process j in *crit1* through *exit1-4*, *head* exactly changes to the second element in the queue whenever the process j reaches *rem1* if there are at least two elements in the queue, that is, at least one process is in *try1-4*, before the process j reaches *rem1*. Moreover since there is a simulation relation from MCS1-SPEC to MCS0-SPEC, we can say the *next* fields of non-tail elements in the queue corresponding to processes in *try1-4* do not change at least until the processes reach *rem1*. Consequently the desired property can be obtained. □

4.3 MCS Queuing Lock

First *qnode*'s *locked* field is used to slightly modify MCS1. The modified version is called MCS2 whose code in a traditional style is as follows:

```
  procedure acquireLock( Q : ^queue, I : ^qnode )
      I->next := nil
      pred : ^qnode := fetch&store( &Q->tail, I )
      if pred = nil
```

```
            I->locked := false
            Q->head := I
        else
            I->locked := true
            pred->next := I
        repeat while I->locked

    procedure releaseLock( Q : ^queue, I : ^qnode )
        if Q->head->next = nil
            if comp&swap( &Q->tail, Q->head, nil )
                return
            repeat while Q->head->next = nil
        Q->head, Q->head->next->locked := Q->head->next, false
```

In the above code, x, $y := v_1$, v_2 means x and y are synchronously set to v_1 and v_2. The specification for ME-CODE with MCS2 in CafeOBJ is called MCS2-SPEC. We can show there exists a simulation relation from MCS2-SPEC to MCS1-SPEC, and then verify MCS2-SPEC w.r.t. **ME1** in the same way as the verification of MCS1-SPEC. Besides MCS2-SPEC can be also verified w.r.t. **ME2** using the simulation relation and UNITY logic in the same way.

In *acquireLock* of MCS2, the **repeat while** statement can be lifted up to the **else** clause of the **if** statement because the **repeat while** statement is meaningless if *pred* = *nil*, that is, if so, *I–>locked* is set to *false*. In *releaseLock* of MCS2, the last assignment can be reduced to *Q–>head–>next–>locked := false*. In addition the second argument *I* must be equal to *Q–>head* because of the proof of the safety property of MCS1 and the existence of a simulation relation from MCS2-SPEC to MCS1-SPEC. Hence all *Q–>head*'s can be changed to the second argument *I*. Moreover the two assignments can be deleted from the **if** clause in *acquireLock* of MCS2. Consequently we can derive the MCS list-based queuing lock by finally modifying the first argument and &*Q–>tail* into *L* : ^*lock* and *L*, respectively.

5 Conclusion

We have described how MCS is verified w.r.t. **ME1** and **ME2** with CafeOBJ and UNITY. The verification is divided into three stages. First MCS0 specified in CafeOBJ by adopting UNITY computational model has been verified w.r.t. **ME1** and **ME2** using UNITY logic with the help of CafeOBJ. Secondly MCS1 specified in the same way as MCS0 has been verified w.r.t. **ME1** by showing the existence of a simulation relation from MCS1 to MCS0, and verified w.r.t. **ME2** using UNITY logic and the simulation relation. Lastly MCS has been derived from MCS2 that is a slightly modified MCS1.

References

1. Chandy, K. M. and Misra, J.: *Parallel Program Design: A Foundation.* Addison-Wesley. 1988
2. Diaconescu, R. and Futatsugi, K.: *CafeOBJ Report. AMAST Series in Computing* **6**. World Scientific. 1998

3. Goguen, J. and Malcolm, G.: A Hidden Agenda. Technical Report CS97-538. Univ. of California at San Diego. 1997
4. Lynch, N. A.: *Distributed Algorithm.* Morgan-Kaufmann. 1996
5. Manna, Z. and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag. 1991
6. Mellor-Crummery, J. M. and Scott, L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* **9** (1). (1991) 21–65
7. Nakagawa, A. T., Sawada, T. and Futatsugi, K: CafeOBJ User's Manual – ver.1.3 –. 1997. Available at http://caraway.jaist.ac.jp/cafeobj/

BDD-Nodes Can Be More Expressive

Frank Reffel

Institut für Logik, Komplexität und Deduktionssysteme; Universität Karlsruhe (TH);
Am Fasanengarten 5; D-76128 Karlsruhe; Germany;
reffel@ira.uka.de

Abstract. Normally a whole BDD is associated with its root node. We state that this is not necessarily the best choice. BDDs can be stored more memory efficient if a possibility is introduced that one node can represent several BDDs.
BDDs made it possible to apply symbolic model checking [15] to systems which could not be verified with methods based on an explicit state space representation. However, they cannot always avoid an exponential blow-up known as the state space explosion which is caused by the fact that the system is build through a composition of modules.
We divide the variable index placed in the BDD nodes into two parts. Only one of it is included in the new nodes called CBDD-nodes. This allows the use of one BDD label, i.e. one BDD node, for different variables. Moreover a branching possibility at the end of a BDD-part is introduced such that BDD-parts situated in different layers of the BDD can be shared.
A prototypical BDD-library was implemented that uses cyclic BDDs (CBDDs). It was used to verify examples out of the following domains: hardware verification, model checking and communication protocols. Some encouraging results have been achieved where it was possible to represent certain functions with a constant number of CBDD-nodes while classical BDDs grow linearly.

1 Introduction

BDDs are the state-of-the-art representation for boolean functions. They allow the application of symbolic model-checking to systems with 10^{20} states and beyond [6]. Most of the systems are build out of several distributed modules communicating with each other. As all sequences of the system have to be considered, the construction of the product automata typically leads to an exponential blow-up in the number of states.

Memory efficiency is a central problem of existing model-checkers. Although BDDs lead to a compact system representation, they can grow exponentially in the size of the system causing an overflow of the main memory. The transition relation and the set of reachable states are represented as BDDs. Model checking temporal logic properties can now be reduced to the calculation of fix-points for functions which represent sets of states. These functions are represented as BDDs and so the calculations can be done very efficiently treating many states in each iteration step.

Often a system contains several modules of the same structure. For example in hardware verification for slightly different purposes the same standard module is used or in protocol verification a common protocol is used by all communicating stations. In this case, the relation describing the transitions of one module has the same structure for a set of modules. As each one is represented with different variables they appear in different layers of the BDD and therefore cannot be shared using the reduction rules of classical BDDs.

The aim of our approach is to provide a possibility to share nodes in different layers. This variation of classical BDDs permits a representation with less nodes while the efficiency of the BDD-algorithms is not affected. As the approach only tries to reduce the memory consumption there is no direct gain of time. But if the verification of a system exhausts the whole main memory and the secondary memory has to be used, the additional reduction of space turns into a significant reduction of time if it allows to perform the computations within the given memory limitations. Therefore, it is not useful to apply this method to small systems which can easily be verified with low main memory requirements. On the other hand, with a growing size of a system it becomes more interesting to use space reduction techniques like the one presented here even when some auxiliary time is spent on the determination of the additional reduction. Moreover the experiments will show that the possibility of an additional reduction increases with the size of a system.

The paper is organized as follows. In the next section, a short survey about some existing BDD variations and related work is given. In section 3 some preliminaries about BDDs and model-checking are explained while in section 4 the new structure of CBDDs is introduced. Finally, some experimental results are presented in section 5.

2 Related Work

There were many attempts of BDD-variations trying to overcome the weaknesses of BDDs to represent for example a multiplication or the hidden-weighted-bit function which require an exponential number of BDD-nodes.

Free BDDs: Free BDDs [12] use different variable orderings on different paths. They are a generalization of BDDs. Instead of a fixed ordering, an *FBDD*-type describes the different orderings on different paths. The *FBDDs* are constructed according to a fixed type which is also a graph. The problem is that no efficient way is known to determine an optimal or at least a “good” type in general.

Zero-suppressed BDDs: Zero-suppressed BDDs [17] use a different reduction rule. A node is omitted if its 0-successor leads to *false* and not when its successors are identical. Depending on the function which is represented with either reduction rule a smaller representation is possible.

Indexed BDDs: Indexed BDDs [3] allow variables to be tested several times on a path. This structure allows a very compact representation of the hidden-weighted-bit function but uniqueness is lost. The algorithms for the manipulation of *IBDDs* and the test of equality become more complicated.

Differential BDDs (Δ-BDDs): In differential BDDs [1], the nodes do not include the index of the variables but the difference of indices to the upper variable. This allows a reduction of nodes in different layers as proposed in our approach. The difference is that two BDD-parts in different layers can only be shared if they are both at the bottom of the Δ-BDD. They are unique but a transformation of a BDD into a Δ-BDD can also lead to a blow-up in the number of nodes.

Function descriptors (for LIFs): Function descriptors (FDs) [13] are a BDD-like representation for linearly inductive functions (LIFs). It is possible to share parts in different layers and they can be manipulated very efficiently. The drawback of the method are the strong regularity constraints. All modules of a system must have exactly the same structure and there must be a way to arrange them hierarchically.

BDD trees: BDD trees were introduced in [16]. They are a generalization of BDDs where the linear ordering is replaced by a tree order. A decomposition of the system has to be found which in case of a hierarchical system can be determined easily. McMillan managed to match the same complexity as classical BDD-algorithms but the algorithms for the determination of logical functions have to be changed and become more complicated. BDDs appearing in different layers cannot be shared and the method is only suitable for hierarchical systems.

Symmetry and abstraction: Symmetry [9, 7] and abstraction [8] are very powerful methods to reduce the necessary resources for the verification of a system. The problem is that it has to be proven manually that the application of a certain abstraction or symmetry relation preserves the correctness of the verification. To exploit symmetry, the orbit relation and a representative for each orbit have to be determined, which can lead to very complex computations and demand an experienced user.

Other approaches try to prove properties for systems with many identical processes [4] or rings of processes [11]. These methods also require the manual proof of preconditions and make restrictions to both the system and the properties which can be shown.

Our approach provides a different way to reduce memory consumption. Taking advantage of the modular structure of a system only the potential structural similarities have to be indicated and there is no need to prove any preconditions. Depending on the structure of the BDD the existent structural identities are found automatically, if they exist, and a more compact representation of the BDD is achieved.

3 Preliminaries

3.1 BDDs

Ordered binary decision diagrams (OBDDs) introduced by Bryant [5] are a graphical representation for boolean functions. An OBDD $G(f, \pi)$ with respect to the function f and the variable ordering π is an acyclic graph with one source and two sinks labeled with *true* and *false*. All other (internal) nodes are labeled with a boolean variable x_i of f and have two outgoing edges *left* and *right*. For all edges from an x_i labeled node to an x_j labeled node, we have $\pi(i) < \pi(j)$, such that on every path in G the variables are tested in the same order and each variable is tested at most once.

Reduced OBDDs with respect to a fixed variable ordering are a canonical representation for boolean functions. An OBDD is reduced if isomorphic sub-BDDs exist only once and nodes whose outgoing edges lead to the same successor are omitted. The reduced OBDD is build directly, integrating the reduction rules into the construction to avoid the construction of the full graph.

The variable ordering π can be chosen freely but it has a great influence on the size of the OBDDs, e.g. there are functions which have OBDDs of linear size for a "good" and of exponential size for a "bad" ordering. To determine the optimal ordering is an NP-hard problem but there exist heuristics which deliver good orderings for most applications [2].

In the following, we will only speak of BDDs, however we always mean reduced OBDDs.

As already mentioned, the variable ordering has a great influence on the size of the BDDs. A popular heuristic is to place variables depending on each other close to each other in the ordering. So we define blocks of variables where each block corresponds to the variables of one module, because it is assumed that the dependence inside a module is stronger than to variables outside the module. The blocks are ordered in a style that communicating modules appear as close as possible in the ordering. The ordering of the variables inside one block can be chosen freely or determined with other heuristics, but in the following it will be important that the orderings are identical for similar modules.

Since we intend to share BDD-parts in different levels of the BDD we will first define what is meant exactly with a BDD-part:

Definition 1 (BDD-part).
A BDD-part *identified with the node x_i is the part of the whole BDD which starts at node x_i and includes all nodes reachable on a path from x_i of the same module.*

A maximal BDD-part *is a BDD-part which has a parent node which belongs to another module.*

We use the convention that the *false*-leaf belongs to a BDD-part if it is reachable from one of its nodes. In contrast, the *true*-leaf is an independent BDD-part and does not belong to any other BDD-parts. This is necessary for the following inductive definition of structural identity for BDD-parts. The motivation for the different treatment of the two leafs is that in most applications an edge to *false* from an upper level of the BDD is much more frequent than an edge leading directly to *true*. The definition of structural identical BDD-parts is necessary to describe the additional reduction provided by CBDDs.

Definition 2 (Structural identity). *Two BDD-parts p and q are* structural identical *with distance k (denoted by* $\mathrm{id}_k(p,q)$*), if and only if:*

- $p \equiv q \equiv false$ *or*
- $\mathrm{var}(p) = x_i$ *and* $\mathrm{var}(q) = x_{i+k}$ *and the successors satisfy the following conditions:*
  - $\mathrm{id}_k(1(p), 1(q))$ *or both do not belong to p, resp. q and*
  - $\mathrm{id}_k(0(p), 0(q))$ *or both do not belong to p, resp. q*

where $0(s)$ *and* $1(s)$ *designate the 0- and the 1-successor of node s, respectively and* $\mathrm{var}(s)$ *the variable index of s.*

3.2 Model Checking

In the following, we will suppose that the system consists of identical modules only, but in fact our approach can be used whenever there exist at least some modules which are represented by isomorphic sets of variables. The only requirement we make is that there exists a partitioning of the system into modules.

A state s of the whole system corresponds to the product of states of the single modules: $s = \{s_1, \ldots, s_n\}$. The transition relation $Trans(s,t)$ is constructed using the transition relations of the single modules $Trans_i(s,t_i)$. Such a relation changes the state of module i from s_i to t_i. As it can communicate with other modules this transition can depend on the states of all modules, hence the whole state s serves as input to $Trans_i$.

In the simplest case, all modules have the same structure. Thus, the same transition relation $Trans_i$ is used for all of them but with different input variables.

There are three types of finite state machines: The synchronous, the asynchronous and the interleaving model. They differ slightly in the structure of their transition relation but all models have in common that their transition relation depends on the relations of the single modules. The following expression describes exemplarily the structure of the transition relation for an interleaving model where $CoStab_i$ fixes the states of all modules except module i.

$$Trans(s,t) = \bigvee_{i=1,\ldots,n} Trans_i(s,t_i) \wedge CoStab_i(s,t) \tag{1}$$

For model checking of any kind of temporal properties the BDDs needed beside this transition relation describe sets of states, for example the set of reachable states.

4 Cyclic BDDs (CBDDs)

CBDDs try to be more memory efficient than classical BDDs through a better exploitation of structural similarities inside the BDD. Concentrating on systems with similar modules structural identical BDD-parts appearing in different layers of the BDD should be represented only once and used several times. For this purpose mainly two characteristics are necessary which are not offered by classical BDDs:

1. One node should have the potential to represent different variables, so the labels of the nodes must be different from variables.
2. A possibility has to be introduced which allows to branch to different nodes after the use of a common BDD-part.

4.1 Labels of CBDD-Nodes

The idea is to split a BDD into layers according to the modules. Suppose that the maximal number of variables in one module is m. In a classical BDD the nodes could be labeled with the values $i * m + k$ where i is the index of the module, and $k \in \{0, \ldots, m-1\}$ is the index of the variable inside the module.

In a CBDD-node the $i*m$ part is omitted and the nodes are only labeled with the index k. In a BDD the knowledge of the root node was sufficient to characterize a boolean function. Now in addition it is necessary to know the module this node belongs to. A boolean function can be derived from the combination of a node and its module index.

When an evaluation path through a CBDD is traced the module index has to be adjusted when the successor node belongs to a different module. A marking on such an edge would be sufficient indicating that the module index has to be increased by one. In case that two nodes belong to non-adjacent modules the index has to be increased by a number greater than one. Therefore we will assume that the marking is realized by an auxiliary special node placed on an edge between two nodes of different modules; it contains the information for the determination of the module index for the next node. The reason why we use a node and not a more expressive marking will be explained in Section 4.2 where these special nodes are used for an additional functionality.

Until now a CBDD is a BDD where blocks of nodes are placed side by side instead of one among another. The nodes are labeled with a restricted set of values and some edges include additional information to determine the module to which the next node of the CBDD belongs to.

As the restricted set of labels can be represented by less bits the size of a normal CBDD-node is smaller than a BDD-node. Such a restriction of the label representation can also be an approach to address more nodes without increasing the node-size since more bits remain for the encoding of the pointers to the successors.

4.2 Branching Points

As introduced in the previous section nodes which belong to different modules are separated by a special node. These nodes can be used as branching points. When a structural identical BDD-part appears in different levels of the CBDD it is not necessary to replicate it twice. It suffices to introduce an edge which leads to the root node of that BDD-part which is originally used in a different level. As this edge can correspond to a back-edge a cycle is introduced into the BDD explaining the name: Cyclic BDDs.

Following a path through a CBDD after each BDD-part a special node is encountered. These special nodes are not anymore only an interruption of an edge but several edges can start from them each labeled with a module index. When a BDD-part is used several times additional edges have to be appended to the special nodes at the end of this BDD-part. Depending on the current module index the corresponding successor node is determined by the choice of the correct outgoing edge of the special node. Furthermore the module index has to be adapted.

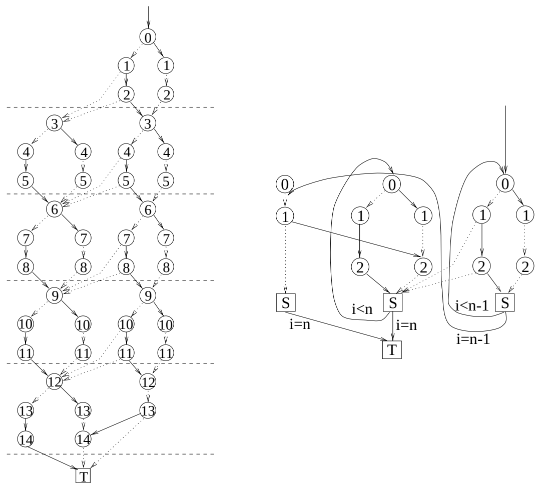


Fig. 1. BDD and CBDD for the same function. The *false* leaf is omitted.

In Figure 1 an example for such a CBDD and its corresponding BDD is shown. It represents the reachable states of Milners scheduler. The BDD grows linearly while the number of CBDD-nodes remains constant and only the degree of the outgoing edges of the special nodes increases.

The additional reduction when variables of different modules are represented by the same nodes is called second order reduction.

When CBDDs are applied to systems which contain no similar modules a certain overhead can be caused by the introduction of the special nodes which are not necessary when BDDs are used instead. Until a reduction is found special nodes are only needed to indicate that the module index has to be increased. This is implicitly clear when an edge leads from a k_0-labeled node to a k_1-labeled node if k_1 is smaller than k_0. A possible optimization is to omit those special nodes until a reduction is found and to introduce them only when they are needed as branching points. This could avoid most of the overhead in practice.

4.3 Definition of CBDDs

After the informal description of the data structure in the last two sections we will now give a more formal definition of CBDDs:

Definition 3 (CBDD). *A CBDD is a directed graph with three types of nodes:*

Leafs *are labeled with* true *and* false *like in BDDs.*
Normal nodes *are labeled with an index* $k \in \{0, \dots, m-1\}$. *They have exactly two outgoing edges which can lead either to the leaf* false*, to a normal node with an index greater than* k *or to a special node.*
Special nodes *have an arbitrary number of successors. Each outgoing edge is labeled with a module index. A successor node of a special node is either the leaf* true*, a normal node, or another special node. The next special node on each path which starts with an i-labeled edge must possess an (i+1)-labeled edge.*

A CBDD must always be reduced according to the usual two BDD-reduction rules.

The BDD reduction rules are applied during the construction of CBDDs just as for classical BDDs. So the number of normal nodes in a CBDD never exceeds the number of nodes for the corresponding BDD. On the other hand it is not necessary that all possible second order reductions are executed. A CBDD where all special nodes have exactly one outgoing edge is a valid CBDD. This offers the possibility to mix CBDDs where the second order reduction has already been applied with unreduced CBDDs. Note that CBDDs are always reduced with respect to the classical BDD reduction rules; they are so-called unreduced if there is no reduction between different levels.

4.4 Manipulation of CBDDs

The major change when CBDDs are used instead of BDDs is that a node without its corresponding module index is not sufficient to characterize a boolean function. So in practice there has to be made a strict distinction between a CBDD-node and a CBDD represented as a tuple (*node, module index*).

Fortunately the algorithms for boolean operators, restriction, quantification and the relational product which have been proven to be very efficient for BDDs are not affected and remain unchanged. The only functions which have to be adapted are the following basic BDD-manipulation operations:

Extraction of a variable from a CBDD-node: It is necessary to know the module index to calculate the correct variable index.

Successor determination: When the 0- or the 1-successor of a CBDD-node should be determined the case has to be respected where the direct successor is a special node. Special nodes can never be the result of a successor operation. The correct normal node (or leaf) has to be chosen with respect to the module index. In combination with the increased module index this tuple is the result of the operation.

Construction of new nodes: Nothing has to be changed if the new node and its successors belong to the same module. In case a successor node belongs to another module a special node has to be inserted.

However, the most important change is the detection and realization of the additional reduction. How the possibilities for the second order reduction are determined efficiently and when this reduction is performed will be explained in detail in the following section.

When the unique-table which contains all nodes is consulted to determine if a node already exists it has to be considered that the table can contain a node more than once. When the direct successor of a node is a special node it can have several normal nodes as "real" successors so it will be inserted into the unique-table with different hash values. This also becomes important when a garbage collection is applied. Nodes can only be deleted when they are unreachable from all modules.

4.5 Determination of the Second Order Reduction

In the last section the necessary modifications were discussed when CBDDs are used instead of BDDs. Those internal changes are only important inside a CBDD-library and do not affect the user. The only information which must be indicated by the user is the affiliation of the variables to the modules. This information could be determined by an intelligent parser and an appropriate input syntax but this has not yet been implemented. Moreover a direct manipulation of the module structure should remain possible for an experienced user to exhaust the potential of CBDDs.

A very important point is that these informations concerning the module structure do not affect the correctness of the verification. When two modules turn out to be different the probability to find structural identities between the BDD-parts is very low but the verification result will still be valid. So the user only influences the application possibilities of the second order reduction but it is not necessary that any preconditions have to be proven for the use of CBDDs.

To apply the second order reduction isomorphic BDD-parts, which correspond to Multi-Terminal-BDDs, have to be determined. To handle the additional reduction, a new hash-table is introduced. It contains the root nodes of the maximal BDD-parts. The hash value depends on the structure of the whole BDD-part. When two entries have the same hash value it is tested whether their structure is identical. Supposing an appropriate choice of the hash function the test if two BDD-parts with the same hash-value are structurally identical has to visit all nodes of the BDD-parts only when the test is successful. In the other case, a difference will in general appear after the comparison of very few nodes, which allows an efficient determination of the second order reduction.

Typical efficient implementations of BDD-libraries do not allow the application of the second order reduction during the construction efficiently, because the reduced nodes could not be removed. As there can be pointers from other nodes and from the computed table to the reduced nodes they had to be kept until a garbage collection is performed or a traversal of the two hash-tables would be necessary. Therefore, we delayed the search for the reduction until a garbage collection takes place and perform all possible second order reductions in a reduction phase. At that point, the hash-tables have to be traversed anyway and after the dead nodes have been marked it is checked whether structural identities allow a further reduction of the remaining nodes.

In [21] it was shown that it is most efficient to perform a garbage collection as late as possible because nodes can be reallocated allowing the reuse of already performed computations. So another approach to integrate the reduction phase into the verification process is to replace a garbage collection by a reduction phase. If enough nodes can be reduced the execution of a garbage collection can be retarded because the same information can be represented with less nodes and there is enough memory left to continue the verification.

4.6 Uniqueness

One of the most important properties of BDDs is the uniqueness of the representation already proven by Bryant [5]. Two boolean functions are equivalent if and only if their BDD-representation is equal. As this test can be performed in constant time it is frequently used by the classical BDD-algorithms and allows the reuse of former results contributing strongly to the efficiency of BDDs.

Unfortunately, CBDDs are not unique. Uniqueness is lost because of different possibilities for the second order reduction. This seems to influence the efficient use of CBDDs badly but this deficiency can be avoided. Similar as for BDDs, the equality-test for CBDDs still can be performed in constant time. In Section 4.4 it was explained that only the algorithms for the basic CBDD-manipulations have to be changed slightly while the boolean operations are not affected. Neither for an efficient CBDD-manipulation nor for an equivalence test in constant time the uniqueness of the underlying data-structure is a necessary condition.

Definition 4 (weak uniqueness). *A set* $S = \{\Phi(f) | f : \mathbb{B}^n \to \mathbb{B}\}$ *of representations* $\Phi(f)$ *for boolean functions* f *is called* weak unique *if equivalent boolean functions are represented by the same representative:*

$$\forall f_1, f_2 : \mathbb{B}^n \to \mathbb{B}, (\Phi(f_1), \Phi(f_2) \in S) \Rightarrow (f_1 \equiv f_2 \Leftrightarrow \Phi(f_1) = \Phi(f_2))$$

Lemma 1. *The set of CBDDs stored by a CBDD-library is weak unique.*

Proof idea: The unique-table guarantees that each newly created node is not contained in the set of stored CBDD-nodes. CBDDs are constructed in a bottom-up style like BDDs. Therefore an induction on CBDD-size is possible. The fact that the successor nodes exist only once is sufficient to guarantee in combination with the use of the unique-table that a set of CBDDs is weak unique. □

Note that the definition of weak uniqueness is different from uniqueness. When a unique representation is used its structure is fixed from the moment when the semantics of a concrete function is defined. If a weak unique structure is used during the construction process of the representation for a concrete function there can be several points where an indeterministic choice has to be made. The important condition to guarantee the weakened form of uniqueness is that once this choice has been made it can be guaranteed that arriving at that point again the same choice will be taken.

5 Experiments

We evaluated our approach on several examples. In [14], a scalable circuit for a division is given. For a dividend of n bits and a divisor of d bits it consists of $(n - d) * d$ cells of two different types. The verification can be divided into two steps. First, an intermediate result has to be determined which contains variables for each cell. These variables are eliminated with a quantification but as the intermediate result is usually bigger than the final BDD, it is important also to reduce this intermediate result. When the bit length of the divisor is fix, the number of CBDD-nodes increases up to a certain bit length of the dividend. Afterwards, the number of CBDD-nodes remains constant, only the number of successors of the special nodes increases linearly. In the corresponding BDD, the number of nodes grows linearly.

As a model checking example we used the tree-arbiter [10]. It is a distributed hardware solution for a mutual exclusion mechanism. There are $2n$ users who can request the use of one resource and the tree-arbiter eventually returns an acknowledge. It has to be assured that no two users are granted simultaneous access to the resource. The tree-arbiter is constructed out of $2n - 1$ cells of the same type which form a pyramid structure where each internal cell communicates with three other cells, so there exists no "good" variable ordering which would be necessary for a linear BDD representation. We experimented with tree-arbiters with 5 to 21 modules. The transition relation ranges from 4200 to 578000 BDD-nodes which could be reduced to 48% – 75% if CBDDs are used. During the iteration process which calculates the set of reachable states the necessary BDDs can be reduced to 75% on average. Note that the best percentage of reduction was obtained for large systems while for arbiters with few modules only a small amount of nodes could be reduced. A similar system is the *DME* (distributed mutual exclusion) where the modules are organized as a ring structure. There we observed the effect that for the transition relation, the number of BDD-nodes increases linearly while from a certain system size on the number of CBDD-nodes remains constant and only the number of successors of the special nodes increases.

An example for protocol verification is the PCI-bus protocol [20]. We experimented with a scenario with up to 6 slots and two modes of policy: round robin and fixed priority. Each of the slots can be empty or equipped with a "user" which eventually requests the bus. So the system is scalable depending on the number of non-empty slots. In addition, an arbitration module with a different structure is needed. We observe that an intermediate result is bigger than the transition relation itself (factor 2 to 8). The BDD representing the reachable states has only very few nodes in comparison to the transition relation. For a scenario where all slots are empty, a very good reduction can be obtained because the transition relations for all slots are equal. For two filled slots the least reduction is found, only 18% can be reduced for a fixed priority policy (12% for round robin). Then the percentage of reduction increases with the number of filled slots. For the round robin policy and 5 equipped slots the BDD of the intermediate result could be reduced from 1.300.000 nodes to 920.000 and for the fixed priority policy with 6 slots only 387.000 CBDD-nodes are necessary instead of 776.000 BDD-nodes corresponding to a reduction of more than 50%.

Big transition relations can also be coped using a partitioned transition relation. In this case only the relations for the single transitions have to be determined each represented by a different BDD because they depend on different variables. In this case CBDDs can also help to save memory because one CBDD can be used to represent several transition relations of single modules which are structurally identical for equal modules.

The experiments show that the best reduction is obtained for big systems which is very desirable because the application field for this method are systems which cannot be represented within the available main memory. Regarding the example of the PCI-Bus, the BDD for the representation of the reachable states is very small but before it can be constructed the BDD for the transition relation has to be determined. It is much bigger and must be held in main memory for the whole calculation of the reachable states. A reduction of this BDD which allows to perform the rest of the calculation without exceeding the space limitations can yield an important gain of time.

In all experiments the additional time necessary for the determination and realization of the second order reduction did never exceed 15% of the time needed for the whole verification.

6 Conclusion

We presented an alternative technique to represent boolean functions using a BDD-variation. It was shown that the characterization of a boolean function with its root node is not necessarily the best choice. The introduced enhancements of the classical BDD structure allow the economization of nodes while the efficiency of BDD-algorithms is not affected.

Depending on the application and its modules, structural identities appear in the BDD which can be exploited by the approach. This leads to a more space efficient system representation increasing the degree of reduction: nodes are "used" from different levels of the CBDD without the obligation to prove any properties by manual interference. The user only has to identify the modules which occur several times or probably have the same structure. The second order reduction is found automatically in case identical BDD-parts exist. As the method only tries to reduce the main memory requirements it causes an overhead in time which is necessary to determine the additional reduction. So its advantages are mainly important for the verification of big systems which cannot be verified within the main memory limitations. In this case the reduction of space quickly turns into a reduction of time. Thus, the use of the second level memory can be avoided which is usually a magnitude slower.

Sharing of BDD-parts in one layer would require the possibility to remove former reductions. As this seems not to be feasible efficiently the reduction is limited to the reuse of BDD-parts in different layers. Therefore the maximal reduction is linear in the number of modules. This cannot avoid an exponential blow-up but the memory consumption can be reduced.

It is intended to evaluate the efficiency of the approach on further examples and to examine the applicability of it to other BDD-variations. Also, the combination of this method with popular BDD-optimizations like dynamic variable reordering [18] will be investigated.

Acknowledgments The author is supported by DFG within the graduate program on controllability of complex systems.

References

[1] A. Anuchitanukul, Z. Manna, and T. E. Uribe. Differential BDDs. *Lecture Notes in Computer Science*, 1000:218–233, 1995.

[2] A. Biere. *Effiziente μ-Kalkül Modellprüfung mit Binären Entscheidungsdiagrammen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1997.

[3] J. Bitner, J. Jain, M. Abadir, J. Abraham, and D. Fussell. Efficient algorithmic verification using indexed BDD's. In *International Symposium on Fault-Tolerant Systems*, pages 266–275, 1994.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, Apr. 1989.

[5] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

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

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification, 10th International Conference, CAV'98*, volume 1427 of *Lecture Notes in Computer Sciences*, pages 147–158, 1998.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, Sept. 1994.

[9] E. M. Clarke and S. Jha. Symmetry and induction in model checking. *Lecture Notes in Computer Science*, 1000:455–470, 1995.

[10] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* An ACM Distinguished Dissertation. The MIT Press, 1988.

[11] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 85–94, San Francisco, 1995.

[12] J. Gergov and C. Meinel. Efficient Analysis and Manipulation of OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43:1197–1209, 1994.

[13] A. Gupta and A. L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *IEEE International Conference on Computer-Aided Design*, pages 192–199, 1993.

[14] T. Kropf. Benchmark-circuits for hardware-verification. ftp://goethe.ira.uka.de/-pub/hvg/benchmarks/Whole_documentation.ps.gz.

[15] K. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[16] K. McMillan. Hierarchical representations of discrete functions, with application to model checking. In *Proceedings of the 6th International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 41–54, 1994.

[17] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, Dallas, 1993.

[18] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 139–144. IEEE, 1993.

[19] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, Jan./Feb. 1983.

[20] B. Yang. Examples from SMV distribution. http://www.cs.cmu.edu/~bwolen/-software/.

[21] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. *Lecture Notes in Computer Science*, 1522:255–289, 1998.

Formal Verification of Descriptions with Distinct Order of Memory Operations

Gerd Ritter, Holger Hinrichsen, and Hans Eveking

Dept. of Electrical and Computer Engineering
Darmstadt University of Technology, D-64283 Darmstadt, Germany
`ritter/hinrichsen/eveking@rs.tu-darmstadt.de`

Abstract. Formal verification tools must often cope with large memory sizes and indirect addressing. This paper presents a new approach of how to handle memory operations in the symbolic simulation of designs with complex control logic, e.g., processors. The simulator is currently used to check the equivalence of two processor descriptions with distinct order of memory operations. During symbolic simulation, relationships between memory operations are automatically detected while addresses and the memory states are given symbolically to summarize many test-vectors. The integration of the technique in the equivalence checker is demonstrated by example designs.

1 Introduction

Formal verification of designs with complex control has to consider memories which have large sizes and are addressed indirectly. In [13] we presented a new approach for the automatic equivalence checking of designs with complex control. Our method is able to cope with different numbers of control steps and different implementational details in the descriptions to be compared. The verification tool combines symbolic simulation with a hierarchy of equivalence checking methods, including decision-diagram based techniques. A complete verification is possible in contrast to "classical" simulation since symbolic values are used. One symbolically simulated path corresponds in general to a large number of "classical" simulation runs. During the symbolic simulation, relationships between symbolic terms, e.g., the equivalence of two terms are detected and recorded. The equivalence detection for memory operations, which has to consider symbolic address relationships, is described in this paper. Addresses are compared using only the information of *equivalence classes* established previously. This makes a fast equivalence detection possible, which can cope with complex reorderings of memory operations.
Currently the symbolic simulator is used to check the *computational equivalence* of two descriptions of complex control logic. Two descriptions are computationally equivalent if both produce the same final values at definite time steps on the same initial values relative to a set of relevant variables. For instance, the two descriptions in Fig. 1 are computationally equivalent with respect to the final value of the relevant variable `z`. Parenthesis enclose synchronous parallel

Specification

```
rf[adrA]←a;
rf[adrB]←b;
mem[adr1]←val;
x←mem[adr2];
z←x+rf[adrR];
```

Implementation

```
(rf[adrB]←b,
 x←mem[adr2]);
(if adrA≠adrB
     then rf[adrA]←a endif,
 mem[adr1]←val);
(if adr1=adr2
     then z←val+rf[adrR]
     else z←x+rf[adrR] endif);
```

Fig. 1. Forwarding example

transfers in Fig. 1. The sequential composition operator ';' separates consecutive transfers. There are two examples for a reordering of memory operations in Fig. 1. Firstly, the order of the `read`- and the `store`-operation to `mem` is reversed in the implementation. Thus, `val` is forwarded if the addresses are identical, otherwise the value assigned to `x` is used. This is a typical forwarding example occurring in pipelined systems. Secondly, the order of the `store`-operations to the register file `rf` is reversed. This can, for example, happen during synthesis of architectures using data memory mapping, i.e., some single registers can be addressed by instructions in the same manner as registers of the register file. This is common for many microcontrollers, e.g., PIC, 8051, etc. Synthesis can change the order of accesses to this "common" data memory, e.g., by introducing pipelining. Formal verification has to consider the access to registers and register file by a single memory model. Otherwise it may remain unrevealed that, for example, the program counter is erroneously overwritten by an instruction due to a lacking address comparison.

Fig. 2 gives a simplified overview of the symbolic simulation algorithm which has been implemented iteratively for optimization [13]. The equivalence checker simulates symbolically all possible paths. Each symbolically executed assignment establishes an equivalence between the destination variable on the left and the term on the right side of an assignment. Additional equivalences between terms are detected during simulation. Equivalent terms are collected in equivalence classes. False paths are avoided by making only consistent decisions at branches in the description. When simulation reaches a condition C that cannot be decided in general but depends on the initial register and memory values (line 2, e.g., `adr1=adr2` in Fig. 1), a case split is performed (line 3). Note that **equivalence_check** is only called recursively with those parts of *spec* and *impl* that are not simulated yet.

A complete path is found when the end of both descriptions is reached. The first check of the final values of the registers (line 4) may lead to a false negative, since equivalence detection during the path search is not complete. Only relationships between terms that are fast to detect or are often crucial to demonstrate computational equivalence are considered on the fly. Therefore, more accurate equivalence checking methods, including decision diagram based techniques, are used to verify if computational equivalence is given in this path but was only

Algorithm equivalence_check

```
INPUT spec, impl;

 1. Simulate spec and impl in parallel UNTIL
    (a) a condition $C$ is reached that cannot be decided in general but depends on
        the initial register and memory values or
    (b) the end of both descriptions is reached.
 2. IF a condition $C$ blocks THEN
 3.     RETURN $(equivalence\_check\,(spec, impl)\,|_{C=FALSE}) \wedge$
               $(equivalence\_check\,(spec, impl)\,|_{C=TRUE})$
 4. ELSEIF final values of registers are equivalent THEN
 5.     RETURN(TRUE);
 6. ELSE
 7.     Use more accurate equivalence checks;
 8.     IF (final values of registers are equivalent) $\vee$
           (a condition has been decided inconsistently) THEN
 9.         RETURN(TRUE);
10.     ELSE
11.         RETURN(FALSE);
```

Fig. 2. Simplified algorithm of the symbolic simulation

not detected on the fly or a condition has been decided inconsistently (line 8), i.e., a false path is reached. Otherwise the counterexample with relevant details about the simulation run for debugging is reported. Our automatic verification process does not require insight of the designer into the verification process.
[13] gives a detailed description of the symbolic simulation, the decision process and the decision diagram based techniques. The efficiency of our symbolic simulation is demonstrated and compared with other approaches. The focus of this paper is the equivalence detection for memory operations, which must consider large memory sizes and indirect addressing.
Some related work is reviewed in section 2. The internal data structure, the memory model and some terminology are given in section 3. Section 4 overviews the equivalence detection for memory operations. A more detailed description is given for `read`-operations in section 5 and for `store`-operations in section 6. Experimental results are presented in section 7. Finally, section 8 gives a conclusion.

2 Related Work

Various representations for memory operations have been proposed for formal verification of designs with complex control. States are represented by decision diagrams in [4] for traversing an automata for model checking and in [3] for equivalence checking. This permits the representation of a register file, but not, e.g., of a large data memory due to the sensitivity to graph explosion. [14] uses decision diagrams combined with an encoding technique to represent uninterpreted symbols. Memories *and* functional units (e.g., the ALU) are abstracted by memory models. OBDD's represent the addresses of those models and are used for address comparison. The approach has the disadvantage that the complexity of the simulation increases exponentially if the memory models are addressed by data of other memory models. This is for example the case when a data memory is addressed by an ALU and writes to the register file. Note that the ALU has to be represented by a memory model. Therefore, the processor they verified in [14] contains neither a data-memory nor branching.

SVC (the Stanford Validity Checker) [1, 2, 11] is a proof tool for automatic verification of formulas which can contain the two array operations *read* and *write* to model memory operations. Verification of control logic is possible using SVC if the verification task can be reduced to a formula which is sufficient to demonstrate the verification goal. For instance, [5] proposed an approach to generate such a formula for the verification of a pipelined system against its sequential specification. Formal verification of control logic with SVC compared to our symbolic simulation method is discussed in [13]. Relationships of *memory operations* are revealed by SVC basically by case analysis. A `read`-operation $\texttt{read(write(}s, aW, v), a1)$ after a `write`-operation is rewritten to $\texttt{ite(}a1 = aW, v, \texttt{read}(s, a1))$. A case analysis is required to prove that $\texttt{read(write(}s, aW, v), a1) = \texttt{read(write(}s, aW, v), a2)$ follows from $a1 = a2$. Our approach avoids case analysis *on memory operations*. Equivalences of memory operations as the example above are detected in a different manner during the simulation. Furthermore, rewriting and case analysis can become also not practicable in a formula prover if the memory operations cause to much case splittings. This is, for example, the case, if operands are read repeatedly from a memory and the result is written back. Consider a simple architecture, where an instruction with two source- and one destination-address is read from an instruction memory. The source values are read from data memory, are added and the result is written back. Finally the program counter is incremented and the next instruction is fetched. Equivalence checking of the data memory after, e.g., six instructions requires already 11,868,920 case splits using SVC (4,396s on a 300 MHz Sun Ultra II), if we reverse the order of the first two instructions addressing distinct places in the data memory. Our approach avoids these case splits.

[12] uses ACL2 to simulate symbolically executable formal specification without requiring expert interaction. Related is the work in [7], where pre-specified microcode sequences of the JEM1 microprocessor are simulated symbolically using PVS. [12] models memories as lists of symbolic values which represent the memory contents, i.e., the length of the lists grows with the memory size. This explicit modeling allows no symbolic values in indirect addressing, since e.g., a store-operation with an symbolic address can change *any* memory place. But the intention of [12] is completely different. The tool provides a *fast* simulation on *some* indeterminate data for debugging a specification, i.e., the instruction sequence at the machine level is fixed. Our tool copes not only with some indeterminate data but verifies every possible control flow.

3 Preliminaries

Our equivalence checker compares two *acyclic* descriptions at the rt-level. For many cyclic designs, e.g., pipelined machines the verification problem can also be reduced to the equivalence check of acyclic sequences [13].
The memory model used by the symbolic simulator assumes an infinite size of each memory in the descriptions. Similar to [5, 1], two array operations are used to model memory access: `read(mem,adr)` returns the value stored at the address `adr` of memory `mem`. The operation `store(mem,adr,val)` returns the whole memory state of `mem` after changing the memory state only at `adr` to `val`. The two operations are used for all accesses to arrays of a description that can be addressed *indirectly*. This includes not only, e.g., the data-memory of a processor but also an indirectly addressed register file. On the other hand, directly addressed memories, i.e., cases where the addresses are constants, need not to be modeled by the `read`/`store`-scheme. A directly addressed memory place can also be considered as a register which is practically done by replacing all memory operations by a new distinct register name (e.g., `mem[3]`$\leftarrow$`x` becomes `mem3`$\leftarrow$`x`).
The inherent timing structure of the initial description is expressed explicitly by indexing the register and memory names. An indexed register name or memory name is called a *RegVal*. A new *RegVal* with an incremented index is introduced after each assignment and after each `store`-operation to a memory. An additional upper index s or i distinguishes the *RegVals* of specification and implementation. For example, `ar`$\leftarrow$`a+b;` is replaced by $\texttt{ar}_2^s \leftarrow \texttt{a}_1^s\texttt{+b}_1^s$`;` in the specification if all registers have been already assigned once. The third `store`-operation to a memory `mem[adr]`$\leftarrow$`val;` becomes $\texttt{mem}_3^s \leftarrow \texttt{store}(\texttt{mem}_2^s, \texttt{adr}_1^s, \texttt{val}_1^s)$. The *RegVals* mem_2^s and mem_3^s represent the memory state before and after the `store`-operation. Only the initial register/memory names as anchors are identical in specification and implementation, since the equivalence of the two descriptions is tested with regard to arbitrary but identical initial register values and memory states. Checking computational equivalence consists of verifying that the *RegVals* of the register or memories with the highest index are always equivalent.
Our symbolic simulation method has to detect equivalent terms.

Definition 1 (Equivalence of terms). *Two terms or RegVals are equivalent* $\equiv_{term}$, *if under the decisions* $C_0, ..., C_n$ *taken preliminary on the path, their values are identical for all initial RegVals. The operator* $\downarrow$ *denotes that each case-split, leading to one of the decisions* $C_0, ..., C_n$, *constrains the set of possible initial RegVals.*

$$term_1 \equiv_{term} term_2 \Leftrightarrow \forall RegVal_{initial} : (term_1 \equiv term_2) \downarrow (C_0 \wedge C_1 ... \wedge C_n)$$

Equivalent terms are detected along valid paths, and collected in *equivalence classes*. We write $term_1 \equiv_{sim} term_2$ if two terms are in the same equivalence class established during simulation. If $term_1 \equiv_{sim} term_2$ then $term_1 \equiv_{term} term_2$. Note that *RegVals* describe also different memory states, e.g., $\texttt{mem}_1^s \equiv_{sim} \texttt{mem}_2^i$ indicates that the two memory states are equivalent. Initially, each *RegVal* and each term gets its own equivalence class. Equivalence classes are unified after every assignment, if two terms are identified to be equivalent and when deciding the condition of an *if-then-else*-clause in a case split. Equivalence classes permit to keep also track about unequivalences of terms:

Definition 2 (Unequivalence of terms). *Two terms or RegVals are unequivalent $\not\equiv_{term}$, if under the decisions $C_0, ..., C_n$ taken preliminary on the path their values are never identical for arbitrary initial RegVals:*

$$term_1 \not\equiv_{term} term_2 \Leftrightarrow \neg\exists RegVal_{initial} : (term_1 \equiv term_2) \downarrow (C_0 \wedge C_1 ... \wedge C_n)$$

We write $term_1 \not\equiv_{sim} term_2$ if two terms are identified to be $\not\equiv_{term}$ during simulation. Equivalence classes containing $\not\equiv_{sim}$ terms are unequivalent, i.e., they contain different constants or terms, that have been decided in a case split to be unequivalent. The information of the equivalence classes is required to decide conditions consistently each time a new *if-then-else* is reached, i.e., to avoid false paths.

4 Equivalences of Memory-Operations

Three types of equivalences have to be detected concerning memory-operations:

- **Value stored by a `store` is equivalent to a `read` Section 5.1, 5.3**
  A `read`-operation reads *always* a value previously stored by an unique `store`-operation. Note that the `read`-operation occurs after the `store`-operation during simulation, i.e., this equivalence is only checked for `read`-operations.
- **Equivalence of two `read`-operations Section 5.2, 5.3**
  Two `read`-operations are equivalent since they *always* yield the same value.
- **Equivalence of two `store`-operations Section 6**
  The resulting memory states are equivalent, i.e., the contents of the memories after the two `store`-operations in specification and implementation, respectively, are *always* identical. Often, the memory states *before* the `store`-operations are also equivalent, which is fast to check. The `stores` can also result in identical memory states in the opposite case for two reasons:
  - a `store`-operation is overwritten by succeeding `stores`, see section 6.2.
  - the order of `store`-operations to the memory is different in specification and implementation, see section 6.3.

  Our equivalence detection is hierarchical: first an identical `store`-order in both descriptions is assumed, i.e., the memory states are pairwise identical. Then possibly overwritten `stores` are considered. Only if still a `store`-operation has no equivalent counterpart in the other description *and* a fast pre-check is satisfied, the more time consuming technique presented in section 6.3 is used to detect a changed order of `store`-operations.

For many operations the equivalence of terms can be decided by simply testing if the arguments are $\equiv_{sim}$ or $\not\equiv_{sim}$ which avoids the expansion of the arguments. This is also consequently used for the equivalence detection of `read`- and `store`-operations. Only the information of the equivalence classes of the addresses is used, i.e., our address comparison checks if two addresses $adr1$ and $adr2$ are

1. in the same equivalence class, i.e., $adr1 \equiv_{sim} adr2$
2. are in unequivalent equivalence classes $adr1 \not\equiv_{sim} adr2$, or
3. if the equivalence depends on the initial register or memory values.

Expansion of arguments as in [14], where boolean expressions are evaluated, is avoided. Note that the equivalence detection first considers the arguments of a term, i.e., the subterms.
Due to the space limitations and to clarify the problems, we use the following abbreviations in the examples of the next sections:

- Only the relevant `read`- and `store`-operations and address relations are shown. The in general complex control structure (e.g., *if-then-else* clauses) and all assignments to registers which do not include a `read`-operation are omitted. We, therefore, consider always only *one path* of the symbolic simulation. Note that our equivalence detection for memory operations does *not* require additional case splits.
- It is assumed that equivalences/unequivalences of the addresses have already been determined by other equivalence detection techniques [13] or are caused by case splits at preceding conditions of *if-then-else*-clauses which are omitted, see above.
- Addresses or values with identical name in specification and implementation, i.e., without the upper index s or i stand for arbitrary terms, which are assumed to be detected previously to be $\equiv_{sim}$. Using $adr1$ can signify textually different terms in both descriptions, e.g., $adr1^s = \mathtt{a}_3^s + \mathtt{b}_2^s$ and $adr1^i = \mathtt{c}_1^i + \mathtt{a}_2^i$, which are equivalent if $\mathtt{b}_2^s \equiv \mathtt{c}_1^i$ and $\mathtt{a}_3^s \equiv \mathtt{a}_2^i$.
- The boxes below the examples indicate, which additional relationships of the addresses must hold for two terms or memory states to be equivalent.

5 Detecting Equivalences of Read-Operations

5.1 Reading a Previously Stored Value

If the address of a `read`-operation reading from a memory and the address of the last `store`-operation referring to this memory are $\equiv_{sim}$, then the value stored by this `store`-operation is always read. Fig. 3 (a) gives an example. The memory state $\mathtt{mem}_1$ resulting from the last `store`-operation is in this case the same as the first argument of the `read`-operation.
If there is another intervening `store`-operation as in Fig. 3 (b), this relationship does not hold, since the second `store`-operation can overwrite the value stored by the first. But if the address of the `read`-operation is $\not\equiv_{sim}$ to the address of the second `store`, its value is *never* read by this `read`-operation. For the `read` it seems as if the last `store` did not happen. In general, all preceding `stores` of a `read` with unequivalent addresses have to be ignored. This is done by calculating the *read access* of a `read`-operation, i.e., the relevant memory state. The addresses of all `store`-operations in-between this memory state and the `read`-operation are unequivalent to the address of the `read`. The `store`

(a)
$$\begin{array}{l}\texttt{mem}_1 \leftarrow \texttt{store}(\texttt{mem}, adr1, val1);\\ \texttt{reg}_1 \leftarrow \texttt{read}(\texttt{mem}_1, adrR);\end{array}$$
$$\boxed{\begin{array}{l} adr1 \equiv_{sim} adrR \\ \Rightarrow \texttt{reg}_1 \equiv_{sim} val1 \end{array}}$$

(b)
$$\begin{array}{l}\texttt{mem}_1 \leftarrow \texttt{store}(\texttt{mem}, adr1, val1);\\ \texttt{mem}_2 \leftarrow \texttt{store}(\texttt{mem}_1, adr2, val2);\\ \texttt{reg}_1 \leftarrow \texttt{read}(\texttt{mem}_2, adrR);\end{array}$$
$$\boxed{\begin{array}{l} (adr2 \not\equiv_{sim} adrR) \wedge (adr1 \equiv_{sim} adrR) \\ \Rightarrow \texttt{reg}_1 \equiv_{sim} val1 \end{array}}$$

Fig. 3. Reading previously stored values

previous to the *read access* has an address that is not unequivalent and its value might be read. If the address of this `store` is even $\equiv_{sim}$, the stored value is always read and, therefore, $\equiv_{sim}$ to the `read`-operation.

5.2 Equivalence of Read-Operations

Two `read`-operations from specification and implementation are equivalent if their addresses and their *read accesses* are equivalent. The equivalence of the

$$\begin{array}{ll} \textit{Specificiation} & \textit{Implementation} \\ \texttt{mem}_1^s \leftarrow \texttt{store}(\texttt{mem}, adr1, val1); & \texttt{mem}_1^i \leftarrow \texttt{store}(\texttt{mem}, adrX, valX); \\ \texttt{reg}_1^s \leftarrow \texttt{read}(\texttt{mem}_1^s, adrR); & \texttt{mem}_2^i \leftarrow \texttt{store}(\texttt{mem}_1^i, adr1, val1); \\ & \texttt{reg}_1^i \leftarrow \texttt{read}(\texttt{mem}_2^i, adrR); \end{array}$$
$$\boxed{adrR \not\equiv_{sim} adrX \Rightarrow \texttt{reg}_1^s \equiv_{sim} \texttt{reg}_1^i}$$
$adr1$ is assumed to be neither $\not\equiv_{sim}$ nor $\equiv_{sim}$ to $adrR$

Fig. 4. Equivalence of two `read`-operations

read accesses guarantees, that all locations of the memory where they might read from (depending on the concrete value of the symbolic address) are identical. This procedure fails in the example of Fig. 4 if $adr1$ is neither $\not\equiv_{sim}$ nor $\equiv_{sim}$ to $adrR$. The first `store` in the implementation is not relevant for the `read`-operation, if its address $adrX$ is unequivalent to $adrR$. But the *read accesses* of the two `read`-operations are not identical because of the blocking second `store` with $adr1$. Note that if $adr1 \not\equiv_{sim} adrR$ holds, the *read accesses* would be both `mem` and if $adr1 \equiv_{sim} adrR$ holds, `val1` would be read in both cases.
A supplementary check for two `read`-operations with equivalent addresses is provided to cope with mismatching *read accesses*. If stored value and address of the "blocking" `store`-operations are equivalent, the *read access* is calculated again for both `read`-operations *without* these `stores`. This process can be repeated until either equivalent *read accesses* are found, i.e., the `read`-operations are equivalent, or `store`-operations block which addresses or values stored are not equivalent. Note that the memory states of the blocking `store`-operations do not need to be equivalent, see the example in Fig. 4.

5.3 Re-checking Read-Operations

Our general equivalence detection considers that the equivalence of the arguments of two terms is in most of the cases already obvious, when the second term is found on the path. Therefore, it is sufficient to check only at the first occurrence of a term whether it is equivalent to some previously found term. When finding a read-operation the first time, not all equivalences and unequivalences relevant for equivalence detection may be already stated, see for instance the forwarding on x in Fig. 1. The equivalence of the read-operation in the specification and in the *else*-path of the implementation is only obvious *after* the case split setting $adr1 \not\equiv_{sim} adr2$. This case is common in processor design with pipelining. A value is read speculatively and used only if there is no data conflict. Otherwise the relevant value is forwarded. The example indicates, that it is important to check read-operations whenever the equivalence classes of the corresponding addresses are modified. Therefore, the read-operations found during the symbolic simulation are marked at the equivalence classes of their addresses as *dependent* read-operations. If there is a change of an equivalence class, either because it is unified or set unequivalent to another equivalence class, all *dependent* read-operations are checked again. In the example of Fig. 1, the read-operation in the specification is marked at the equivalence class of adr2. When setting the equivalence classes of adr1 and adr2 unequivalent, the equivalence of the read-operations is detected.

6 Detecting Equivalent Memory States

Detecting the equivalence of two memory states is necessary to show computational equivalence but also required to be able to argue about the equivalence of two read-operations in specification and implementation. Since a store-operation returns the whole new memory state, finding equivalent memory states is the same as detecting equivalent store-operations.

6.1 Identical Order of Store-Operations

In some designs, the order of store-operations is identical in the two descriptions to compare. A sufficient, but not necessary condition for the equivalence of two store-operations and, therefore, the resulting memory states, is that addresses, the values stored, and the *previous* memory states are pairwise in the same equivalence class, which is fast to test and, therefore, checked first when finding a new store-operation. The final value of a memory in implementation and specification depends on the last two stores on both sides, which use the result of the previous stores as first arguments. By means of an inductive argument, when building a list in order of appearance of the stores in implementation and specification, every store may have its “partner” on the other side, if the order of store-operations is identical. The first store-operations on both sides have the initial memory state as first argument, which is identical for both sides.
Specification and implementation can have also only *partially* identical orders of stores, which begin from two equivalent memory states. The latter may be either the initial memory state or memory states that have been identified to be equivalent by one of the techniques described in section 6.2 and 6.3. The partially

identical `store`-order ends before the first `store`-operation-pair, where either address or stored value are not equivalent. The order of `store`-operations has to

`store(dmem,adr1,val1)`		`store(dmem,adr1,val1)`
`store(rf,adr2,val2)`	have the same store order	`store(dmem,adr3,val3)`
`store(dmem,adr3,val3)`	for both `rf` and `dmem`	`store(rf,adr2,val2)`
`store(rf,adr4,val4)`		`store(rf,adr4,val4)`

Fig. 5. Identical `store`-orders

be the same in specification and implementation only with regard to the same specific memory. The interleaving of `store`-operations to different memories can be arbitrary, see for example Fig. 5.

6.2 Overwritten Store-Operations

An identical `store`-order as defined in the previous section requires an equal number of `store`-operations, which is not given in the example of Fig. 6. Nevertheless, the final memory states are identical if the value stored by the second `store`-operation of the implementation is *always* overwritten by the third `store`-operation, i.e., if the addresses are $\equiv_{sim}$. This situation can occur for in-

Specificiation	*Implementation*
$\mathtt{mem}_1^s \leftarrow \mathtt{store}(\mathtt{mem}, adr1, val1)$;	$\mathtt{mem}_1^i \leftarrow \mathtt{store}(\mathtt{mem}, adr1, val1)$;
$\mathtt{mem}_2^s \leftarrow \mathtt{store}(\mathtt{mem}_1^s, adr2, val2)$;	$\mathtt{mem}_2^i \leftarrow \mathtt{store}(\mathtt{mem}_1^i, adrX, valX)$;
	$\mathtt{mem}_3^i \leftarrow \mathtt{store}(\mathtt{mem}_2^i, adr2, val2)$;

$$\boxed{adrX \equiv_{sim} adr2 \Rightarrow \mathtt{mem}_2^s \equiv_{sim} \mathtt{mem}_3^i}$$

Fig. 6. Example for an overwritten `store`-operation

stance if the second `store` is speculative, but the speculation fails and the third `store` is used to correct the fault. Let's assume that $val2$ and $valX$ are not $\equiv_{sim}$. Therefore, $\mathtt{mem}_2^s$ and $\mathtt{mem}_2^i$ cannot be in the same equivalence class and the equivalence detection of section 6.1 will fail. But for the last `store`-operation in the implementation there is no difference if the previous memory state is $\mathtt{mem}_1^i$ or $\mathtt{mem}_2^i$. Therefore the *relevant preceding memory state* is calculated for equivalence checking. This is either the memory state after the first preceding `store`-operation, which is not overwritten by the new `store`-operation or the initial memory state. Two `store`-operations in specification and implementation are equivalent if addresses, stored values and the *relevant preceding memory state* are $\equiv_{sim}$. This criterion copes with different number of overwritten `store`-operations in specification and implementation. Determining the *relevant preceding memory state* is fast, since, again, only the information of the equivalence classes is used. Furthermore, its calculation is only necessary if there exists a potential "counterpart" with $\equiv_{sim}$ address and stored value.
Note that by considering overwritten `stores`, there are some special cases where more than two `store`-operations, one of the specification and one of the implementation respectively, are in a single equivalence class. For instance, the mem-

ory states after the second and the third `store` in the implementation in Fig. 6 are identical if $adr2 \equiv_{sim} adrX$ and $val2 \equiv_{sim} valX$ holds.

6.3 Changed Order of Store-Operations

If the `store`-order is changed as in the example of Fig. 1 for `rf` and Fig. 7 for `mem`, the final memory states can be equivalent, if the addresses of the `store`-operations are $\not\equiv_{sim}$. A correct reordering of `store`-operations can be the result, for example, of synthesizing designs with data mapping, see section 1.
When a new `store`-operation is found and all previous checks fails, there might exist a `store` in the other description with equivalent address and stored value, which is the "counterpart" in a changed `store` order. Since the new `store` is the most recent in its description, there must be some `store`-operations *before* it, which happen *after* the "counterpart" in the other description. Assume that

	Specificiation		*Implementation*
$\mathtt{A}^s$	$\mathtt{mem}^s_1 \leftarrow \mathtt{store(mem},adrA,\ldots\mathtt{);}$	$\mathtt{A}^i$	$\mathtt{mem}^i_1 \leftarrow \mathtt{store(mem},adrA,\ldots\mathtt{);}$
$\mathtt{O1}^s$	$\mathtt{mem}^s_2 \leftarrow \mathtt{store(}$*overwritten later*$\mathtt{);}$	$\mathtt{D}^i$	$\mathtt{mem}^i_2 \leftarrow \mathtt{store(mem}^i_1,adrD,\ldots\mathtt{);}$
$\mathtt{B}^s$	$\mathtt{mem}^s_3 \leftarrow \mathtt{store(mem}^s_2,adrB,\ldots\mathtt{);}$	$\mathtt{C}^i$	$\mathtt{mem}^i_3 \leftarrow \mathtt{store(mem}^i_2,adrC,\ldots\mathtt{);}$
$\mathtt{O2}^s$	$\mathtt{mem}^s_4 \leftarrow \mathtt{store(}$*overwritten later*$\mathtt{);}$	$\mathtt{O3}^i$	$\mathtt{mem}^i_4 \leftarrow \mathtt{store(}$*overwritten later*$\mathtt{);}$
$\mathtt{C}^s$	$\mathtt{mem}^s_5 \leftarrow \mathtt{store(mem}^s_4,adrC,\ldots\mathtt{);}$	$\mathtt{B}^i$	$\mathtt{mem}^i_5 \leftarrow \mathtt{store(mem}^i_4,adrB,\ldots\mathtt{);}$
$\mathtt{D}^s$	$\mathtt{mem}^s_6 \leftarrow \mathtt{store(mem}^s_5,adrD,\ldots\mathtt{);}$		

$$\boxed{(adrD \not\equiv_{sim} adrC) \wedge (adrD \not\equiv_{sim} adrB) \wedge (adrB \not\equiv_{sim} adrC) \Rightarrow \mathtt{mem}^s_6 \equiv_{sim} \mathtt{mem}^i_5}$$

Fig. 7. Changed order of `store`-operations

the new `store` is $\mathtt{D}^s$ and the "counterpart" $\mathtt{D}^i$ in Fig. 7. The stores `B` and `C` are before `D` in the specification but after `D` in the implementation. The `stores O1, O2, O3` are overwritten by succeeding `store`-operations, i.e., $\mathtt{B}^s$, $\mathtt{C}^s$, $\mathtt{D}^s$ or $\mathtt{B}^i$. A valid reordering of the `store`-operations requires that the addresses of `D` on the one hand and `B`, `C` on the other hand are $\not\equiv_{sim}$. But we do not know that only `B` and `C` have to be checked, since there might be some overwritten stores $\mathtt{O1}^s$, $\mathtt{O2}^s$ `or` $\mathtt{O3}^i$ in-between or before `B` or `C` (see Fig. 7). For a quick test, first two sets containing all memory states *previous* to $\mathtt{D}^s/\mathtt{D}^i$ are determined, where all `store`-operations after those memory states and before $\mathtt{D}^s/\mathtt{D}^i$ have a determined address relationship; i.e., the addresses of those `store`-operations must be either $\not\equiv_{sim}$ to the address of $\mathtt{D}^s/\mathtt{D}^i$ or $\equiv_{sim}$ to the address of one of the succeeding `store`-operations. A changed `store`-order is only checked, if there are equivalent memory states in those two sets calculated for $\mathtt{D}^s$ and $\mathtt{D}^i$. In the following this is called that $\mathtt{D}^s$ and $\mathtt{D}^i$ have a *common access state.*
The next step is to determine the two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ containing the same `store`-operations appearing in the two descriptions in changed order. This is not obvious since only the end of $\mathcal{S}_1$ and the beginning of $\mathcal{S}_2$ are known. Furthermore, overwritten stores have to be considered correctly, i.e., $\mathcal{S}_1 = \{\mathtt{B}^s,\mathtt{C}^s,\mathtt{D}^s\}$ `and` $\mathcal{S}_2 = \{\mathtt{D}^i,\mathtt{C}^i,\mathtt{B}^i\}$ in Fig. 7. We assume in the following that all `store`-operations of the changed `store`-order have already appeared first in the implementation ($\mathtt{A}^i$ to $\mathtt{B}^i$) and now the last `store` of the opposite sequence $\mathtt{store}^{\mathcal{S}_1}_{end}$ is detected

during the simulation, i.e., $\mathtt{D}^s$. This is the first time where again equivalent memory states can be reached. Since $\mathtt{D}^s$ is the newest `store` detected during the simulation, the algorithm assumes that this is the last element missing and that it is the end of $\mathcal{S}_1$. Tracing back from this point, the first (previous) memory state is searched, which has an equivalent counterpart in the other description, i.e., $\mathtt{mem}_1^s$ and $\mathtt{mem}_1^i$ in Fig. 7. All preceding `stores` do not have to be considered since they lead to an equivalent memory state in implementation and specification. The `store`-operations in the two descriptions directly after this equivalent memory state $\mathtt{store}_{begin}^{\mathcal{S}_1}$ ($\mathtt{01}^s$) and $\mathtt{store}_{begin}^{\mathcal{S}_2}$ ($\mathtt{D}^i$) are the beginnings of the $\mathcal{S}_1$ and $\mathcal{S}_2$ before eliminating overwritten `store`-operations.
Overwritten `stores` can be removed easily in $\mathcal{S}_1$ since the latest $\mathtt{store}_{end}^{\mathcal{S}_1}$ ($\mathtt{D}^s$) is known. Tracing back from $\mathtt{store}_{end}^{\mathcal{S}_1}$ ($\mathtt{D}^s$) to $\mathtt{store}_{begin}^{\mathcal{S}_1}$ ($\mathtt{01}^s$), all `store`-operations with an address which is $\equiv_{sim}$ to the address of a succeeding `store` are eliminated, which results $\mathcal{S}_1 = \{\mathtt{B}^s, \mathtt{C}^s, \mathtt{D}^s\}$.
The end $\mathtt{store}_{end}^{\mathcal{S}_2}$ of the sequence $\mathcal{S}_2$ is unknown, which makes eliminating overwritten `store`-operations harder. Symbolic simulation may have already reached some `store`-operation *after* $\mathtt{B}^i$ which overwrites for instance $\mathtt{C}^i$ but has to be ignored to find $\mathcal{S}_2$ correctly. All `store`-operations *after* the unknown end $\mathtt{store}_{end}^{\mathcal{S}_2}$ ($\mathtt{B}^i$) do not have to be considered when eliminating overwritten `stores` in $\mathcal{S}_2$. $\mathcal{S}_2$ is determined by beginning with $\mathtt{store}_{begin}^{\mathcal{S}_2}$ and adding successively succeeding `stores`. Every time a new `store` is added, eventually overwritten `stores` are eliminated. This process is stopped, when the number of `store`-operations in $\mathcal{S}_2$ is the same as in $\mathcal{S}_1$.
Finally it is controlled, if every `store`-operation in $\mathcal{S}_1$ has its partner in $\mathcal{S}_2$ with $\equiv_{sim}$ address, $\equiv_{sim}$ stored value and *common access state* (see above). In this case, the memory states after $\mathtt{store}_{end}^{\mathcal{S}_1}$ and $\mathtt{store}_{end}^{\mathcal{S}_2}$, i.e., $\mathtt{D}^s$ and $\mathtt{B}^i$ are equivalent. Note that the technique described in this section is not limited with respect to the length of the changed `store` order, which is three in our example.
The handling of some exceptional situations is omitted in this paper due to the space limitations. Consider for example that a `store` $\mathtt{E}^i$ succeeds directly $\mathtt{B}^i$, which overwrites $\mathtt{C}^i$ with exactly the same value as $\mathtt{C}^i$. $\mathtt{D}^s$ is then not only equivalent to $\mathtt{B}^i$ but also to $\mathtt{E}^i$. This is detected by building two sequences $\mathcal{S}_{2a}$ and $\mathcal{S}_{2b}$ with $\mathtt{B}^i$ and $\mathtt{E}^i$ as last elements in this special case.

7 Experimental Results

Results of the verification of four designs with increasing complexity *concerning the equivalence detection of read/store-operations* are given in Tab. 1. In all examples, a sequential specification is compared with the corresponding pipelined implementation. The specifications reflect a subset of the instructions of the respective architecture, i.e., the Alpha-architecture from Digital [6], the DLX-architecture [8], and the PIC16C5X-processor from Microchip [10]. The implementations have been generated automatically from the specifications using the synthesis tool described in [9]. Verification of the pipelined designs was done using the flushing approach of [5], see also [13].

Tab. 1 gives the verification time, the number of instruction classes and the total number of paths checked during the symbolic simulation. The sixth column shows in how many paths `stores` are overwritten (section 6.2). The number of paths with changed order of the `store`-operations (section 6.3) is given in the last column. Paths with changed `store`-order are not considered in the sixth column although in these paths `stores` may be overwritten, too.
The `store`-order in the DLX-example is always identical in specification and implementation and no overwritten `stores` have to be considered. We obtained the same result for the verification of two structural DLX-descriptions with 5 pipeline-stages (see [13] for details). The Alpha-example requires additionally the consideration of overwritten `store`-operations. Consider two `stores` to the register file of the Alpha with equivalent addresses, which are executed consecutively in the sequential description. One of them is skipped if they are in different instruction stages which are parallelized by the synthesis tool. Note that the register file of the DLX (respectively the data memory) is always written in the same instruction stage.
All techniques presented in the previous sections are required to verify the two PIC-examples. The `store`-order was changed significantly in many paths after introducing pipelining. The reason is the data memory mapping used by this

Description	Pipeline stages	Instruction classes	Verification time	Total paths	Paths with `stores`	
					overwritten	changed order
DLX	5	5	561.1 s	225687	-	-
Alpha	3	10	7.84 s	2374	88	-
PIC 1	2	17	252.6 s	107655	3151	1741
PIC 2	2	17	379.6 s	161622	4338	5252

Table 1. Experimental results. Measurements on a Sun Ultra II (300 MHz)

architecture, i.e., single registers are addressed in the same manner as registers of the register file which makes synthesis (and verification) more complicated since numerous additional data-conflicts have to be resolved. This is also demonstrated by the higher complexity of PIC 2 compared to PIC 1. The only difference of PIC 1 is that the STATUS-register is excluded from data mapping. Another reason for the complexity of the PIC-examples compared to the Alpha- and DLX-example (which have more pipeline stages) is the larger number of instruction classes.
Verification with the symbolic simulator revealed a rare bug in the *implementation* of the synthesis tool. The decrement-instruction in the PIC-implementations was not writing correctly its value in the register file in a corner case due to an erroneous simplification of a condition during synthesis.
We verified the Alpha-example with the test for changed `store` order switched off and the DLX-example also without the checks for overwritten `stores`. The computation time changed only less than one second, which demonstrates that the overhead introduced by testing for complex `read/store`-schemes in the equivalence detection is acceptable.

8 Conclusion

Symbolic simulation must be able to handle memory operations to make formal verification of designs with complex control logic possible. The verification tool has to cope with two aspects: Firstly, the in general large sizes of the memories. We argue only about memory operations, i.e, `store`- and `read`-operations. Therefore, the symbolic simulator has to detect equivalences of memory operations in order to model correctly the behavior of the memory.
Secondly, indirect addressing has to be considered. This makes a reasoning process about the relationships of the addresses during the symbolic simulation necessary, since they can be arbitrary symbolic terms. Collecting equivalent symbolic terms in equivalence classes permits us to establish a fast address comparison for our equivalence detection methods. The new technique makes an efficient automatic equivalence checking of descriptions with complex reorderings of memory operations possible.

References

[1] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, volume 1166 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC'98*, 1998.

[3] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC'94*, 1994.

[4] J. R. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *DAC'90*, 1990.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, volume 818 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[6] Digital Equipment Corporation. *Alpha architecture handbook*, 1992.

[7] D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD'98*, volume 1522 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[8] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Morgan Kaufman, CA, second edition, 1996.

[9] H. Hinrichsen, H. Eveking, and G. Ritter. Formal synthesis for pipeline design. In *DMTCS+CATS'99, Auckland*, 1999.

[10] Microchip Technology Inc. *Microchip data book*, 1993.

[11] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD'95*, 1995.

[12] J. S. Moore. Symbolic simulation: an ACL2 approach. In *FMCAD'98*, volume 1522 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[13] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *CHARME'99*, *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[14] M. N. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *FMCAD'98*, volume 1522 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

Beyond Tamaki-Sato Style Unfold/Fold Transformations for Normal Logic Programs*

Abhik Roychoudhury[1], K. Narayan Kumar[1,2], C.R. Ramakrishnan[1], and I.V. Ramakrishnan[1]

[1] Dept. of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA.
{abhik,kumar,cram,ram}@cs.sunysb.edu
[2] Chennai Mathematical Institute, 92 G.N. Chetty Road, Chennai, India.
kumar@smi.ernet.in

Abstract. Unfold/fold transformation systems for logic programs have been extensively investigated. Existing unfold/fold transformation systems for normal logic programs allow only Tamaki-Sato style folding using clauses from a previous program in the transformation sequence: i.e., they fold using a single, non-recursive clause. In this paper we present a transformation system that permits folding in the presence of recursion, disjunction, as well as negation. We show that the transformations are correct with respect to various semantics of negation including the well-founded model and stable model semantics.

1 Introduction

Unfold/fold transformation systems for logic programs have been used for automated deduction [8, 17], and program specialization and optimization [2, 4, 10, 15]. Normal logic programs consist of definitions of the form $A{:-}\ \phi$ where A is an atom and ϕ is a boolean formula over atoms. Unfolding replaces an occurrence of A in a program with ϕ while folding replaces an occurrence of ϕ with A. Folding is called *reversible* if its effects can be undone by an unfolding, and *irreversible* otherwise.

Given a logic program P, an unfold/fold transformation system generates a sequence of programs $P = P_0, P_1, \ldots, P_n$, such that for all $0 \leq i < n$, P_{i+1} is obtained from P_i by applying one of the two transformations. Unfold/fold transformation systems are proved correct by showing that all programs in any transformation sequence $P_0, P_1, \ldots, P_n$ are equivalent under a suitable semantics, such as the well-founded model semantics for normal logic programs. A comprehensive survey of research on logic program transformations appears in [14].

As an illustration of unfolding/folding, consider the sequence of normal logic programs in figure 1. In the figure, P_1 is derived from P_0 by unfolding the occurrence of `q(X)` in the first clause of P_0. Program P_2 is derived from P_1 by folding the literal `q(Y)` in the body of the second clause of `p/1` into `p(Y)` by

* This work was partially supported by NSF grants CCR-9711386, CCR-9876242, CDA-9805735 and EIA-9705998.

using `p(X) :- q(X)` in P_0. This clause from a previous program which is used in a folding step (the clause `p(X) :- q(X)` of P_0 in this case) is called the *folder.*

An unfold/fold transformation system for definite logic programs was first described in a seminal paper by Tamaki and Sato [20]. It allows folding using a single clause only (*conjunctive* folding) from the initial program. This folder clause is required to be non-recursive, but need not be present in the current program P_i. Maher [12] proposed a transformation system using only reversible folding in which the folder clause is always drawn from the current program. However, reversibility is a restrictive condition that limits the power of unfold/fold systems by disallowing many correct transformations, such as the one used to derive P_2 from P_1 in Figure 1. Hence, there was considerable interest in developing irreversible unfold/fold transformation systems, for both definite and normal logic programs.

Existing unfold/fold transformation systems for normal logic programs [1, 13, 18, 19] are either extensions of Maher's reversible transformation system [12] or the original Tamaki-Sato system [20]. Even for definite logic programs, irreversible transformations of programs were, until recently, restricted to either folding using non-recursive clauses (see [7]) or a single recursive clause (see [9, 21]). In [16] we proposed a transformation framework for definite logic programs which generalized the above systems by permitting folding using multiple recursive clauses. Construction of such a general transformation system for *normal* logic programs has remained open. Below, we describe a solution to this problem.

Overview of the results: The main result of this paper is a unfold/fold transformation system that performs folding in the presence of recursion, disjunction as well as negation (see Section 2). The transformations of [16] associates *counters* with program clauses (*a la* Kanamori and Fujita [9]) to determine the applicability of fold and unfold transformations. In this paper, we extend this book-keeping to accommodate negative literals. We show that this extension is sufficient to guarantee that the resulting transformation system preserves a variety of semantics for normal logic programs, such as the well-founded model, stable model, partial stable model, and stable theory semantics. Central to this proof is the result due to Dung and Kanchanasut [6] that preserving the *semantic kernel* of a program is sufficient to guarantee the preservation of the different semantics for negation listed above. However, in contrast to [1] where this idea was used to prove the correctness of Tamaki-Sato style transformations,

`p(X):- q(X).`
`q([]).`
`q([X|Y]):- ¬r(X),q(Y).`
	`p([]).`
	`p([X|Y]):- ¬r(X),q(Y).`
	`q([]).`
	`q([X|Y]):- ¬r(X),q(Y).`
		`p([]).`
		`p([X|Y]):- ¬r(X),p(Y).`
		`q([]).`
		`q([X|Y]):- ¬r(X),q(Y).`

Program P_0	Program P_1	Program P_2

Fig. 1. Example of an unfold/fold transformation sequence

we present a two-step proof which explicitly uses the operational counterpart of semantic kernels (see Section 3). In the first step of our proof, we show that the transformations preserve positive ground derivations, which are derivations of the form $A \leadsto \neg B_1, \neg B_2, \ldots, \neg B_n$ such that there is a proof tree rooted at A with leaves labeled $\neg B_1$ through $\neg B_n$ (apart from true). We then show that preserving positive ground derivations is equivalent to preserving the semantic kernel of the program. Thus positive ground derivations are the operational analogues of semantic kernels.

This proof suggests that we can treat the negative literals in a program as atoms of new predicates defined in a different (external) module. The correctness of the transformation system is assured as long as the transformations respect module boundaries (see Section 4). This observation indicates how a transformation system originally designed for definite logic programs (such as the one we proposed in [16]) can be readily adapted for normal logic programs.

2 The Transformation System

Below we present our *unfold* and *fold* transformations for normal logic programs. In the following we assume familiarity with the standard notions of terms, substitutions, unification, atoms, literals. We will use the following symbols (possibly with primes and subscripts): P to denote a normal logic program; C and D for clauses; A, B to denote atoms ; L, K to denote literals ; $\mathcal{N}$ to denote sequence of literals and σ, θ for substitutions.

In any transformation sequence $P_0, P_1, \ldots, P_n$ we annotate each clause C in program P_i with a pair $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$ where $\gamma^i_{lo}(C), \gamma^i_{hi}(C) \in \mathbb{Z}$ and $\gamma^i_{lo}(C) \leq \gamma^i_{hi}(C)$. Thus, γ^i_{lo} and γ^i_{hi} are functions from the set of clauses in program P_i to the set of integers $\mathbb{Z}$. The transformation rules dictate the construction of γ^{i+1}_{lo} and γ^{i+1}_{hi} from γ^i_{lo} and γ^i_{hi}. We assume that for any clause C in the initial program P_0, $\gamma^0_{lo}(C) = \gamma^0_{hi}(C) = 1$. Intuitively, $\gamma^i_{lo}(C)$ and $\gamma^i_{hi}(C)$ for a clause C are analogous to the Kanamori-Fujita-style counters [9]; the separation of *hi* and *lo* permits us to store estimates of the counter values in the presence of disjunctive folding.

Rule 1 (Unfolding) Let C be a clause in P_i and A a positive literal in the body of C. Let $C_1, ..., C_m$ be the clauses in P_i whose heads are unifiable with A with most general unifiers $\sigma_1, ..., \sigma_m$. Let C'_j be the clause that is obtained by replacing $A\sigma_j$ by the body of $C_j\sigma_j$ in $C\sigma_j$ $(1 \leq j \leq m)$.

Then, assign $P_{i+1} := (P_i - \{C\}) \cup \{C'_1, ..., C'_m\}$. Set $\gamma^{i+1}_{lo}(C'_j) = \gamma^i_{lo}(C) + \gamma^i_{lo}(C_j)$ and $\gamma^{i+1}_{hi}(C'_j) = \gamma^i_{hi}(C) + \gamma^i_{hi}(C_j)$. The annotations of all other clauses in P_{i+1} are inherited from P_i. □

Rule 2 (Folding) Let $\{C_1, ..., C_m\}$ be clauses in P_i and $\{D_1, ..., D_m\}$ be clauses in P_j $(j \leq i)$ where C_l denotes the clause $A{:-}\ L_{l,1}, \ldots, L_{l,n_l}, L'_1, \ldots, L'_n$ and D_l denotes the clause $B_l{:-}\ K_{l,1}, \ldots, K_{l,n_l}$. Also, let
1. $\forall 1 \leq l \leq m\ \exists \sigma_l\ \forall 1 \leq k \leq n_l\ L_{l,k} = K_{l,k}\sigma_l$, where σ_l is a substitution.
2. $B_1\sigma_1 = B_2\sigma_2 = ... = B_m\sigma_m = B$

3. $D_1, ..., D_m$ are the only clauses in P_j whose heads are unifiable with B.
4. $\forall 1 \leq l \leq m$ σ_l substitutes the internal variables of D_l to distinct variables which do not appear in $\{A, B, L'_1, ..., L'_n\}$.
5. $\forall 1 \leq l \leq m$ $\gamma^j_{hi}(D_l) < \gamma^i_{lo}(C_l)$ + Number of positive literals in the sequence $L'_1, \ldots, L'_n$.

Then, assign $P_{i+1} := (P_i - \{C_1, ..., C_m\}) \cup \{C'\}$ where $C' \equiv A{:}-\ B, L'_1, ..., L'_n$. Set $\gamma^{i+1}_{lo}(C') = min_{1 \leq l \leq m}(\gamma^i_{lo}(C_l) - \gamma^j_{hi}(D_l))$ and $\gamma^{i+1}_{hi}(C') = max_{1 \leq l \leq m}(\gamma^i_{hi}(C_l) - \gamma^j_{lo}(D_l))$. The annotations of all other clauses in P_{i+1} are inherited from P_i. □

An Example: The following example (derived from [7]) illustrates the use of our basic unfold/fold transformation system.

```
C1 : in_position(X,L) :- in_odd(X,L), ¬ even(X). (1,1)
C2 : in_position(X,L) :- in_even(X,L), ¬ odd(X). (1,1)
C3 : in_odd(X,[X|L]).                            (1,1)
C4 : in_odd(X,[Y,Z|L]) :- in_odd(X,L).           (1,1)
C5 : in_even(X,[Y,X|L]).                         (1,1)
C6 : in_even(X,[Y,Z|L]) :- in_even(X,L).         (1,1)
```

In the above program, `in_odd(X,L)` (`in_even(X,L)`) is true if `X` appears in an odd (even) position in list `L`. Thus, `in_position(X,L)` is true if `X` is in an odd (even) position in list `L`, and `X` is not an even (odd) number. The `odd/1` and `even/1` predicates are encoded in the usual way and are not shown.
Unfolding `in_odd(X,L)` in C_1 we get the following clauses:

```
C7 : in_position(X,[X|L]) :- ¬ even(X).                   (2,2)
C8 : in_position(X,[Y,Z|L]) :- in_odd(X,L), ¬ even(X). (2,2)
```

Unfolding `in_even(X,L)` in C_2 yields the following clauses:

```
C9 :  in_position(X,[Y,X|L]) :- ¬ odd(X).                   (2,2)
C10 : in_position(X,[Y,Z|L]) :- in_even(X,L), ¬ odd(X). (2,2)
```

Finally, we fold clauses $\{C_8, C_{10}\}$ using the clauses $\{C_1, C_2\}$ from the initial program as the folder to obtain the following definition of `in_position/1`.

```
C7 :  in_position(X,[X|L]) :- ¬ even(X).          (2,2)
C9 :  in_position(X,[Y,X|L]) :- ¬ odd(X).         (2,2)
C11 : in_position(X,[Y,Z|L]) :- in_position(X,L). (1,1)
```

Note that the final step is an irreversible folding in presence of negation that uses multiple clauses as the folder. Such a folding step is neither allowed in Tamaki-Sato style transformation systems for normal logic programs [1, 18, 19] nor in reversible transformation systems [13].

Remark: We can maintain more elaborate book-keeping information than integer counters, thereby deriving more expressive unfold/fold systems. For instance, as in the SCOUT system described in [16], we can make the counters range over use a tuple of integers, and obtain a system that is strictly more powerful than the existing Tamaki-Sato-style systems [20, 21, 9, 18, 19, 7, 1]. The construction parallels that of the SCOUT system in [16]; details are omitted.

3 Proof of Correctness

In this section, we show that our unfold/fold transformation system is correct with respect to various semantics of normal logic programs. This proof proceeds in three steps. First, we introduce the notion of positive ground derivations and show that it is preserved by the transformations. Secondly, we show that preserving positive ground derivations is equivalent to preserving semantic kernel [6]. Finally, following [1], preserving semantic kernel implies that the transformation system is correct with respect to various semantics for normal logic programs including well-founded model, stable model, partial stable model, and stable theory semantics. We begin with a review of semantic kernels.

3.1 Semantic Kernel of a Program

Definition 1 (Quasi-Interpretation) [6, 1] *A quasi-interpretation of a normal logic program P is a set of ground clauses of the form $A{:-}\ \neg B_1, \ldots, \neg B_n$ $(n \geq 0)$ where $A, B_1, \ldots, B_n$ are ground atoms in the Herbrand Base of P.*

Quasi-interpretations form the universe over which semantic kernels are defined. For a given normal logic program P, the set of all quasi-interpretations of P (denoted $QI(P)$) forms a complete partial order with a least element (the empty set ϕ) with respect to the set inclusion relation $\subseteq$.

Definition 2 *Given a normal logic program P, let $Gnd(P)$ denote the set of all possible ground instantiations of all clauses of P. The function S_P on quasi-interpretations of P is defined as*

$$S_P : QI(P) \rightarrow QI(P)$$
$$S_P(I) = \{\mathcal{R}(C, D_1, \ldots, D_m) \mid C \in Gnd(P) \wedge D_i \in I,\ 1 \leq i \leq m\}$$

where, if $D_i (1 \leq i \leq m)$ are ground clauses

$$A_i{:-}\ \neg B_{i,1}, \ldots, \neg B_{i,n_i}\ (n_i \geq 0)$$

and $A_1, \ldots, A_m (m \geq 0)$ are the only positive literals appearing in the body of ground clause C, then $\mathcal{R}(C, D_1, \ldots, D_m)$ is the clause obtained by resolving the positive body literals $A_1, \ldots, A_m$ in C using clauses $D_1, \ldots, D_m$ respectively. □

If P is a definite program, then the function S_P is identical to the logical consequence operator T_P [11]. The semantic kernel of the program P is defined in terms of S_P as:

Definition 3 (Semantic Kernel) [6, 1] *The semantic kernel of a normal logic program P, denoted by $SK(P)$, is the least fixed point of the function S_P, i.e.,*

$$SK(P) = \bigcup_{n \in \omega} SK^n(P) \text{ where } SK^0(P) = \phi \text{ and } SK^{n+1}(P) = S_P(SK^n(P))$$

Example : Consider the following normal logic program P:

```
p :- ¬ q, r.
r :- ¬ r.
```

The semantic kernel of P will be computed as follows.

$SK^0(P) = \{\}$.
$SK^1(P) = S_P(SK^0(P)) = \{$ (`r :- ¬ r`) $\}$
$SK^2(P) = S_P(SK^1(P)) = \{$ (`r :- ¬ r`), (`p :- ¬ q, ¬ r`) $\}$
$SK^3(P) = S_P(SK^2(P)) = SK^2(P)$
Therefore, $SK(P) = \{$ (`r :- ¬ r`), (`p :- ¬ q, ¬ r`) $\}$

The following theorem from [1] formally states the equivalence of P and $SK(P)$ with respect to various semantics of normal logic programs.

Theorem 1 [Aravindan and Dung] *Let P be a normal logic program and $SK(P)$ be its semantic kernel. Then :*
(1) If P is a definite logic program, then P and $SK(P)$ have the same least Herbrand Model.
(2) If P is a stratified program, then P and $SK(P)$ have the same perfect model semantics.
(3) P and $SK(P)$ have the same well-founded model.
(4) P and $SK(P)$ have the same stable model(s).
(5) P and $SK(P)$ have the same set of partial stable models.
(6) P and $SK(P)$ have the same stable theory semantics.

3.2 Preserving the Semantic Kernel

We now show that in any transformation sequence $P_0, P_1, \ldots, P_n$ where $\forall 0 \leq i < n$ P_{i+1} is obtained from P_i by applying unfolding (rule 1) or folding (rule 2), the semantic kernel is preserved, i.e., $SK(P_0) = SK(P_1) = \ldots = SK(P_n)$. To do so, we introduce the following notion of a positive ground derivation:

Definition 4 (Positive ground derivation) *A positive ground derivation of a literal in a normal logic program P is a tree T such that: (1) each internal node of T is labeled with a ground atom (2) each leaf node of T is labeled with a negative ground literal or the special symbol* `true`*, and (3) for any internal node A of T, A:– $L_1, \ldots, L_n$ must be a ground instance of a clause in program P where $L_1, \ldots, L_n$ are the children of A in T.*

Thus, consider any positive ground derivation T in program P. Let the root of T be labeled with the ground literal L and let $\mathcal{N}$ be the *sequence of negative literals* derived in T, i.e., $\mathcal{N}$ is formed by appending the negative literals appearing in the leaf nodes of T from left to right. Then we say that L derives $\mathcal{N}$ in P, and denote such derivations by $L \leadsto_P \mathcal{N}$ (and $L \leadsto \mathcal{N}$ if P is obvious from the context). We overload this notation, often denoting existence of such derivations also by $L \leadsto_P \mathcal{N}$. Note that if L is a ground negative literal, there is only one positive ground derivation for L in any program, namely the *empty* derivation $L \leadsto L$. We now define:

Definition 5 (Weight of a positive ground derivation) *Let $L \leadsto_P \mathcal{N}$ be a positive ground derivation. The number of internal nodes in this derivation (*i.e.* the number of nodes labeled with a ground positive literal) is called the weight of the derivation.*

Definition 6 (Weight of a pair) *Let $P_0, P_1, \ldots, P_n$ be a transformation sequence of normal logic programs. Let L be a ground literal, $\mathcal{N}$ be a (possibly empty) sequence of ground negative literals s.t. $L \leadsto_{P_0} \mathcal{N}$. Then, the weight of $(L, \mathcal{N})$, denoted by $w(L, \mathcal{N})$, is the minimum of the weights of positive ground derivations of the form $L \leadsto_{P_0} \mathcal{N}$.*

Note that for *any* program P_i in the transformation sequence, the weight of any pair $w(L, \mathcal{N})$ is defined as the weight of the smallest derivation $L \leadsto_{P_0} \mathcal{N}$.

Definition 7 *Let $P_0, P_1, ..., P_n$ be a transformation sequence of normal logic programs. A positive ground derivation $L \leadsto_{P_i} \mathcal{N}$ is said to be weakly weight-consistent if for every ground instance A:– $L_1, ..., L_k$ of a clause C used in this derivation, we have $w(A, \mathcal{N}_A) \le \gamma^i_{hi}(C) + \sum_{1 \le l \le k} w(L_l, \mathcal{N}_l)$ where $\mathcal{N}_A, \mathcal{N}_1, ..., \mathcal{N}_k$ are the sequence of negative literals derived from $A, L_1, ..., L_k$ in this derivation.*

Definition 8 *Let $P_0, P_1, ..., P_n$ be a transformation sequence of normal logic programs. A positive ground derivation $L \leadsto_{P_i} \mathcal{N}$ is said to be strongly weight-consistent if for every ground instance A:– $L_1, ..., L_k$ of a clause C used in this derivation, we have*

– $w(A, \mathcal{N}_A) \ge \gamma^i_{lo}(C) + \sum_{1 \le l \le k} w(L_l, \mathcal{N}_l)$
– $\forall 1 \le l \le k \;\; w(A, \mathcal{N}_A) > w(L_l, \mathcal{N}_l)$

where $\mathcal{N}_A, \mathcal{N}_1, ..., \mathcal{N}_k$ are the negative literal sequences derived from $A, L_1, ..., L_k$ in this derivation.

Definition 9 (Weight consistent program) *Let $P_0, P_1, \ldots, P_n$ be a transformation sequence of normal logic programs. Then, program P_i is said to be weight consistent if*

– *for any pair $(L, \mathcal{N})$, whenever L derives $\mathcal{N}$ in P_i, there is a strongly weight consistent positive ground derivation $L \leadsto_{P_i} \mathcal{N}$.*
– *every positive ground derivation in P_i is weakly weight consistent.*

Using the above definitions, we now state certain invariants which always hold after the application of any unfold/fold transformation.

– $I1(P_i) \equiv \forall L \forall \mathcal{N}$ (L derives $\mathcal{N}$ in $P_0 \Leftrightarrow L$ derives $\mathcal{N}$ in P_i).
– $I2(P_i) \equiv$ P_i is a weight consistent program

We now show that these invariants are maintained after every unfolding and folding step. This allows us to claim that the set of positive ground derivations of P_0 is identical to the set of positive ground derivations of program P_i.

Lemma 1 *If $(\forall j \leq i\ I1(P_j))$ holds, then $\forall L \forall \mathcal{N}$ (L derives $\mathcal{N}$ in $P_{i+1} \Rightarrow L$ derives $\mathcal{N}$ in P_i)*

Lemma 2 (Preserving Weak Weight Consistency) *Let $P_0, ..., P_i, P_{i+1}$ be an unfold/fold transformation sequence s.t. $\forall 0 \leq j \leq i\ I1(P_j) \wedge I2(P_j)$. Then, all positive ground derivations of P_{i+1} are weakly weight consistent.*

The proofs for both Lemma 1 and 2 follow by induction on the weight of positive ground derivations in P_{i+1}. We now establish the main theorem concerning the preservation of positive ground derivations in a transformation sequence.

Theorem 2 *Let $P_0, P_1, \ldots$ be a sequence of normal logic programs where P_{i+1} is obtained from P_i by applying unfolding (rule 1) or folding (rule 2). Then $\forall i \geq 0\ I1(P_i) \wedge I2(P_i)$.*

Proof : The proof proceeds by induction on i. For the base case, $I1(P_0)$ holds trivially, and $I2(P_0)$ holds because every positive ground derivation of P_0 is weakly weight consistent, and for any pair $(L, \mathcal{N})$ the smallest positive ground derivation $L \rightsquigarrow_{P_0} \mathcal{N}$ is strongly weight consistent.

For the induction step, we need to show $I1(P_{i+1}) \wedge I2(P_{i+1})$. By Lemma 1 we have $L \rightsquigarrow_{P_{i+1}} \mathcal{N} \Rightarrow L \rightsquigarrow_{P_i} \mathcal{N}$, and by Lemma 2 we know that all positive ground derivations of P_{i+1} are weakly weight consistent. We need to show that *(i)* $L \rightsquigarrow_{P_i} \mathcal{N} \Rightarrow L \rightsquigarrow_{P_{i+1}} \mathcal{N}$, and *(ii)* for any pair $(L, \mathcal{N})$ s.t. $L \rightsquigarrow_{P_{i+1}} \mathcal{N}$, there exists a strongly weight consistent derivation $L \rightsquigarrow \mathcal{N}$ in P_{i+1}. Thus, it suffices to prove that for any pair $(L, \mathcal{N})$ s.t $L \rightsquigarrow_{P_i} \mathcal{N}$, there exists a strongly weight consistent derivation $L \rightsquigarrow_{P_{i+1}} \mathcal{N}$.

Consider a pair $(L, \mathcal{N})$ such that $L \rightsquigarrow_{P_i} \mathcal{N}$. Since P_i is weight consistent, therefore there exists a strongly weight consistent derivation $L \rightsquigarrow \mathcal{N}$ in P_i. Let this be called Dr. We now construct a strongly weight consistent derivation $Dr' \equiv L \rightsquigarrow_{P_{i+1}} \mathcal{N}$. Construction of Dr' proceeds by induction on the *weight of* $(L, \mathcal{N})$ *pairs*. The base case occurs when L is a negative literal, $\mathcal{N} = L$ and $w(L, \mathcal{N}) = 0$. We then trivially have the same derivation $L \rightsquigarrow \mathcal{N}$ in P_{i+1} as well. Otherwise if L is a positive literal, let C be the clause used at the root of Dr. Let $L{:\!-}\ L_1, \ldots, L_n$ be the ground instantiation of C used at the root of Dr. Since Dr is strongly weight consistent $w(L, \mathcal{N}) > w(L_l, \mathcal{N}_l)$ where $\mathcal{N}_l$ is the sequence of negative literals derived by L_l for all $1 \leq l \leq n$. Hence, we have strongly weight consistent derivations $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$. We construct Dr' by considering the following cases :

Case 1: C is inherited from P_i to P_{i+1}

Dr' is constructed with the clause $L{:\!-}\ L_1, \ldots, L_n$ at the root and then appending the derivations $L_l \rightsquigarrow_{P_{i+1}} \mathcal{N}_l$ for all $1 \leq l \leq n$. This derivation Dr' is strongly weight consistent.

Case 2: C is unfolded.

Let the L_1 be the positive body literal of C that is unfolded. Let the clause used to resolve L_1 in the derivation Dr be C_1 and the ground instance of C_1 used be

L_1:– $L_{1,1}, \ldots, L_{1,k}$. By definition of unfolding L:– $L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ is a ground instance of a clause C_1' in P_{i+1} with $\gamma_{lo}^{i+1}(C_1') = \gamma_{lo}^i(C) + \gamma_{lo}^i(C_1)$. Also, let $\mathcal{N}_{1,1}, \ldots, \mathcal{N}_{1,k}$ be the sequence of negative literals derived by $L_{1,1}, \ldots, L_{1,k}$ in Dr. Then, by strong weight consistency $w(L_{1,l}, \mathcal{N}_{1,l}) < w(L_1, \mathcal{N}_1) < w(L, \mathcal{N})$ for all $1 \leq l \leq k$. Thus we have strongly weight consistent derivations $L_{1,l} \leadsto_{P_{i+1}} \mathcal{N}_{1,l}$. We construct Dr' by applying L:– $L_{1,1}, \ldots, L_{1,k}, L_2, \ldots, L_n$ at the root and then appending the strongly weight consistent derivations $L_{1,l} \leadsto_{P_{i+1}} \mathcal{N}_{1,l}$ (for all $1 \leq l \leq k$) and $L_l \leadsto_{P_{i+1}} \mathcal{N}_l$ (for all $2 \leq l \leq n$). Since Dr is strongly weight consistent, therefore

$$\begin{array}{ll} & w(L,\mathcal{N}) \geq \gamma_{lo}^i(C) + \sum_{1\leq l\leq n} w(L_l, \mathcal{N}_l) \\ \text{and} & w(L_1,\mathcal{N}_1) \geq \gamma_{lo}^i(C_1) + \sum_{1\leq l\leq k} w(L_{1,l}, \mathcal{N}_{1,l}) \\ \Rightarrow & w(L,\mathcal{N}) \geq \gamma_{lo}^{i+1}(C_1') + \sum_{1\leq l\leq k} w(L_{1,l}, \mathcal{N}_{1,l}) + \sum_{2\leq l\leq n} w(L_l, \mathcal{N}_l) \end{array}$$

This shows that Dr' is strongly weight consistent.

Case 3: C is folded

Let C (potentially with other clauses) be folded, using folder clause(s) from $P_j (j \leq i)$, to clause C' in P_{i+1}. Assume that $L_1, \ldots, L_k$ are the instances of the body literals of C which are folded. Then, C' must have a ground instance of the form $L : -B, L_{k+1}, \ldots, L_n$, where B:– $L_1, \ldots, L_k$ is a ground instance of a folder clause D in P_j. Since, we have derivations $L_l \leadsto_{P_i} \mathcal{N}_l$ for all $1 \leq l \leq k$, therefore by $I1(P_i) \wedge I1(P_j)$ there exist derivations $L_l \leadsto_{P_j} \mathcal{N}_l$. Then, there exists a derivation $B \leadsto_{P_j} \mathcal{N}_B$ where $\mathcal{N}_B$ is obtained by appending the sequences $\mathcal{N}_1, \ldots, \mathcal{N}_k$. Since P_j is a weight consistent program, this derivation must be weakly weight consistent, and therefore $w(B, \mathcal{N}_B) \leq \gamma_{hi}^j(D) + \sum_{1\leq l\leq k} w(L_l, \mathcal{N}_l)$. By strong weight consistency of Dr, we have

$$\begin{aligned} w(L,\mathcal{N}) &\geq \gamma_{lo}^i(C) + \sum_{1\leq l\leq k} w(L_l,\mathcal{N}_l) + \sum_{k+1\leq l\leq n} w(L_l,\mathcal{N}_l) \\ &\geq \gamma_{lo}^i(C) + w(B,\mathcal{N}_B) - \gamma_{hi}^j(D) + \sum_{k+1\leq l\leq n} w(L_l,\mathcal{N}_l) \qquad \cdots\cdots(*) \\ &> w(B,\mathcal{N}_B) \text{ (by condition (5) of folding)} \end{aligned}$$

Hence there exists a strongly weight consistent derivation $B \leadsto_{P_{i+1}} \mathcal{N}_B$. We now construct Dr' with L:– $B, L_{k+1}, \ldots, L_n$ at the root and then appending below the strongly weight consistent derivations $B \leadsto_{P_{i+1}} \mathcal{N}_B, L_{k+1} \leadsto_{P_{i+1}} \mathcal{N}_{k+1}, \ldots, L_n \leadsto_{P_{i+1}} \mathcal{N}_n$. To show that Dr' is strongly weight consistent, note that $\gamma_{lo}^{i+1}(C') \leq \gamma_{lo}^i(C) - \gamma_{hi}^j(D)$ since C and D are folded and folder clauses. Combining this with (*),

$$w(L,\mathcal{N}) \geq \gamma_{lo}^{i+1}(C') + w(B,\mathcal{N}_B) + \sum_{k+1\leq l\leq n} w(L_l,\mathcal{N}_l)$$

This completes the proof. □

Thus we have shown that all positive ground derivations are preserved at every step of our transformation. Now we show how our notion of positive ground derivations directly corresponds to the notion of semantic kernel. Intuitively, this connection is clear, since a clause in the semantic kernel of program P is derived by repeatedly resolving the positive body literals of a ground instance of a clause in P until the body contains only negative literals. Formally, we prove that :

Theorem 3 *Let P be a normal logic program and $A, B_1, \ldots, B_n (n \geq 0)$ be ground atoms in the Herbrand base of P. Let $\mathcal{N}$ be the sequence $\neg B_1, \ldots, \neg B_n$. Then, A derives $\mathcal{N}$ in P iff $(A{:}-\ \mathcal{N}) \in SK(P)$*

Proof Sketch: We prove $A \leadsto_P \mathcal{N} \Rightarrow (A{:}-\ \mathcal{N}) \in SK(P)$ by strong induction on the weight (*i.e.* the number of internal nodes, refer definition 5) in the derivation $A \leadsto_P \mathcal{N}$. The proof for $(A{:}-\ \mathcal{N}) \in SK(P) \Rightarrow A \leadsto_P \mathcal{N}$ follows by fixed-point induction. □

We can now prove that the semantic kernel is preserved across any unfold/fold transformation sequence.

Corollary 3 (Preservation of Semantic Kernel) *Suppose $P_0, \ldots, P_n$ is a sequence of normal logic programs where P_{i+1} is obtained from P_i by unfolding (Rule 1) or folding (Rule 2). Then $\forall 0 \leq i < n\ SK(P_i) = SK(P_0)$.*

Proof: We prove that $SK(P_0) = SK(P_i)$ for any arbitrary i. By Theorem 2 we know that $A \leadsto_{P_0} \mathcal{N} \Leftrightarrow A \leadsto_{P_i} \mathcal{N}$ for any ground atom A and sequence of ground negative literals $\mathcal{N}$. Then, using Theorem 3 we get $(A{:}-\ \mathcal{N}) \in SK(P_0) \Leftrightarrow (A{:}-\ \mathcal{N}) \in SK(P_i)$. Thus, $SK(P_0) = SK(P_i)$. □

Following Theorem 1 and Corollary 3 we have:

Theorem 4 (Correctness of Unfolding/Folding) *Let $P_0, \ldots, P_n$ be a sequence of normal logic programs where P_{i+1} is obtained from P_i by an application of unfolding (Rule 1) or folding (Rule 2). Then, for all $0 \leq i < n$ we have*
(1) If P_0 is a definite logic program, then P_0 and P_i have the same least Herbrand Model.
(2) If P_0 is a stratified program, then P_0 and P_i have the same perfect model semantics.
(3) P_0 and P_i have the same well-founded model.
(4) P_0 and P_i have the same stable model(s).
(5) P_0 and P_i have the same set of partial stable models.
(6) P_0 and P_i have the same stable theory semantics.

4 Discussion

In this paper we have presented an unfold/fold transformation system, which to the best of our knowledge, is the first to permit folding in the presence of recursion, disjunction, as well as negation. Such a system is particularly important for verifying temporal properties of parameterized concurrent systems (such as a n-bit shift register for any n) using logic program evaluation and deduction [5, 17].

The transformation system presented in this paper can be extended to incorporate a *goal replacement* rule which allows the replacement of a conjunction of atoms in the body of a clause with another semantically equivalent conjunction of atoms provided certain conditions are satisfied (which ensure preservation of weight consistency). In future, it would be interesting to study how we can perform multiple replacements simultaneously without compromising correctness (as discussed in [3]).

Apart from the transformation system, the details of the underlying correctness proof reveal certain interesting aspects generic to such transformation systems. First of all, our proof exploits a degree of modularity that is inherent in the unfold/fold transformations for logic programs. Consider a modular decomposition of a definite logic program where each predicate is fully defined in a single module. Each module has a set of "local" predicates defined in the current module and a set of "external" predicates used (and not defined) in the current module. It is easy to see that Lemma 1, 2 and Theorem 2 can be modified to show that unfold/fold transformations preserve the set of *local ground derivations* of a program. We say that $A \leadsto B_1, B_2, \ldots, B_n$ is a local ground derivation (analogous to a positive ground derivation), if each B_i contains an external predicate, and there is a proof tree rooted at A whose leaves are labeled with $B_1, \ldots, B_n$ (apart from true). Consequently, transformations of a normal logic program P, can be simulated by an equivalent positive program module Q obtained by replacing negative literals in P with new positive external literals. The newly introduced literals can be appropriately defined in a separate module. Thus any transformation system for definite logic programs that preserves local ground derivations also preserves the semantic kernels of normal logic programs.

Secondly, we showed that positive ground derivations form the operational counterpart to semantic kernels. This result, which makes explicit an idea in the proof of Aravindan and Dung [1], enables the correctness proof to be completed by connecting the other two steps: an operational first step, where the measure consistency technique is used to show the preservation of positive ground derivations and the final model-theoretic step that applies the results of Dung and Kanchanasut [6] relating semantic kernels to various semantics for normal logic programs.

Semantic kernel is a fundamental concept in the study of model-based semantics. By it very nature, however, semantic kernels cannot be used in proving operational equivalences such as finite failure and computed answer sets. The important task then is to formulate a suitable operational notion that plays the role of semantic kernel in the correctness proofs with respect to these equivalences.

References

[1] C. Aravindan and P.M. Dung. On the correctness of unfold/fold transformations of normal and extended logic programs. *Journal of Logic Programming*, pages 295–322, 1995.

[2] A. Bossi, N. Cocco, and S. Dulli. A method of specializing logic programs. *ACM TOPLAS*, pages 253–302, 1990.

[3] A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, February 1996.

[4] D. Boulanger and M. Bruynooghe. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, pages 495–521, 1993.

[5] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In *Proceedings of PLILP/ALP, LNCS 1490*, pages 1–20, 1998.

[6] P.M. Dung and K. Kanchanasut. A fixpoint approach to declarative semantics of logic programs. *Proceedings of North American Conference on Logic Programming*, 1:604–625, 1989.

[7] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In *Proceedings of PLILP, LNCS 844*, pages 340–354, 1994.

[8] T. Kanamori and H. Fujita. Formulation of Induction Formulas in Verification of Prolog Programs. *Proceedings of International Conference on Automated Deduction (CADE)*, pages 281–299, 1986.

[9] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. In *USA-Japan Seminar on Logics of Programs*, 1987.

[10] M. Leuschel, D. De Schreye, and A. De Waal. A conceptual embedding of folding into partial deduction : Towards a maximal integration. In *Joint International Conference and Symposium on Logic Programming*, pages 319–332, 1996.

[11] J.W. Lloyd. *Foundations of Logic Programming, 2nd Ed.* Springer-Verlag, 1993.

[12] M. J. Maher. Correctness of a logic program transformation system. Technical report, IBM T.J. Watson Research Center, 1987.

[13] M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.

[14] A. Pettorossi and M. Proietti. *Transformation of logic programs*, volume 5 of *Handbook of Logic in Artificial Intelligence*, pages 697–787. Oxford University Press, 1998.

[15] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proceedings of POPL*, pages 414–427, 1997.

[16] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *Principles and Practice of Declarative Programming (PPDP), LNCS 1702*, pages 396–413, 1999.

[17] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. Proofs by program transformations. *To appear in proceedings of Logic-based Program Synthesis and Transformation (LOPSTR)*, 1999.

[18] H. Seki. Unfold/fold transformation of stratified programs. *In Theoretical Computer Science*, pages 107–139, 1991.

[19] H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *In Journal of Logic Programming*, pages 5–23, 1993.

[20] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.

[21] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/ fold logic program transformation. Technical report, Ibaraki University, Japan, 1986.

On $L^k(Q)$ Types and Boundedness of $IFP(Q)$ on Finite Structures[⋆]

Anil Seth

Silicon Automation Systems
Fac-A, 3008, 12 B Main, 8th Cross, HAL II Stage
Indiranagar, Bangalore - 560008, India.
seth@sasi.com

Abstract. We show that there is a class C of finite structures and a PTIME quantifier Q such that

(1) $IFP(Q)$ is bounded on C but $L^{\omega}_{\infty,\omega}(Q) \neq PFP(Q) \neq IFP(Q) = FO(Q)$ over C.
(2) For all $k \geq 2$, $IFP^k(Q)$ is bounded but not uniformly bounded over C. ($IFP^k(Q)$ denotes the k-variable fragment of $IFP(Q)$)
(3) For all $k \geq 2$, $IFP^k(Q)$ is not uniformly bounded over C but $IFP^k(Q) = L^k(Q)$ over C.

1 Introduction

First order logic (FO) has limited expressive power on finite structures. Therefore, fixed point logics, such as LFP and PFP have been studied widely in finite model theory. On ordered structures LFP expresses exactly PTIME queries but on unordered structures its expressive power is limited and is hard to characterize.

In the study of expressive power of LFP and other fixed point logics, an important role is played by an infinitary logic with finite variables, denoted $L^{\omega}_{\infty,\omega}$. LFP can be viewed as a fragment of $L^{\omega}_{\infty,\omega}$. To understand the relative expressive power of FO, LFP and $L^{\omega}_{\infty,\omega}$ on subclasses of finite structures, McColm made two conjectures in [10]. These conjectures are centered around the notion of boundedness for fixed point formulae over a class of finite structures. Let C be any class of finite structures. McColm's first conjecture states that LFP collapses to FO on C iff LFP is bounded on C. McColm's second conjecture states that $L^{\omega}_{\infty,\omega}$ collapses to FO on C iff LFP is bounded on C.

The first conjecture was refuted in [5] and the second conjecture was confirmed in [8]. Further, in [9] ramified versions of these conjectures, that is analogous questions for fixed variable fragment of these logics, have been studied. For this a suitable definition of k-variable fragment of LFP, LFP^k is formulated. The notion of boundedness for LFP^k over a class of structures can be defined

[⋆] This work was done while the author was with the Institute of Mathematical Sciences, Madras.

in the usual manner, by requiring bounded induction for each LFP^k formula (or system of L^k formulae). However for LFP^k a stronger notion of boundedness called uniform boundedness [9] can also be defined. Uniform boundedness requires that there is a constant which bounds the number of inductive stages over C for *all* systems of formulae in LFP^k. It is shown in [9] that on any class C of finite structures, $L^k_{\infty,\omega}$ collapses to L^k on C iff LFP^k is uniformly bounded on C.

After some successful development of model theory of finite variable logics in recent years, for example as witnessed by elegant results mentioned above, it is natural to examine if these results also hold for more general logics or if the techniques developed there can also be applied to richer contexts. Recently, extensions of first order, fixed point and infinitary logics with generalized quantifiers have been studied extensively [7,2,3]. In this paper, we study questions analogous to McColm's conjectures and ramified versions of these for logics with generalized quantifiers.

We show that McColm's second conjecture cannot be extended to logics with arbitrary generalized quantifiers whereas a ramified version of the second conjecture does hold for arbitrary finite set of generalized quantifiers. Our main result is a construction which disproves extension of McColm's second conjecture for logics with generalized quantifiers. We construct a generalized quantifier Q and a class C of finite structures such that $IFP(Q)$ is bounded over C but the number of $L^2(Q)$ types realized is unbounded over C. This in turn implies that $L^\omega_{\infty,\omega}(Q) \neq \mathrm{FO}(Q)$ over C. Further, we can also construct a PTIME computable generalized quantifier with the above properties.

A byproduct of this construction is a different proof of a result in [3], that there is a PTIME computable generalized quantifier Q such that $L^k(Q)$ types cannot be ordered in $IFP(Q)$. The construction of [3] does not imply the result mentioned above as the class C over which $IFP(P) \neq PFP(P)$ (and hence $FO(P) \neq L^\omega_{\infty,\omega}(P)$) is established in [3] for a quantifier P, admits an unbounded induction because a linear order is explicitly defined on part of each structure in C. The linear order in that construction is used in an essential way.

For the same class of structures C and PTIME, quantifier Q, used to disprove the extension of McColm's second conjecture, we show that $IFP^k(Q)$ is bounded but not uniformly bounded on C. To our knowledge, this is the first example in which the two notions of boundedness for (extensions of) fixed point logic are provably distinct. It also follows easily that $IFP^k(Q) = L^k(Q)$ on this class which disproves a version of ramification of the first conjecture with generalized quantifiers. It is interesting to note that these results hold for an extension of IFP which is still in PTIME. It should be noted that both these questions, boundedness vs. uniform boundedness and ramified version of McColm's first conjecture are open when no generalized quantifiers are allowed.

Our construction of the generalized quantifier is by diagonalization. Generally, this process for constructing quantifiers can be thought of as similar to oracle constructions in complexity theory. However, the diagonalization process in constructing quantifiers is more involved than the usual oracle constructions in complexity theory. One reason for this is that in constructing generalized quantifiers one does not operate just on the level of strings or a specific representation of a structure but on a more abstract level of the structure itself. If we include (exclude) a structure in the class defining a generalized quantifier then we also have to include (exclude) all structures isomorphic to it, in the class.

In our construction we need to define an equivalence relation coarser than isomorphism on structures such that whenever we include (exclude) a structure in the class defining generalized quantifier then we also have to include (exclude) all structures related to it, in the class. To go through the construction we look into the structure of equivalence classes of this relation and find an invariant of an equivalence class. The technical machinery devloped to achieve this may be useful to construct generalized quantifiers in other contexts also.

2 Preliminaries

We assume the reader to be familiar with basic notions of finite model theory. In this section, we will present only some concepts and notations relevant to us. For a detailed introduction see, [4,1].

We consider finite structures over (finite) relational vocabularies. If $\mathfrak{A}$ denotes a structure then $|\mathfrak{A}|$ will denote its domain and $||\mathfrak{A}||$ will denote cardinality of the domain.

We use $\oplus$ to denote disjoint union of two structures. That is if $\mathfrak{A}, \mathfrak{B}$ are σ structures, then $\mathfrak{A} \oplus \mathfrak{B}$ is a σ structure with domain $\{0\} \times |\mathfrak{A}| \cup \{1\} \times |\mathfrak{B}|$. For any relation R or arity i in σ and any $d_1, \ldots, d_i \in |\mathfrak{A} \oplus \mathfrak{B}|$, $\mathfrak{A} \oplus \mathfrak{B} \models R(d_1, \ldots, d_i)$ iff either (i) $d_1 = 0a_1, \ldots, d_i = 0a_i$ and $\mathfrak{A} \models R(a_1, \ldots, a_i)$ or (ii) $d_1 = 1b_1, \ldots$ $\ldots, d_i = 1b_i$ and $\mathfrak{B} \models R(b_1, \ldots, b_i)$. The operation $\oplus$ extends to more than two structures in an obvious way.

Generalized Quantifiers: Let K be any class of structures over the vocabulary $\sigma = \{R_1, \ldots, R_m\}$, where R_i has arity n_i. We associate with K the generalized (or Lindström) quantifier Q_K.

For a logic L, define the extension $L(Q_K)$ as the set of formulas arising by adding to the formulae formation rules of L, the following new rule for formulae formation: if $\phi_1, \ldots, \phi_m$ are formulas of $L(Q_K)$ and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ are tuples of variables (with the length of $\boldsymbol{x}_i$ being n_i), then $Q_K \boldsymbol{x}_1, \ldots, \boldsymbol{x}_m(\phi_1, \ldots, \phi_m)$ is a formula of $L(Q_K)$. The type of this quantifier is the tuple $(n_1, \ldots, n_m)$ and its arity is $\max(n_i)$.

The semantics of the quantifier is given by:
$\mathfrak{A} \models Q_K \boldsymbol{x}_1, \ldots, \boldsymbol{x}_m(\phi_1(\boldsymbol{x}_1, \boldsymbol{y}), \ldots, \phi_m(\boldsymbol{x}_m, \boldsymbol{y}))[\boldsymbol{a}]$ if and only if $(|\mathfrak{A}|, \phi_1^{\mathfrak{A}}[\boldsymbol{a}], \ldots, \phi_m^{\mathfrak{A}}[\boldsymbol{a}]) \in K$, where $\phi_i^{\mathfrak{A}}[\boldsymbol{a}] = \{\boldsymbol{b} \in |\mathfrak{A}|^{n_i} \mid \mathfrak{A} \models \phi_i[\boldsymbol{b}, \boldsymbol{a}]\}$.

Fixed point logics, infinitary logic and types: The definition of LFP (Least fixed point logic) and IFP (Inflationary fixed point logic), can be found in [4,1]. As a technical remark, we just mention that we do not allow any parameter variables in the definition of fixed point, this is convenient for our purposes and is not a real restriction, see [4, Lemma 7.1.10 (b), pp. 171].

Both LFP and IFP are equivalent in expressive power (over finite structures) (see [6]). However, IFP is more robust in that it can be naturally extended with arbitrary generalized quantifiers, so we will use mostly IFP in this paper.

We denote the k-variable fragment of first order logic by L^k. For the definition of infinitary logic, L^k-types and $L^k(Q)$-types, see [1].

McColm's conjectures: A detailed exposition of McColm's conjectures may be found in [1,8,9]. These conjectures are centered around the following definition of boundedness.

Definition 1 *Let $\phi(x_1, \ldots, x_n, S)$ be a first order formula positive in S over vocabulary $\sigma \cup \{S\}$, where S is an n-ary relation symbol not in σ. ϕ is bounded over a class C of structures if there is an m such that on every structure $\mathfrak{A} \in C$, LFP iteration of ϕ converges in $\leq m$ steps. LFP is bounded over C if every first order formula ϕ as above is bounded over C.*

Note that boundedness for IFP can be defined in the same way as has been defined for LFP and LFP is bounded on a class of finite structures iff IFP is bounded on this class. In [10], McColm made following conjecture relating boundedness of LFP over a class C to the relative expressive power of fixed point and infinitary logic over C.

Conjecture Let C be a class of structures. The following are equivalent.

(i) LFP is bounded over C.
(ii) LFP = FO over C.
(iii) $L^{\omega}_{\infty,\omega}$ = FO over C.

The equivalence between (i) and (ii) above is termed as McColm's first conjecture in [8], and was refuted in [5]. The equivalence between (i) and (iii) is termed as McColm's second conjecture in [8], where it was shown to hold.

Bounded variable fragment of fixed point logics: In [9] the k-variable fragment of LFP, LFP^k, is defined as a system of positive L^k formulae. The semantics of this system is obtained by taking simultaneous fixed points of formulae of the system. Essentially the same definition works for $IFP^k(\mathbf{Q})$. We also define fragments of $IFP^k(\mathbf{Q})$ based on the number of components in the system.

Definition 2 *$IFP^{k,i}(\mathbf{Q})$ denotes the fragment of $IFP^k(\mathbf{Q})$ obtained by taking systems of at most i formulae of $L^k(\mathbf{Q})$.*

Given a system of $L^k(\mathbf{Q})$ formulae, its closure function and change function is defined in exactly the same way as in [9]. We recall the definition of *uniform* boundedness of $IFP^k(\mathbf{Q})$ over a class C of structures as in [9].

Definition 3 *Let k be a positive integer, C a class of σ structures and $\mathbf{Q}$ a set of generalized quantifiers. $IFP^k(\mathbf{Q})$ is* uniformly *bounded on C if there is a positive integer m_0 such that for all $l > 0$, for every system $S = (\phi_1, \ldots, \phi_l)$ of $L^k(\mathbf{Q})$ formulae and for every σ-structure $\mathfrak{A} \in C$ we have that, $ch(S)(\mathfrak{A}) \leq m_0$.*

Following is the ramified version of McColm's second conjecture, proved in [9].

Theorem 1 *[9] Let k be a positive integer. The following are equivalent over a class C of finite structures.*

1. *LFP^k is uniformly bounded over C.*
2. *$L^k_{\infty,\omega} = L^k$ over C.*

3 Constructing a Quantifier

In this section we construct a class C of rigid structures over the vocabulary $\Sigma = \{R\}$, consisting of one binary relation, and a quantifier P such that $IFP(P)$ is bounded on C but $L^\omega_{\infty,\omega}(P) \neq FO(P)$ over C. The quantifier P is over the signature σ consisting of one binary and two unary relations.

We construct a sequence of rigid connected graphs $\mathfrak{M}_1, \mathfrak{M}_2, \ldots, \mathfrak{M}_i, \ldots$. Let $\mathfrak{G}_i = \mathfrak{M}_1 \oplus \ldots \oplus \mathfrak{M}_i$. The class C is defined as $\{\mathfrak{G}_n | n \geq 1\}$. The quantifier P is obtained by diagonalization over a fixed enumeration $(\phi_i)_{i\in\omega}$ of all $IFP(P)$ formulae. Each $\mathfrak{G}_i$ has at least i many, $L^2(P)$ types and the number of iterations for each IFP occurrence in ϕ_i is bounded by $(d.||\mathfrak{G}_i||^d + d)$, for a fixed constant d, over the entire class C. For brevity, we use $\mathfrak{M}_{i,j}$ denote $\mathfrak{M}_i \oplus \ldots \oplus \mathfrak{M}_j$.

Following lemma gives some information about structures in class C. It can be easily proved using EF games.

Lemma 1 *Let k be a positive integer.*
Let $C_k = \{\mathfrak{A}_1 \oplus \ldots \oplus \mathfrak{A}_n \mid n \geq 1,$ each $\mathfrak{A}_i$ satisfies k-extension axioms $\}$. Then the number of L^k types realized in each structure $\mathfrak{A} \in C_k$ is bounded, that is, it is only a function of k and is independent of the structure.

Proof: Easy. □

Lemma 2 *Let k, m, n be positive integers. Let $\mathfrak{A} = \mathfrak{B}_0 \oplus \mathfrak{B}_1 \oplus \ldots \oplus \mathfrak{B}_n$ be a structure such that $\mathfrak{B}_1, \ldots, \mathfrak{B}_n$ satisfy k-extension axioms. Let $\boldsymbol{a_1} \in |\mathfrak{A}|^i$, $i \leq m$. Suppose $\boldsymbol{a_1}$ is extended to $(\boldsymbol{a_1}, \boldsymbol{a_2})$, where $\boldsymbol{a_2} \in |\mathfrak{B}_1 \oplus \ldots \oplus \mathfrak{B}_n|^{m-i}$ and $\boldsymbol{a_2}$ has at least one element not in $\boldsymbol{a_1}$. If $(\boldsymbol{a_1}, \boldsymbol{a_2})$ has type τ in $\mathfrak{A}$ then there are at least $\lfloor k/m \rfloor$ distinct extensions of $\boldsymbol{a_1}$ which have type τ in $\mathfrak{A}$.*

Proof: This can be easily proved using the definition of extension axioms and observing the structure of k-types for each $\mathfrak{A} \in C_m$, in the proof of Lemma 1. □

Note that the restriction that $\boldsymbol{a_2} \not\subseteq \boldsymbol{a_1}$ is needed otherwise there is a unique extension of $\boldsymbol{a_1}$ to type τ.

A quantifier P can be seen as a collection $(P^n)_{n\in\omega}$, where P^n is the set of structures in P whose domain size is n. P^n can be viewed as an isomorphism closed class of structures which have domain of size n.

For our construction below, we need to give a definition of a partial quantifier on structures of domain size n. Following definition of this is along the expected lines. We consider structures equal upto isomorphism.

Definition 4 *A total quantifier P^n is a mapping from $\{\mathfrak{A} \mid \mathfrak{A}$ a σ structure, $||\mathfrak{A}|| = n\}$ to $\{true, false\}$. A partial quantifier P^n is a mapping from $\{\mathfrak{A} \mid \mathfrak{A}$ a σ structure, $||\mathfrak{A}|| = n\}$ to $\{true, false, *\}$. The $*$ can be thought of as 'undefined' element. The domain of a partial quantifier P^n is the pre-image of $\{true, false\}$ under P^n. A partial quantifier P_2^n extends a partial quantifier P_1^n if P_2^n restricted to the domain of P_1^n is the same as P_1^n.*

We will often use the following easy observation. Let $\phi(\boldsymbol{x})$ be a $L(P_1)$ formula for a logic L and generalized quantifier P_1. If evaluation of ϕ on structure $\mathfrak{A}$ does not entail evaluation of a subformula of ϕ beginning with P_1 on a σ structure which is outside the domain of P_1 then for all quantifiers P_2 which extend P_1, $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{x})[P_2/P_1]$ are equivalent over $\mathfrak{A}$.

Consider a structure $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$. We wish to define subsets of $|\mathfrak{A}|^k$ whose projection over $\mathfrak{B}_2$ is closed under the $\equiv^k$ relation over $\mathfrak{B}_2$.

Definition 5 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a structure. Let k be a positive integer and $[k] = \{1, \ldots, k\}$. Let $\pi \subseteq [k]$ with $|\pi| = r$. Let $E \subseteq |\mathfrak{B}_2|^{k-r}$ be an equivalence class of the $\equiv^k$ relation on $|\mathfrak{B}_2|^{k-r}$ and let $\boldsymbol{a} \in |\mathfrak{B}_1|^r$. We call $(E, \boldsymbol{a}, \pi)$ a* **k-triple over** *$(\mathfrak{B}_1, \mathfrak{B}_2)$. For such a triple $(E, \boldsymbol{a}, \pi)$ we define a set $S_{(E,\boldsymbol{a},\pi)} = \{\boldsymbol{b} \in |\mathfrak{A}|^k \mid$ projection of $\boldsymbol{b}$ on π is $\boldsymbol{a}$ and projection on $[k] - \pi$ is in $E\}$. Note that if $r = 0$, then we just have (E, π) as triple and $S_{(E,\pi)} \subseteq |\mathfrak{B}_2|^k$. On the other hand if $r = k$ then we have $(\boldsymbol{a}, \pi)$ as triple and $S_{(\boldsymbol{a},\pi)} \subseteq |\mathfrak{B}_1|^k$.*

A **set T is k-closed over $\mathfrak{B}_2$** *if it is a union of sets of the form S_e, where e ranges over some set of k-triples over $(\mathfrak{B}_1, \mathfrak{B}_2)$.*

Let $\phi_1(x_1, \ldots, x_k), \phi_2(x_1, \ldots, x_k), \phi_3(x_1, \ldots, x_k)$ be formulae (On a structure $\mathfrak{A}$ these can also be thought of subsets of $|\mathfrak{A}|^k$) and let α, β, γ be permutations on $\{1, \ldots, k\}$. We can associate with this data a formula ϕ as follows.

$$\phi(x_1, \ldots, x_k) = P\, x_{\alpha(1)} x_{\alpha(2)}, x_{\beta(1)}, x_{\gamma(1)}(\phi_1(x_{\alpha(1)}, \ldots, x_{\alpha(k)}), \\ , \phi_2(x_{\beta(1)}, \ldots, x_{\beta(k)}), \phi_3(x_{\gamma(1)}, \ldots, x_{\gamma(k)}))$$

For a ϕ as above and $(a_1, \ldots, a_k) \in |\mathfrak{A}|^k$, $\phi(a_1, \ldots, a_k)$ naturally gives rise to a structure over vocabulary σ of quantifier P. This structure is $(|\mathfrak{A}|, R, U, V)$, where $R = \{(b_1, b_2) \mid \mathfrak{A} \models \phi_1(b_1, b_2, a_{\alpha(3)}, \ldots, a_{\alpha(k)})\}$
$U = \{b \mid \mathfrak{A} \models \phi_2(b, a_{\beta(2)}, \ldots, a_{\beta(k)})\}$, $V = \{b \mid \mathfrak{A} \models \phi_3(b, a_{\gamma(2)}, \ldots, a_{\gamma(k)})\}$

We will call $(|\mathfrak{A}|, R, U, V)$ the structure associated with $\phi(a_1, \ldots, a_k)$. Note that $\mathfrak{A} \models \phi(a_1, \ldots, a_k)$ iff P is true on $(|\mathfrak{A}|, R, U, V)$.

Definition 6 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a Σ structure. We call two structures $\mathfrak{A}_1$ and $\mathfrak{A}_2$ over σ to be $(\mathfrak{B}_1, \mathfrak{B}_2)$ related if there are ϕ_1, ϕ_2, ϕ_3 defining k-closed sets over $\mathfrak{B}_2$, permutations α, β, γ and tuples $\boldsymbol{b_1}, \boldsymbol{b_2} \in S_{(E,\boldsymbol{a},\pi)}$, for some k-triple $(E, \boldsymbol{a}, \pi)$ over $(\mathfrak{B}_1, \mathfrak{B}_2)$ such that $\mathfrak{A}_1$ corresponds to $\phi(\boldsymbol{b_1})$ and $\mathfrak{A}_2$ corresponds to $\phi(\boldsymbol{b_2})$, where $\phi(\boldsymbol{x})$ is as defined above.*

It is easily seen that transitive closure of the above relation is an equivalence relation. We denote it by $EQ_{\mathfrak{B}_1,\mathfrak{B}_2}$ (we will drop the subscript when it is fixed by the context). By $EQ(\mathfrak{A}_1)$, we denote the equivalence class of this relation containing the structure $\mathfrak{A}_1$.

We define the quantifier on $\mathfrak{G}_j = \mathfrak{M}_{1,j}$, in j steps. At step i, $i \leq j$, we make sure that $IFP(P)$ formula ϕ_i, can 'access' only sets which are closed over $(\mathfrak{M}_{1,i}, \mathfrak{M}_{i+1,j})$. This will ensure that all inductions in ϕ_i are bounded in terms of $|\mathfrak{G}_i|$ only.

The idea of EQ classes is introduced because if we start with k-closed sets ϕ_1, ϕ_2, ϕ_3 and wish that ϕ constructed from generalized quantifier P using these sets also gives rise to k-closed sets, then P should be assigned the same value on all elements of an EQ class.

In order to extend the quantifier further, we need some information about the elements in the EQ class of a σ structure to know which structures have been included in the domain of the quantifier. Also, we require many unrelated structures with respect to relation EQ, so that including a structure in the domain of the quantifier still leaves enough structures outside the domain of the quantifier on which the value of the quantifier can be decided at later stages. We achieve this by requiring our structure $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ to satisfy the following property.

Definition 7 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a structure. We say that $\mathfrak{A}$ satisfies the k-extension property over $\mathfrak{B}_2$, if for each $\boldsymbol{b_1} \in |\mathfrak{A}|^i$ and $\boldsymbol{b_2} \in |\mathfrak{B}_2|^j$, $\boldsymbol{b_2} \not\subseteq \boldsymbol{b_1}$, $i, i+j \leq k$, there are at least $|\mathfrak{B}_1|+1$ distinct $\boldsymbol{c}$'s such that $(\boldsymbol{b_1}, \boldsymbol{b_2}) \equiv^k (\boldsymbol{b_1}, \boldsymbol{c})$ in $\mathfrak{A}$. In other words, if a k-type can be extended to another k-type by adding* **new** *elements then it can be extended to that type in at least $|\mathfrak{B}_1| + 1$ distinct ways.*

The following Lemma gives some information about the structures in $EQ(\mathfrak{D})$ in terms of the substructures they may possess. We first define notion of an isolated substructure and a unique substruture of a σ structure which is suitable for our purpose.

Definition 8 *Let $\mathfrak{M} = (|\mathfrak{M}|, R, U, V)$ be a σ structure. Let $S \subseteq |\mathfrak{M}|$, we call $(S, R \cap (S \times S), U \cap S, V \cap S)$ an isolated substructure of $\mathfrak{M}$, if $|S| \geq 2$, there are no R edges between S and $|\mathfrak{M}| - S$ and each element of S is connected to at least one another element of S. The size or cardinality of this substructure is $|S|$.*

Definition 9 *Let $\mathfrak{A} = (|\mathfrak{A}|, R, U, V)$ be a σ structure. A substructure $\mathfrak{A}_1 = (|\mathfrak{A}_1|, R_1, U_1, V_1)$ of $\mathfrak{A}$ is a unique substructure of $\mathfrak{A}$ if for any isolated substructure of $\mathfrak{A}$ isomorphic to $\mathfrak{A}'_1 = (|\mathfrak{A}_1|, R_1, U'_1, V'_1)$, we have $U_1 = U'_1$ and $V_1 = V'_1$. In other words, up to an isomorphism, $(|\mathfrak{A}_1|, R_1)$ has a unique extension to an isolated substructure of $\mathfrak{A}$.*

Lemma 3 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a structure such that $\mathfrak{A}$ satisfies the k-extension property over $\mathfrak{B}_2$. If a σ structure $(|\mathfrak{A}|, R, U, V)$ has a* unique *substructure $\mathfrak{H}$, $|\mathfrak{H}| \leq |\mathfrak{B}_1|$, then all structures $(|\mathfrak{A}|, R', U', V')$ in $EQ_{(\mathfrak{B}_1, \mathfrak{B}_2)}(|\mathfrak{A}|, R, U, V)$ have a* unique *substructure isomorphic to $\mathfrak{H}$.*

Proof: This nontrivial proof requires a careful analysis of substructures of two structures related by $EQ_{(\mathfrak{B}_1,\mathfrak{B}_2)}$ relation. Details are given in the full version. $\square$

Definition 10 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a structure and P a partial quantifier on $\mathfrak{A}$. We say that P is consistent over $(\mathfrak{B}_1, \mathfrak{B}_2)$, if for every EQ class τ over $(\mathfrak{B}_1, \mathfrak{B}_2)$ and every $\mathfrak{C}_1, \mathfrak{C}_2 \in \tau$ in the domain of P either P is true both on $\mathfrak{C}_1, \mathfrak{C}_2$ or false on both $\mathfrak{C}_1, \mathfrak{C}_2$.*

A consistent partial quantifier P is complete over $(\mathfrak{B}_1, \mathfrak{B}_2)$ if for each EQ class τ over $(\mathfrak{B}_1, \mathfrak{B}_2)$, P either includes the entire class τ in its domain or it includes no structure from τ in its domain.

Given a P consistent over $(\mathfrak{B}_1, \mathfrak{B}_2)$, its least $(\mathfrak{B}_1, \mathfrak{B}_2)$ complete extension P' is an extension of P with the smallest domain size which is (consistent and) complete over $(\mathfrak{B}_1, \mathfrak{B}_2)$. Note that it always exists.

Definition 11 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a structure and P a partial quantifier on $\mathfrak{A}$. We say* **that $L^k_{\infty,\omega}(P)$ is k-closed over** *$\mathfrak{B}_2$, if each $L^k_{\infty,\omega}(P)$ formula $\phi(\boldsymbol{x})$, whose evaluation on $\mathfrak{A}$ does not require a query to P outside its domain, defines a subset of $|\mathfrak{A}|^k$ which is k-closed over $\mathfrak{B}_2$. (Note that even if ϕ has less than k free variables, it can be thought of as defining a subset of $|\mathfrak{A}|^k$ by adding dummy free variables).*

Lemma 4 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ and let Q be a partial quantifier on $\mathfrak{A}$ which is consistent over $(\mathfrak{B}_1, \mathfrak{B}_2)$. Then $L^k_{\infty,\omega}(Q)$ is k-closed over $\mathfrak{B}_2$, in the sense of definition 11.*

Proof: This follows by an easy induction on $L^k_{\infty,\omega}(Q)$ formulae. $\square$

This Lemma confirms the intuition mentioned in the remarks after Definition 6.

Lemma 5 *Let $\mathfrak{A} = \mathfrak{B}_1 \oplus \mathfrak{B}_2$ be a rigid structure and let Q be a partial quantifier on $\mathfrak{A}$ such that $L^k_{\infty,\omega}(Q)$ is k-closed over $\mathfrak{B}_2$. Then for each $IFP(Q)$ formula ϕ, with at most k variables, whose evaluation on $\mathfrak{A}$ does not query Q outside its domain, there is a polynomial P_ϕ such that in its evaluation on $\mathfrak{A}$ subformulae beginning with Q need to be evaluated at most $P_\phi(|\mathfrak{B}_1|, t)$ times, where t is the number of k-types realized in $|\mathfrak{B}_2|$. Further, all IFP stages in ϕ define k-closed sets over $\mathfrak{B}_2$.*

Proof: In full version. $\square$

The point to note about the polynomial bound in the Lemma above is that it is polynomial in $|\mathfrak{B}_1|$ and not in $|\mathfrak{A}|$.

Finally, we are ready for the actual construction. As mentioned in the beginning of this section, we construct a class C of rigid structures, over the vocabulary $\Sigma = \{R\}$, consisting of one binary relation, and a quantifier P over σ.

The construction is by diagonalization over a fixed enumeration $(\phi_i)_{i\in\omega}$ of all $IFP(P)$ formulae. We construct a sequence of rigid connected graphs

$\mathfrak{M}_1, \mathfrak{M}_2, \ldots, \mathfrak{M}_i, \ldots$. Let $\mathfrak{G}_i = \mathfrak{M}_1 \oplus \ldots \oplus \mathfrak{M}_i$. The class C is defined as $\{\mathfrak{G}_n | n \geq 1\}$. In stage i, we come up with $\mathfrak{M}_i$ and define $P^{||\mathfrak{G}_i||}$. Each $a \in |\mathfrak{G}_i|$ has a distinct $L^2(P)$ type. The number of stages in each IFP iteration of ϕ_i is bounded by $(d.||\mathfrak{G}_i||^d + d)$, for a fixed constant d which may depend on ϕ_i alone, over the entire class C.

For any j, let $m_j \geq 3$ be such that all $\phi_1, \ldots, \phi_j$ have at most m_j variables and let $p_j(x, y)$ be $P_{\phi_1} + \ldots + P_{\phi_j}$, where P_{ϕ_l}, $l \leq j$, is as in Lemma 5. We will assume, without loss of generality, that all our bounding polynomials are monotone. Using Lemma 1, let t_m be the upper bound on the number of m types realized in structures of class C_m.

We now describe stage i in the construction of P, C. Let n be such that for all $m \geq n$, $2^m - m - m^2 > p_i(m, t_{m_i})$. Choose $\mathfrak{M}_i$ a rigid, connected graph of size $\geq n$ which satisfies $m_i + m_i.||\mathfrak{G}_{i-1}||$-extension axioms. This ensures the desired m_i-extension property by Lemma 2. Let the size of $|\mathfrak{M}_i|$ be n_i. Recall that we use the abbreviation $\mathfrak{M}_{i,j}$ for $\mathfrak{M}_i \oplus \ldots \oplus \mathfrak{M}_j$

Note that, it follows by our definition of isolated substructures and the sizes of various $\mathfrak{M}_i$'s, that for every $j \leq i$, $(\mathfrak{M}_{1,j}, U \cap |\mathfrak{M}_{1,j}|, V \cap |\mathfrak{M}_{1,j}|)$ is a unique substructure of $(\mathfrak{G}_i, U, V)$.

Let $\leq_{\mathfrak{M}_1}, \ldots, \leq_{\mathfrak{M}_i}$ be canonical orderings on $|\mathfrak{M}_1|, \ldots, |\mathfrak{M}_i|$ respectively (such orderings can be defined in the second level of the polynomial hierarchy).
We will define sets $T_{i,0} \subset T_{i,1} \subset \ldots \subset T_{i,i-1}$, such that $T_{i,0} = \emptyset$, $T_{i,j} \subseteq |\mathfrak{M}_{i,j}|$ and $|T_{i,j}| \geq |T_{i,j-1}| + 2$, $1 \leq j \leq i-1$. The quantifier $P^{||\mathfrak{G}_i||}$ will be defined as **true** on σ structures (of domain size $||\mathfrak{G}_i||$), which have a unique substructure as (at least) one of the structures, $(\mathfrak{M}_{i,j}, \{a\} \cup T_{i,j-1}, \{b\} \cup T_{i,j-1})$, where $a \leq_{\mathfrak{M}_j} b$ and $1 \leq j \leq i$. $P^{||\mathfrak{G}_i||}$ is **false** on other σ structures (of domain size $||\mathfrak{G}_i||$).

We claim that, for any such $T_{i,0}, T_{i,1}, \ldots, T_{i,i-1}$, any subset $S \subseteq |\mathfrak{M}_j|$ will be $L^2(P)$ definable. More precisely, for each S as above there is $L^2(P)$ formula $\phi_S(x)$ such that for all $a \in |\mathfrak{G}_i|$, $\mathfrak{G}_i \models \phi_S(a)$ iff $a \in S$. We prove this by induction on j.
Base case $j = 1$: Note that $L^2(P)$ formula
$\theta(x, y) = Pxy, y, x(R(x, y), y = x, x = y)$ when interpreted over $\mathfrak{G}_i$ defines a linear order on $|\mathfrak{M}_1|$ and does not relate any other elements of $\mathfrak{G}_i$. (The clever cross-wiring of variables in the above formula is from [2, Lemma 4.9, pp. 179]). Using this order we can define any $S \subseteq |\mathfrak{M}_1|$ by an $L^2(P)$ formula.
Induction step $j = k + 1$**:** Assume that the induction hypothesis holds for $j = k$, Therefore $T_{i,k}$ is $L^2(P)$ definable. Let $\psi(y)$ be the formula defining $T_{i,k}$.

Note that $L^2(P)$ formula
$\theta(x, y) = Pxy, y, x(R(x, y), y = x \vee \psi(y), x = y \vee \exists y[y = x \wedge \psi(y)])$, when interpreted over $\mathfrak{G}_i$ defines a linear order on $|\mathfrak{M}_{k+1}|$ and does not relate any other elements of $|\mathfrak{G}_i|$. Using this order we can define any $S \subseteq |\mathfrak{M}_{k+1}|$ by an $L^2(P)$ formula. This completes the induction.

Using the procedure below we define $T_{i,0}, T_{i,1}, \ldots, T_{i,i-1}$ with the properties mentioned above and define the quantifier $P^{||\mathfrak{G}_i||}$ in the following i steps.

$P_{i,0}^{||\mathfrak{G}_i||} = \emptyset$; $T_{i,0} = \emptyset$; (we drop the superscript in $P_{i,j}^{||\mathfrak{G}_i||}$ in the loop below)

For $j = 1$ **to** i **do**

1. Let $P_{i,j'}$ be the least such that, $P_{i,j'} \supseteq P_{i,j-1}$ and $P_{i,j'}$ is **true** on any σ structure which has a unique substructure $(\mathfrak{M}_{1,j}, \{a\} \cup T_{i,j-1}, \{b\} \cup T_{i,j-1})$ for some $a, b \in |\mathfrak{M}_j|, a \leq_{\mathfrak{M}_j} b$.
   Note that $P_{i,j'}$ is complete over $(\mathfrak{M}_{1,j}, \mathfrak{M}_{j+1,i})$, by Lemma 3.
2. Consider the evaluation of $\phi_j(P_{i,j'})$ on $\mathfrak{G}_i$, answering any query to $P_{i,j'}$ outside its domain as **false** and extending the domain of $P_{i,j'}$ to reflect this. Let $P_{i,j''}$ be the quantifier obtained after this evaluation.
3. Since $P_{i,j'}$ is complete over $(\mathfrak{M}_{1,j}, \mathfrak{M}_{j+1,i})$ and we have only added structures with same truth values to the domain of $P_{i,j'}$, quantifier $P_{i,j''}$ is consistent over $(\mathfrak{M}_{1,j}, \mathfrak{M}_{j+1,i})$. By Lemma 4, we get that every $L^k_{\infty,\omega}(P_{i,j''})$ formula is k-closed over $(\mathfrak{M}_{1,j}, \mathfrak{M}_{j+1,i})$. All IFP iterations in $\phi(P_{i,j''})$ terminate in the number of stages bounded by the number of m_j-**triples** over $(\mathfrak{M}_{1,j}, \mathfrak{M}_{j+1,i})$, which is bounded by $(2^{m_j}.|\mathfrak{M}_{1,j}|^{m_j}.t_{m_j})$.
4. By choice of $\mathfrak{M}_j$ and using Lemma 5, there is a $T \subseteq |\mathfrak{M}_j|$, $|T| \geq 2$, such that $P_{i,j''}$ is not queried on any σ structure which has a unique substructure $(\mathfrak{M}_{1,j}, T_{i,j-1} \cup T, T_{i,j-1} \cup T)$. We define $T_{i,j} = T_{i,j-1} \cup T$. Let $P_{i,j}$ be $P_{i,j''}$.

End For

Finally, $P^{||\mathfrak{G}_i||}$ is obtained by extending $P^{||\mathfrak{G}_i||}_{i,i''}$ so that on all inputs outside the domain of $P^{||\mathfrak{G}_i||}_{i,i''}$, $P^{||\mathfrak{G}_i||}$ is set to false. We define P as $\bigcup_{i\in\omega} P^{||\mathfrak{G}_i||}$.

To see that $IFP(P)$ is bounded on C, consider ϕ_j in the enumeration of $IFP(P)$ formulae. As shown above, for all i, all $IFP(P_{i,j''})$ iterations in ϕ_j on $\mathfrak{G}_i$ are bounded by $(2^{m_j}.|\mathfrak{M}_{1,j}|^{m_j}.t_{m_j})$. P extends $P_{i,j''}$, so by an observation before, $IFP(P_{i,j''})$ and $IFP(P)$ operate identically on $\mathfrak{G}_i$. Therefore $IFP(P)$ iterations on $\mathfrak{G}_i$ are also bounded by a constant independent of $\mathfrak{G}_i$.

We have also seen above that each element in $|\mathfrak{G}_i|$ has a distinct $L^2(P)$ type. Therefore the number of $L^2(P)$ types realized over structures in C is unbounded. Hence we have proved the following.

Theorem 2 *There is a quantifier P and a class C of rigid structures, such that $L^2(P)$-types realized over C are unbounded but $IFP(P)$ over C is bounded.*

Combining [2, Corollary 4.5, pp. 178] with Theorem 5, we immediately get the following corollary which shows that McColm's second conjecture can not be extended in the presence of generalized quantifiers.

Corollary 1 *There is a quantifier P and a class C of rigid structures, such that $IFP(P)$ is bounded on C but $L^2_{\infty,\omega}(P) \nsubseteq FO(P)$ over C.*

We can modify the construction presented above to make the quantifier in Theorem 2, PTIME computable. The idea is to pad each $\mathfrak{G}_i$ in that construction with a large clique. So each $\mathfrak{G}_i$ will now be replaced by $\mathfrak{H}_i = \mathfrak{M}_{1,i} \oplus \mathfrak{N}_i$, where $\mathfrak{N}_i$ is a large enough clique. As a result we do not get a class of rigid structures this time. The construction goes through with some easy changes. Due to lack of space we leave details of this to the full version and only note the statements of the results below.

Theorem 3 *There is a PTIME computable quantifier Q and a class D of finite structures, such that $L^2(Q)$-types realized over D are unbounded but $IFP(Q)$ over D is bounded. It follows that $L^2_{\infty,\omega}(Q) \not\subseteq FO(Q)$ over class D.*

Corollary 2 *[3] There is a PTIME computable quantifier Q such that $\equiv^{k,Q}$ is not $IFP(Q)$ definable over the class of finite structures.*

4 Ramified Conjectures with Generalized Quantifiers

By the standard techniques for obtaining normal forms for the fixed point logics (see ch. 7, [4]), it can be shown that $IFP(\mathbf{Q}) = \cup_k IFP^k(\mathbf{Q})$. The following generalization of Theorem 2.3 of [9] can be proved easily.

Lemma 6 *[9] Let k be a positive integer and let* $\mathbf{Q}$ *be a set of generalized quantifiers. Let $(\phi_1, \ldots, \phi_l)$ be a system of $L^k(\mathbf{Q})$ formulae. Then the following hold for $1 \leq i \leq l$.*

(i) Each component Φ_i^m of every stage $\Phi^m = (\Phi_1^m, \ldots, \Phi_l^m)$, $m \geq 0$, of the inflationary operator Φ is defined by an $L^k(\mathbf{Q})$ formula.

(ii) Each component Φ_i^∞ of the inflationary fixed point $(\Phi_1^\infty, \ldots, \Phi_l^\infty)$ of the system is $L^k_{\infty,\omega}(\mathbf{Q})$ -definable. As a result, we have that $IFP^k(\mathbf{Q}) \subseteq L^k_{\infty,\omega}(\mathbf{Q})$.

4.1 Relating Number of *k*-Types and *Uniform* Boundedness of LFP^k and IFP^k

The following lemma shows that, for any given $\mathfrak{A}$ there is a k-variable $IFP^k(\mathbf{Q})$ formula which can pass through as many distinct stages, during the computation of a fixed point, as the number of $L^k(\mathbf{Q})$-types in $\mathfrak{A}$.

Lemma 7 *Let σ be a vocabulary, k a positive integer and* $\mathbf{Q}$ *a finite set of generalized quantifiers. Let $\mathfrak{A}$ be a structure over σ realizing m, $L^k(\mathbf{Q})$ types. Then there is an $IFP^{k,1}(\mathbf{Q})$ system S such that $ch(S)(\mathfrak{A}) = m$.*

Proof: Easy. See full version. □

Corollary 3 *Let $\mathfrak{A}$ be a structure realizing m L^k types. Then there is an $LFP^{k,1}$ system S such that $ch(S)(\mathfrak{A}) = m$.*

Proof: Just observe that Y occurs only positively in the formula $\phi_{\mathfrak{A}}$ constructed in the proof of Lemma 7. □

Using Lemma 7 and Theorem 3, we get the following theorem.

Theorem 4 *There is a class D of structures and a PTIME computable quantifier Q such that for each $k \geq 2$, $IFP^k(Q)$ over D is bounded but not uniformly bounded.*

4.2 Second Conjecture

Theorem 5 *Let k be a positive integer, C a class of finite structures and $\mathbf{Q}$ a finite set of generalized quantifiers. Then the following are equivalent.*

1. $L^k(\mathbf{Q}) = L^k_{\infty,\omega}(\mathbf{Q})$ *over C.*
2. $IFP^k(\mathbf{Q})$ *is uniformly bounded on C.*
3. $IFP^{k,1}(\mathbf{Q})$ *is uniformly bounded on C.*

Proof: See full version. □

Using Corollary 3, we also get a slightly stronger version of a result in [9] as follows.

Corollary 4 *Let k be a positive integer and C a class of finite structures. Then the following are equivalent.*

1. $L^k = L^k_{\infty,\omega}$ *over C.*
2. LFP^k *is uniformly bounded on C.*
3. $LFP^{k,1}$ *is uniformly bounded on C.*

4.3 First Conjecture

Because the two notions of boundedness are different we have two versions of this conjecture. We disprove the version with uniform boundedness.

Theorem 6 *There is a PTIME computable quantifier Q and a class D of structures, such that $IFP^2(Q)$ is not uniformly bounded on D but for each $k \geq 2$, $IFP^k(Q) = L^k(Q)$ over D.*

Proof: Straightforward using boundedness of $IFP(Q)$ and Lemma 6. □

The other version of the first conjecture remains an open question.
Open: Let C be a class of finite structures and $\mathbf{Q}$ be a finite set of generalized quantifiers. Is $IFP^k(\mathbf{Q}) = L^k(\mathbf{Q})$ on C iff $IFP^k(\mathbf{Q})$ is bounded on C.

References

1. A. Dawar. *Feasible Computation through Model Theory.* PhD thesis, University of Pennsylvania, Philadelphia, 1993.
2. A. Dawar and L. Hella. The expressive power of finitely many generalized quantifiers. *Information and Computation*, 123(2):172–184, 1995.
3. A. Dawar, L. Hella, and A. Seth. Ordering finite variable types with generalized quantifiers. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.
4. H. Ebbinghaus and J. Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer, 1995.
5. Y. Gurevich, N. Immerman, and S. Shelah. Mccolm's conjectures. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 10–19, 1994.

6. Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.
7. L. Hella. Logical hierarchies in PTIME. *Information and Computation*, 129:1–19, 1996.
8. P. G. Kolaitis and M. Y. Vardi. Fixpoint logic vs. infinitary logic in finite model theory. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 46–57, 1992.
9. P. G. Kolaitis and M. Y. Vardi. On the expressive power of variable confined logics. In *Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 348–359, 1996.
10. G. L. McColm. When is arithmetic possible? *Annals of Pure and Applied Logic*, 50:29–51, 1990.

Separators Are as Simple as Cutsets
(Short Version)

Hong Shen[1]*, Keqin Li[2], and Si-Qing Zheng[3]

[1] School of Computing and Information Technology, Griffith University
Nathan, QLD 4111, Australia
hong@cit.gu.edu.au

[2] Department of Mathematics and Computer Science, State University of New York
New Paltz, NY 12561-2499, USA.
li@mcs.newpaltz.edu

[3] Department of Computer Science, University of Texas at Dallas
Richardson, TX 75083-0668, USA.
sizheng@utdallas.edu

Abstract. We show that all minimal a-b separators (vertex sets) disconnecting a pair of given non-adjacent vertices a and b in an undirected and connected graph with n vertices can be computed in $O(n^2 R_{ab})$ time, where R_{ab} is the number of minimal a-b separators. This result matches the known worst-case time complexity of its counterpart problem of computing all a-b cutsets (edge sets) [13] and solves an open problem posed in [11].

Keywords: Algorithms, complexity analysis, cutsets, data structures, separators.

1 Preliminaries

A separator S in a connected graph G is a subset of vertices whose removal separates G into at least two connected components. An a-b separator in G is a separator whose removal disconnects vertices a and b in G. That is, removal of S results that a and b reside in two different connected components of G [5]. An a-b separator is *minimal* if it does not contain any other a-b separator. When a and b are two vertex sets A and B in V respectively, an a-b separator becomes more general A-B separator that disconnects A and B.

Computing separators under various constraints [2,3,7] is closely related to determining (vertex) connectivity of a graph, which is a fundamental graph-theoretical problem with many important applications. The problem of computing (enumerating) all minimal a-b separators in a graph is not only of theoretical interest but also of significance in applications such as scheduling problems and network reliability analysis [1,4,7]. Due to its importance, this problem has been

* Author for correspondence. This work was done when the author was visiting Hong Kong Baptist University in 1998 and was partially supported by the Australia Research Council under its Small Grants Scheme.

studied by many researchers within various contexts [2,4,7,8,11]. The current best known result is due to Shen et al [11] that computes all R_{ab} minimal *a*-*b* separators in an n-vertex graph in $O(n^3 R_{ab})$ time[1]. It was posed as an open problem whether there is an algorithm that finds all minimal *a*-*b* separators in quadratic time in n per separator. That is, whether the complexity of finding *a*-*b* separators can match that for finding *a*-*b cutsets*, where an *a*-*b* cutset is an edge set whose removal disconnects a and b [3,11].

In this paper, we give a firm answer to the above open problem by presenting an efficient data structure, namely the *n-way bitwise ordered tree*, to maintain all distinct separators generated by level-to-level adjacent-vertex replacement.

First we present some notations and the level-by-level adjacent-vertex replacement approach proposed in [11] which we will also use later in this paper.

Let $G = (V, E)$ be an undirected connected simple graph. For any $X, Y \subset V$, let $X - Y = X \backslash Y = \{u \in X | u \notin Y\}$. For notational simplicity, sometimes we also use $X(\bar{Y})$ to denote $X - Y$. We denote the subgraph *induced* by the vertices of X by $G[X] = (X, E(X))$, where $E(X) = \{(u, v) \in E | u, v \in X\}$, and a neighbourhood set of X by $N(X) = \{w \in V - X | \exists v \in X, (v, w) \in E\}$.

Given an *a*-*b* separator S defined previously, we denote the connected components containing a and b in $G[V - S]$ by C_a and C_b respectively. For any $X \subset V$, We define the *isolated set* of X, denoted by $I(X)$, to be the set of vertices in X that have no adjacent vertices in C_b of $G[V - X]$ and hence are not connected to C_b.

Let $d(x)$ be the length of the shortest path (distance) from vertex $x \in V$ to a. For any $x \in V$, we define

$$N^+(x) = \{v | (x, v) \in E, v \in V \textit{ and } d(v) = d(x) + 1\}. \tag{1}$$

That is, $N^+(x)$ is a set of x's neighbours whose distance to a is greater than that of x precisely by 1.

Let vertices be assigned level numbers according to their distances from a. That is, a vertex is assigned level number i if it is i-edge distant from a, indicating that it is ranked at level i from a. If all vertices in separator S are at level i, this separator is said at level i and denoted by $S^{(i)}$. It has been shown in [11] that all (minimal) *a*-*b* separators can be generated by the so-called *level-by-level adjacent-vertex replacement* as follows:

Theorem 1. *Let L_i be the collection of all a-b separators at level i, where $1 \leq i \leq h$ and $h \leq n - 3$ is the maximal distance from a to any other vertex than b in G, and $L_0 = \{N(a) - I(N(a))\}$. The union of all L_i's, $\cup_{i=0}^{h} L_i$, contains all minimal a-b separators, if each separator $S^{(i)}$ in L_i is obtained by adjacent-vertex replacement on some $x \in S^{(i-1)}$ in L_{i-1} according to the following equation:*

$$S^{(i)} = (S^{(i-1)} \cup N^+(x)) - I(S^{(i-1)} \cup N^+(x)). \tag{2}$$

[1] A little more detailed analysis shows that their algorithm runs in $O(nmR_{ab})$ time for an m-edge G.

Clearly from each $S^{(i-1)}$ we can generate at most $|S^{(i-1)}|$ new minimal a-b separators in next level L_i. We say that separator $S^{(i-1)}$ *precedes* separator $S^{(i)}$, denoted by $S^{(i-1)} \prec S^{(i)}$, if $S^{(i)}$ is generated from $S^{(i-1)}$ by the above vertex replacement scheme.

Let $L_{-1} = \{a\}$. We can now construct a unique *expansion tree* by taking L_{-1} as the root and elements (nodes) of L_i as the nodes at level i and connecting a node $S^{(i-1)}$ at level $i-1$ to any node $S^{(i)}$ in level i if $S^{(i-1)} \prec S^{(i)}$, $0 \leq i \leq n-3$. Apparently this expansion tree contains all minimal a-b separators [11].

If we construct the above expansion tree simply level by level, we may end up with a huge-size tree with many duplicate notes because every node at level $i-1$ produces new nodes at level i independently. We therefore need to apply appropriate techniques to ensure that only distinct nodes are generated at each level of the expansion in order to result in a minimal-size tree. We denote such an expansion tree by $\mathcal{T}$.

Clearly $\mathcal{T}$ is constructed dynamically as level-by-level expansion proceeds. So a critical question is how we can incrementally maintain $\mathcal{T}$ for each insertion of a new node. In [11] the AVL tree is employed for this purpose, which requires $O(n \log |\mathcal{T}|)$ time. In this paper we propose a more efficient data structure to maintain the expansion tree that uses the binary representation of separators to guide fast n-way searching for duplicates. Our new data structure can insert a new separator with ordered nodes (with respect to indices) in $O(n)$ time.

In the next section we describe this new data structure and analyze its time and space complexity.

2 Maintaining the Expansion Tree

We use an n-way bitwise ordered tree to represent the (minimal) expansion tree $\mathcal{T}$ as follows:

Originally $\mathcal{T}$ contains only one node (root) $L_0 = S^{(0)} = N(a) - I(N(a))$ at level 0. When level-by-level expansion proceeds $\mathcal{T}$ is dynamically growing up to $h \leq n-3$ levels. Each node t_i at level i is an array of $n-i$ bits representing the last (greatest) $n-i$ vertices in lexicographical order in V: $B^{(i)}_{t_i}[0..n-i-1]$, $1 \leq t \leq |L_i|$. Clearly $|L_i| \leq \binom{n-2}{i}$ (the total number of ways choosing i elements from $V - \{a, b\}$). All edges in $\mathcal{T}$ are implicitly represented: An edge from $B^{(i-1)}_{t_{i-1}}[t_i]$ to $B^{(i)}_{t_i}[j]$ is represented by $B^{(i)}_{t_i}[j] = 1$ which indicates that x_j appears as the ith element with x_{t_i} being its precedent in at least one separator. $B^{(i)}_{t_i}[j] = 0$ otherwise. A separator of $s \leq n-2$ ordered elements, $S = \{x_0, x_1, \ldots, x_{s-1}\}$, is uniquely represented by a path from level 0 to level $s-1$: $B^{(0)}[x_0] \rightarrow B^{(1)}_{x_0}[x_1] \rightarrow \cdots \rightarrow B^{(s-1)}_{x_{s-2}}[x_{s-1}]$, where $B^{(0)}[x_0] = 1$ and $B^{(i)}_{x_{i-1}}[x_i] = 1$ for $1 \leq i \leq s-1$. Since S is lexicographically ordered, clearly $x_{i-1} \leq x_i \leq x_{i+1}$ for all i. Therefore $n-i$ bits of a node are sufficient to represent x_i at level i in $\mathcal{T}$. The algorithm for maintaining $\mathcal{T}$ works as follows: for each newly generated separator $S = \{x_0, x_1, \ldots, x_{s-1}\}$ search $\mathcal{T}$ level by level from level 0; if there is an i such that $B^{(i)}_{t_i}[x_i] = 0$ the separator is new and the algorithm will insert it at the ith level by assigning $B_{t_i}^{(i)}[x_i] = 1$; otherwise the separator is a duplicate and therefore should be discarded.

Figure 1 depicts the structure of our k-way bitwise ordered tree.

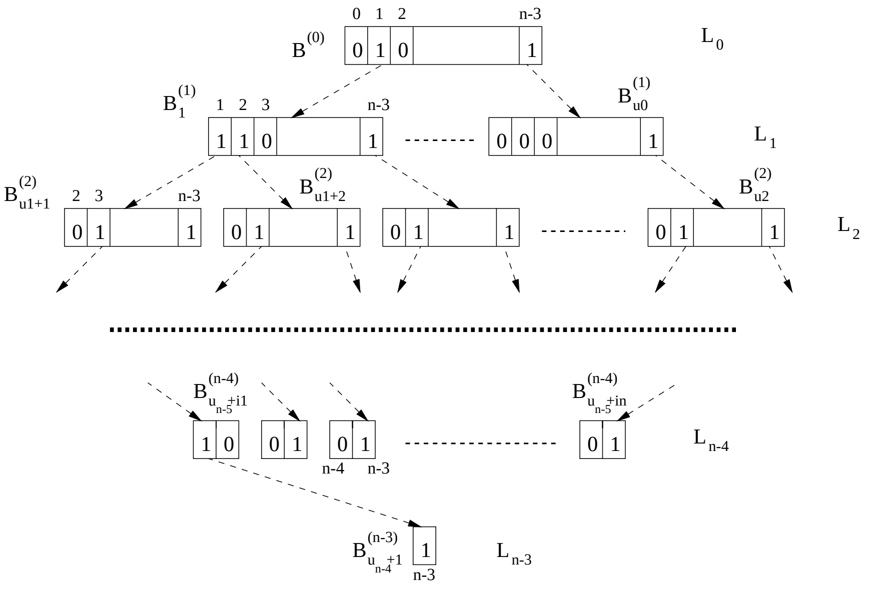


Fig. 1. Structure of the k-way bitwise ordered tree $\mathcal{T}$

Note that in fact $B_{t_i}^{(i)}$ requires only $n - t_i$ bits instead of $n - i$ because any element falling into $B_{t_i}^{(i)}$ has definitely a greater rank than t_i of its predecessor in the previous level due to the property that elements in the separator are ordered. This shows that we need at most half of the space allocated to each level, for instance, $\sum_{i=1}^{n-1} i$ rather than n^2 for L_1. However, we use here the maximal fixed length for bit arrays in each level for notational and analytical simplicity. We will use the precise figure later in the analysis of maximal space requirement of $\mathcal{T}$.

We construct the expansion tree through dynamically maintaining it for node (separator) insertions. Our algorithm for maintaining $\mathcal{T}$ with respect to insertion of a new node S is described as follows.

```
Procedure ExpansionTree($\mathcal{T}$, $S$)
  {*Insert ordered minimal $a$-$b$ separator $S = \{x_0, x_1, \ldots, x_{s-1}\}$ to expansion
  tree $\mathcal{T}$. $\mathcal{T}$ has $h \leq n-2$ levels: $L_0, L_1, \ldots, L_{h-1}$, where $L_0 = \{N(a) -$
  $I(N(a))\}$ and non-zero binary array $B_{t_i}^{(i)}[0..n-i-1]$ is a node in level $L_i$
  connected to the $t_i$th bit ("1") of some node $B_{t_{i-1}}^{(i-1)}[t_i]$ in $L_{i-1}$, $0 \leq t_i \leq n-i$,
  $1 \leq i \leq h$, $0 \leq s \leq n-2$.*}
```

```
i := 0;   t_0 := 0;
while $(B^{(i)}_{t_i} \in L_i)$AND$(B^{(i)}_{t_i}[x_i] = 1)$AND$(i \le s-1)$ do
        $t_{i+1} := x_i$;   $i := i + 1$;
if $i \le s-1$ then
    {*$S$ is new and should be inserted to $L_i$ of $\mathcal{T}$.*}
            for $j = i$ to $s-1$ do $B^{(j)}_{x_{j-1}}[x_j] := 1$
    {*Build a corresponding path $B^{(0)}_0[x_0] \to B^{(1)}_{x_0}[x_1] \to \cdots \to B^{(s-1)}_{x_{s-2}}[x_{s-1}]$ to
    represent $S$, where all the edges are implicit. If $B^{(j)}_{x_{j-1}} \notin L_j$, get a new cell
    ($n$ bits) and insert it to $L_j$.*}
        else discard $S$
    {*$S$ is a duplicate.*}
end.
```

It is not difficult to see that Algorithm `ExpansionTree` correctly maintains the expansion tree $\mathcal{T}$ that contains all distinct minimal a-b separators inserted to it (proof in the long version of the paper [12]. We now analyze the time and space complexities of the algorithm. Time complexity of the algorithm can be easily seen $O(s) = O(n)$. Space complexity analysis is little bit more complex. Due to the property of minimal separator, we know that if S and S' are two distinct minimal separators, then $S \not\subset S'$ and $S' \not\subset S$. Assume that a single word requires $\log n$ bits to store. We call the storage of $|S| \log n$ bits for separator S the *minimum ordinary storage*, as this is needed in ordinary word-by-word information storage. Clearly for $|S| = i$ there are $\binom{n-2}{i}$ separators corresponding to at most $\binom{n-2}{i}$ nodes at level L_i in $\mathcal{T}$. This indicates that the number of nodes in L_i of $\mathcal{T}$ is at most $\binom{n-2}{i}$. We use i-separator to denote a minimal a-b separator of size i. As we are mainly interested only in terms of magnitude, for simplicity in notation sometimes we use n and $n-2$ indistinguishably in our analysis.

Lemma 1. *Let M be the minimum ordinary storage for storing all minimal a-b separators. The space requirement of $\mathcal{T}$ is at most $\Theta(\frac{n}{\log n})M$ in the worst case and $\Theta(\frac{1}{\log n}M)$ in the best case.*

Proof. See the long version of the paper [12].

Another important measure we need to work out is the maximal space requirement of $\mathcal{T}$. Here we use the precise space allocation to nodes in $\mathcal{T}$, that is, $B^{(i)}_{t_i}$ requires only $n - t_i$ bits instead of $n - i$. We shall prove the following lemma:

Lemma 2. *The maximal space requirement of $\mathcal{T}$ is $O(2^n)$ bits for a maximal number of $O(2^n/\sqrt{n})$ minimal a-b separators, each containing $\frac{n-2}{2}$ elements. This space is effectively $\sqrt{n}$ bits per separator on average and therefore is significantly smaller than the ordinary storage which requires $n \log n/2$ bits per separator.*

See the long version of the paper [12].

Proof.

Lemma 2 shows an interesting property of our data structure $\mathcal{T}$ that it saves $\Theta(\sqrt{n}\log n)$ factor of the maximum space required by the ordinary storage for separators.

Now let us look at how likely the worst and best cases can occur respectively.

Lemma 3. *If the number of all minimal a-b separators is s, the worst case space complexity of $\mathcal{T}$ occurs at probability less than n^{-s+1}, whereas the best case space complexity occurs at probability greater than $n^{-\frac{s+n}{2}}$.*

Proof. First we have all elements in each separator ordered lexicographically. The probability of each case can be worked out by looking at the ways of distributing possible elements at the ith position of each separator among all bits node by node at level L_i and finding out the ratio of the number of ways forming the required distribution (worst and best) versus the number of all possible ways such that each node gets at least one element. It is easy to see that when the level-by-level distribution proceeds the probability in either case decreases because the above ratio for each node is always smaller than 1 and the probability is the product of all the ratios on the nodes to which elements are already distributed. Therefore the ratio of any two of them will remain about the same at L_0 and all levels in $\mathcal{T}$. Thus working out this ratio on the root at level L_0 is sufficient for simplicity.

The number of all possible ways of distributing all first elements in s separators to n bits of L_0 is the number of ways of distributing values 0 to $n-1$ to s elements with repetitions allowed: $U = n^s$.

We use $\#C(p_1+p_2+\ldots+p_i = s)$ to denote the number of i-part compositions of integer s. Represent integer s by a straight line of length s, with left endpoint 0 and right endpoint s, which is divided into s segments of unit length at $s-1$ intermediate points. Selecting i points from the $s-1$ intermediate points $(0 \leq i \leq s-1)$ results in $i+1$ intervals on the line which correspond to a composition of s containing $i+1$ parts (elements) [10]. This simple analogy shows that

$$\#C(p_1+p_2+\ldots+p_i = s) = \binom{s-1}{i-1}.$$

In the worst case all s first elements must be the same which can be any value between 0 and $n-1$. Therefore the number of ways of distributing them to one bit is

$$E^+ = \binom{n}{1}\#C(p_1 = s) = n\binom{s-1}{0} = n.$$

In the best case all s first elements must cover the whole range of values from 0 to $n-1$, which is only possible when $s \geq n$. In this case, the number of ways of distributing them to all n bits is

$$E^- = \binom{n}{n}\#C(p_1+p_2+\ldots+p_n = s) = \binom{s-1}{n-1}.$$

Since $s \leq \binom{n}{n/2} \approx 2^n/\sqrt{\pi n}$ by Stirling's formula, the proportion that $s \geq n$ covers the whole range of s is more than $\frac{2^n/\sqrt{\pi n}-n}{2^n/\sqrt{\pi n}} \approx 1$. So we can s imply regard that s covers the whole range. We approximate E^- to:

$$\begin{aligned} E^- &= \frac{(s-1)(s-2)\cdots n}{(s-n)(s-n-1)\cdots 1} \\ &= \frac{(s-1)}{(s-n)}\frac{(s-2)}{(s-n-1)}\cdots\frac{n}{1} \\ &\geq (\frac{s-1}{s-n}n)^{\frac{s-n}{2}} > n^{\frac{s-n}{2}} \end{aligned}$$

Thus we have that probabilities of the worst and best cases occurrence respectively:

$$Pr[E^+] = E^+/U = n^{-s+1} \tag{3}$$
$$Pr[E^-] = E^-/U \geq n^{-\frac{s+n}{2}}, \quad s \geq n. \tag{4}$$

From the above lemma we know that $Pr[E^-] > Pr[E^+]$ when $s \geq n$. Now let us see what their relationship is when $s < n$ by relaxing the the best case definition to that each node in $\mathcal{T}$ contains s "1" bits when $s < n$. In this case, $E^- = \binom{n}{s}\#C(p_1 + p_2 + \ldots + p_s = s) = \binom{n}{s}$. Taking the average value over $0 < s < n$, we have

$$E^- = \frac{1}{n-1}\sum_{s=1}^{n-1}\binom{n}{s} = \frac{2^n-2}{n-1},$$

and

$$U = \frac{1}{n-1}\sum_{s=1}^{n-1} n^s = \frac{n^n-n}{n-1}.$$

Hence

$$Pr[E^-] \approx (2/n)^n/(n-1)^2, \quad s < n. \tag{5}$$

This shows that $Pr[E^-] \gg Pr[E^+]$ also holds for $s < n$. Therefore we can conclude that the worst case occurs always at a probability much smaller than the best case.

We now find out the probability at which our $\mathcal{T}$ requires less space than the minimum ordinary storage M. Clearly $\mathcal{T}$ requires less space than M if every node in $\mathcal{T}$ on average carries more than $n/\log n$ "1" bits representing $n/\log n$ elements of some separators, because the minimum ordinary storage for $n/\log n$ is n bits which is the maximum size of a node in $\mathcal{T}$. Same as the analysis of the worst and best cases, for simplicity we only consider the first level L_0. We first compute the combined probability $Pr[\bar{E}]$ of L_0 containing i "1" bits for $i = 1$

or 2, ..., or $n/\log n$. Since L_0 (and each other node) must contain at least one "1" bit, the probability of L_0 containing more than $n/\log n$ "1" bits is simply $P[E] = 1 - Pr[\bar{E}]$. Let $s > n/\log n$, which is a necessary condition to enable $\mathcal{T}$ requiring less space than M, since otherwise L_0 can contain at most $s \leq n/\log n$ "1" bits. For notational simplicity, we assume that $s \geq n$. The same way of analysis also applies to the case of $s < n$.

The total number of ways of distributing s first elements to i bits in L_0 for $i = 1, 2, \ldots, n/\log n$ is

$$\begin{aligned}
\bar{E} &= \sum_{i=1}^{n/\log n} \binom{n}{i} \#C(p_1 + p_2 + \ldots + p_i = s) \\
&= \sum_{i=1}^{n/\log n} \binom{n}{i}\binom{s-1}{i-1} \\
&= \sum_{i=1}^{n/\log n} \frac{(n+1)}{(n-i+1)}\frac{(n+2)}{(n-i+2)} \cdots \frac{s}{(s-i)} \binom{n}{i}^2 \\
&\leq \frac{1}{\log n} \sum_{i=1}^{n'} (\frac{n+1}{n-i+1})^{s-n} \binom{n}{i}^2 \\
&= n^{s-n} \frac{1}{\log n} \binom{2n}{n} \approx \frac{2^{2n} n^{s-n}}{\sqrt{\pi n} \log n}.
\end{aligned}$$

The probability for the above event to occur thus is

$$Pr[\bar{E}] = \bar{E}/U \leq (4/n)^n / (\sqrt{\pi n} \log n) \tag{6}$$

Consequently, the probability that L_0 contains more than $n/\log n$ "1" bits, and hence $\mathcal{T}$ requires less space than M is

$$Pr[E] = 1 - Pr[\bar{E}] \geq 1 - (4/n)^n / (\sqrt{\pi n} \log n). \tag{7}$$

Apparently $Pr[E]$ is close to 1 as $Pr[\bar{E}] \ll 1$ and asymptotically 1 for large n and s. Hence we have

Lemma 4. *$\mathcal{T}$ requires less space than the ordinary storage $N \log n$ at an asymptotic 1 probability at least $1 - (4/n)^n / (\sqrt{\pi n} \log n)$, where N is the total number of elements in all s a-b separators drawn from $[0, n-1]$ and $s \geq n$.*

In the next section we show how to reduce the time complexity of other steps in level-by-level adjacent-vertex replacement by precomputing necessary data structures. Based on the n-way bitwise ordered tree, we then present an efficient algorithm that computes all minimal a-b separators at the cost of $O(n^2)$ per separator.

3 Computing All *a*-*b* Separators

Our algorithm for computing all minimal a-b separators computes separators level by level according to (2) starting from $S^0 = N(a) - I(N(a))$, where all distinct separators are correctly kept (and duplicates are henceforth deleted) by efficiently maintaining an minimal-size expansion tree $\mathcal{T}$ for inserting newly generated separators by algorithm `Expansion Tree`. In order to avoid repeating the same computation and hence improve the efficiency of the algorithm, we precompute necessary common data which will be used frequently in different steps. Typically, we need to precompute and store C_b, and, for every $x \in V$, $N^+(x)$, $C_b(N^+(x))$ and $I(N^+(x))$. Clearly, the space requirement for storing these data is $O(n^2)$ which is much less than that for $\mathcal{T}$.

We now present our algorithm as follows.

```
Procedure (a,b)-separators($G$, $a$, $b$, $\mathcal{T}$)
    {*Generate all distinct minimal $a$-$b$ separators for given non-adjacent vertices
    $a$ and $b$ in $G = (V, E)$, $|V| = n$. Input $G$, $a$ and $b$. Output $\mathcal{T} = \cup_{i=0}^{n-3} L_i$,
    where $L_i$ contains the nodes of the $i$th level in $\mathcal{T}$.*}
1       Compute the distance $d(v)$ from $a$ to every vertex $x \in V$ and label $v$ with
    $d(v)$;
2       Compute the connected component $C_b$ (containing $b$) of graph $G[V - \{a\}]$;
3       for each $x \in V$ do
  3.1       Compute $N^+(x) = \{v|(x, v \in E, d(v) = d(x) + 1\}$;
  3.2       Compute $C_b(N^+(x)) = C_b - N^+(x)$;
  3.3       Compute $I(N^+(x)) = N^+(x) - C_b$;
4       $L_{-1} := \{S\}$; $S := \{a\}$; $k := -1$; $\mathcal{T} := \emptyset$;
5       while $(k \leq n - 3) \wedge (C_b \neq \emptyset)$ do
          for each $S \in L_k$ do
  5.1           $I(S) := S - C_b$;
              for each $x \in S$ that is not adjacent to $b$ do
  5.2               $C_b(N^+(x) \cup S) := C_b(N^+(x)) - S$;
              if $C_b(N^+(x) \cup S) \neq \emptyset$ then
  5.3               $I(S \cup N^+(x)) := I(S) \cup I(N^+(x))$;
  5.4               $S' := (S \cup N^+(x)) - I(S \cup N^+(x))$;
    {*Generate a new separator $S'$ for the next level $L_{k+1}$.*}
  5.5               Sort $S'$ in lexicographical order using bucket sort;
  5.6               ExpansionTree($\mathcal{T}$, $S'$);
    {*Insert $S'$ to $\mathcal{T}$: insert $S'$ to $\mathcal{T}$ and also add it to $L_{k+1}$ if $S'$ is new (distinct);
    discard $S'$ otherwise (duplicate).*}
        $k := k + 1$
end.
```

The algorithm correctly computes all minimal a-b separators using level-by-level adjacent-vertex replacement as stated in Theorem 1, where each separator is generated by Equation (2). Moreover, all distinct separators generated are kept implicitly in $\mathcal{T}$ as paths from the root to leaves that connect the corresponding "1" bits across different levels, whereas all duplicates are discarded when inserting them to $\mathcal{T}$ by Step 5.5.

Figure 2 shows the process of generating all minimal *a*-*b* separators of a graph by the above algorithm and their representation in the expansion tree $\mathcal{T}$.

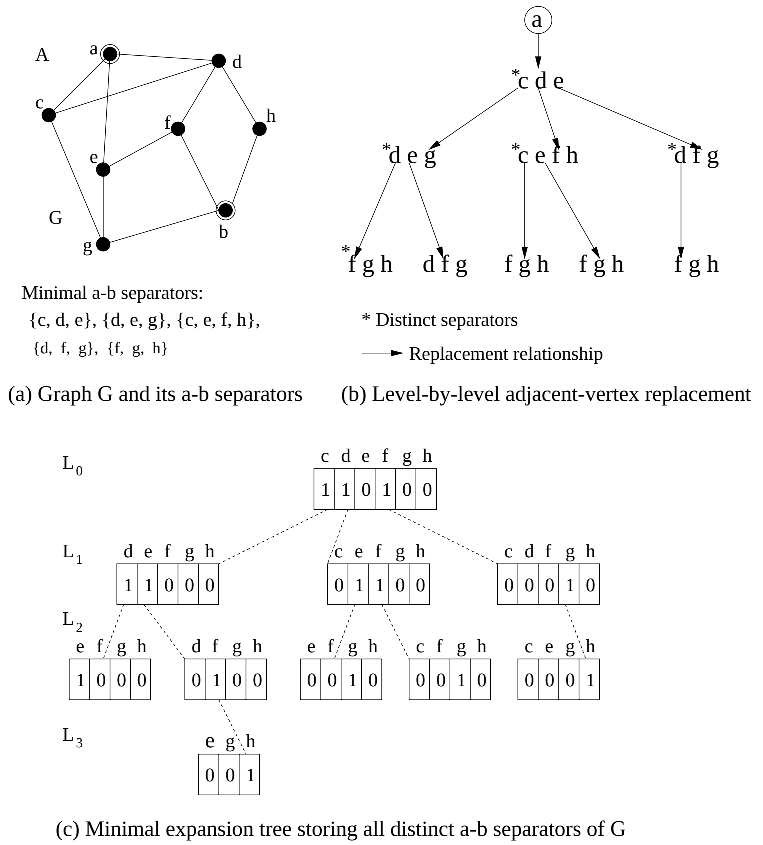


Fig. 2. Computing all minimal *a*-*b* separators and their representation in $\mathcal{T}$

Let us now analyze the time complexity of the algorithm: Step 1 can be done in $O(n^2)$ time by using Dijkstra's one-to-all shortest path

algorithm to construct a shortest path tree rooted at a and then labeling all vertices level by level in the tree. Step 2 computing the connected component C_b containing b in graph $G[V - \{a\}]$ can be done $O(n^2)$ time by first computing the connected components of $G[V - \{a\}]$, which takes time $O(|V| + |E|) = O(n^2)$, and then finding the one containing b in at most $O(n)$ time (there are at most $n - 1$ connected components of $G[V - N(a)]$). For each x, Steps 3.1, 3.2, and

3.3 each requires $O(n)$ time, therefore the whole Step 3 can be done in $O(n^2)$ time. Step 4 requires $O(n)$ time. For Step 5, the outmost loop is executed at most $n-2$ times ($|S| \le n-2$), and the next two nested loops are executed $\sum_{i=0}^{n-2} |L_i| = R_{ab}$ times, where $R_{ab} \le$ is the total number of distinct minimal a-b separators maintained in $\mathcal{T}$ since $\mathcal{T}$ does not contain any duplicate separators. Because $|S|, |C_b| \le n$, Step 5.1 requires $O(n)$ time. Moreover, since the size of each of the participating sets in the computation in Steps 5.2 – 5. 4 is at most n, Steps 5.2 – 5.4 all can be done in $O(n)$ time. Step 5.5 requires $O(n)$ time by well-known buck sort [9], and the same for Step 5.6 by Lemma 2. So the whole Step 5 can be done in $O(n^2 R_{ab})$. As shown in the previous section, the maximal number of minimal a-b separators is bounded by

$$R_{ab} \le \binom{n-2}{(n-2)/2} \approx 2^{n-2}/\sqrt{\pi(n-2)/2} < 2^{n-2}/\sqrt{n-2}. \qquad (8)$$

As a result we have the following theorem.

Theorem 2. *Given an n-vertex undirected graph, all minimal a-b separators for non-adjacent vertices a and b can be generated in $O(n^2 R_{ab})$ time, where $R_{ab} \le 2^{n-2}/\sqrt{n-2}$ is the number of minimal a-b separators.*

Replacing the single vertex a with set A and b with B in `(`a`,`b`)-separators` we can generalize the algorithm for the case that a and b are two non-adjacent vertex sets A and B in G. Noticing that $N^+(A)$ can be obtained in $O(|A|n)$ time, and that C_B in $O((n-|A|)^2)$ time by first computing C_B in $G[V-A]$ and then finding out which one (in case C_B consists of several connected components) contains all vertices of B. In Step 5 the dominating part of `(`a`,`b`)-separators`, clearly the outmost loop now needs to be executed only $n-|A|-|B|$ times, and the total number of iterations of the next two nested loops is equal to the number of all minimal A-B separators, R_{AB}. All steps inside the loops (5.1–5.4) need $O(n)$ time each. Therefore we have

Corollary 1. *Let A and B be two non-adjacent subsets of V in $G(V,E)$. We can generate all minimal A-B separators in $O(n(n-n_A-n_B)R_{AB})$ time, where $n_A = |A|$, $n_B = |B|$, $n = |V|$ and R_{AB} is the number of minimal A-B separators.*

The above corollary shows an improvement of $O(n)$ factor over the previous known result for computing A-B separators [11].

4 Concluding Remarks

The proposed algorithm can be easily generalized to the case that a and b are two non-adjacent vertex sets A and B in G. In this case for $n_A = |A|$, $n_B = |B|$, all minimal A-B separators can be computed in $O(n(n-n_A-n_B)R_{AB})$ time, where R_{AB} is the number of all minimal A-B separators.

Our algorithm can also be employed to compute all separators of a graph in the same way as [11] to improve the $O(n^5 R_\Sigma)$ time which was needed for computing R_{Σ} minimal separators of the graph. However, this is not necessary in most cases because all separators of G can be computed more efficiently by the following simple algorithm: One-by-one compute $\binom{n}{i}$ i-element combinations for $i = 1, 2 \ldots, n-2$, check whether each separates V, and output those separating V. Compute and output each combination chosen from n vertices require $O(n)$ time. For each such combination S, check whether it separates V can be done in $O(n^2)$ time by computing the connected components of $G[V - S]$. So the total time required is

$$T_{\Sigma} = O(n^2 \sum_{i=1}^{n-2} \binom{n}{i}) = O(n^2 2^n). \tag{9}$$

Hence we have

Theorem 3. *All separators in an n-vertex undirected graph can be computed in $O(n^2 2^n)$ time.*

By (8), since $R_{ab} < 2^{n-2}/\sqrt{n-2}$, above result shows that computing all a-b minimal separators is at least $\Theta(\sqrt{n})$ times more cheaper than computing all minimal separators.

References

1. H. Ariyoshi, Cut-set graph and systematic generation of separating sets, *IEEE Trans. Circuit Theory* **CT-19**(3) (1972) 233-240.
2. S. Arnberg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey, *BIT* **25** (1985) 2-23.
3. A. Gibbons, *Algorithmic Graph Theory*, Cambridge Univ. Press, 1985.
4. L.A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge Univ. Press, 1993.
5. M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
6. D.S. Hirschberg, A.K. Chandra and D.V. Sarwate, Computing connected components on parallel computers, *Commu. ACM* **22**(8) (1979) 461-464.
7. A. Kanevsky, On the number of minimum size separating vertex sets in a graph and how to find all of them, *Proc. 1st Ann. ACM-SIAM Symp. Discrete Algorithms* (1990) 411-421.
8. T. Kloks and D. Kratsh, Finding all minimal separators of a graph, *Proc. Theoretical Aspects of Computer Sci.* LNCS 775 (1994) 759-767.
9. D.E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching*, Addison-Wesley, 1973.
10. E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs (1977).
11. H. Shen and W. Liang, *Efficient enumeration of all minimal separators in a graph*, *Theoretical Computer Science* **180** (1997) 169-180.
12. H. Shen, K. Li and S.Q. Zheng, *Separators are as simple as cutsets* (long version), manuscript.
13. S. Tsukiyama, I. Shirakawa and H. Ozaki, An algorithm to enumerate all cutsets of a graph in linear time per cutset, *J. ACM* **27**(4) (1980) 619-632.

Ant Colony Optimization for the Ship Berthing Problem

Chia Jim Tong, Hoong Chuin Lau, and Andrew Lim

School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119260
chia@acm.org, lauhc@comp.nus.edu.sg, alim@comp.nus.edu.sg

Abstract. Ant Colony Optimization (ACO) is a paradigm that employs a set of cooperating agents to solve functions or obtain good solutions for combinatorial optimization problems. It has previously been applied to the TSP and QAP with encouraging results that demonstrate its potential. In this paper, we present FF-AS-SBP, an algorithm that applies ACO to the ship berthing problem (SBP), a generalization of the dynamic storage allocation problem (DSA), which is NP-complete. FF-AS-SBP is compared against a randomized first-fit algorithm. Experimental results suggest that ACO can be applied effectively to find good solutions for SBPs, with mean costs of solutions obtained in the experiment on difficult (compact) cases ranging from 0% to 17% of optimum. By distributing the agents over multiple processors, applying local search methods, optimizing numerical parameters and varying the basic algorithm, performance could be further improved.

1 Ant Colony Optimization

The Ant Colony Optimization (ACO) paradigm was introduced in [1], [2] and [3] by Dorigo, Maniezzo and Colorni. ACO has been applied effectively to the traveling salesman problem (TSP) [4] and the quadratic assignment problem (QAP) [5], among several other problems. The basic idea of ACO is inspired by the way ants explore their environment in search of a food source, wherein the basic action of each ant is: to deposit a trail of pheromone (a kind of chemical) on the ground as it moves, and to probabilistically prefer moving in directions with high concentrations of pheromone deposit.

As an ant moves, the pheromone it leaves on the ground marks the path that it takes. Another ant that passes by later can detect the pheromone and decide to follow the trail with high probability. If it does follow the trail, it leaves its own pheromone on it, thus reinforcing the existing pheromone deposit. By this mechanism, the movement of ants along a path between the nest and the food reinforces the pheromone deposit on it, and this in turn encourages further traffic along the path. This behavior characterized by positive feedback is described as autocatalytic.

On the other hand, ants may take a direction other than the one with the highest pheromone concentration. In this way, an ant does not always have to travel on the path most traveled. If an ant takes a path less traveled that deviates slightly from a popular path, and also happens to be better (shorter) than other popular paths, the pheromone it deposits encourages other ants to also take this new path. Since this path is shorter, the rate of pheromone deposit per ant that travels on it is higher, as an ant traverses a shorter distance in one trip. In this way, positive feedback can occur on this path and it can start to attract ants from other paths.

By the interplay of these two mechanisms, better and better paths emerge as the exploration proceeds. For the purpose of designing an algorithm based on this idea drawn from nature, an analogy can be made of: 1) real ants vs. artificial agents, 2) ants' spatial environment vs. space of feasible solutions, 3) goodness of a given path vs. objective function of a given solution, 4) desirability of taking a particular direction vs. desirability of making a certain decision in constructing the solution, 5) real pheromone at different parts of the environment vs. artificial pheromone for different solution choices. One of the main ideas behind ACO algorithms is how relatively simple agents can, without explicit communication, cooperate to solve a problem by indirect communication through distributed memory implemented as pheromone.

In this paper, we study how ACO can be applied effectively to the ship berthing problem (SBP), through the FF-AS-SBP algorithm, an application of ACO to the SBP. The focus of this study is not on the SBP itself or on fine-tuning our algorithm for maximum performance. Rather, it is on demonstrating that ACO can be applied effectively to the SBP. In Section 2, we formally describe the SBP. In Section 3, we describe a candidate solution representation, from which we adapt an indirect, first-fit (FF), solution approach in Section 4 so that it becomes more suitable for the complex nature of the SBP. In this section, we also describe a randomized FF algorithm and the basis of FF-AS-SBP. FF-AS-SBP is described in Section 5. By naming the algorithm FF-AS-SBP, we acknowledge that there could be many other ACO algorithms for the SBP. In Section 6, we describe the experiment and report and interpret the results, comparing FF-AS-SBP against the randomized FF algorithm. In this section, we also discuss how results could be further improved, how the algorithm lends itself to parallelization, and possible future work. Finally, Section 7 is the conclusion.

2 The Ship Berthing Problem

This problem, which has been studied in [6] and [7], can be defined as follows: ships ($\mathcal{S} = \{S_i: i = 1, 2, \ldots, n\}$) are specified to have lengths l_i, arrive at a port at specified times t_i and stay at the port for specified durations d_i. Each ship that arrives is to be berthed along a wharf line of length L, i. e., it is placed at the interval $(b_i, b_i + l_i)$ along the wharf line. Once berthed, its location is fixed for the entire duration of its stay. Also, each ship has a minimum inter-ship clearance distance c_i^s and a minimum end-berth clearance distance c_i^b . Four types of constraints apply:

– Ships can only be berthed within the extend of the wharf line. No part of any ship can extend beyond the beginning or the end of the wharf line. More strongly, the distance from either end of a ship to either end of the wharf line cannot be less than the minimum end-berth clearance distance.

$$\forall i \in \{1, 2, \ldots, n\} \quad c_i^b \leq b_i \leq L - l_i - c_i^b$$

– No two ships can share the same space along the wharf line if the time intervals in which they are berthed intersect. More strongly, the end-to-end distance between them cannot be less than the minimum inter-ship clearance of either one of them.

$$\begin{aligned}
&\forall i, j \in \{1, 2, \ldots, n\} \\
&\quad (t_i, t_i + d_i) \cap (t_j, t_j + d_j) \neq \emptyset \\
&\rightarrow \left(b_i - \max\left\{c_i^s, c_j^s\right\}, b_i + l_i + \max\left\{c_i^s, c_j^s\right\}\right) \cap (b_j, b_j + l_j) = \emptyset
\end{aligned}$$

– A ship may be given a fixed berthing location (b_i is fixed for some values of i).
– A ship may be prohibited from berthing in certain intervals of the wharf line. More precisely, the interval bounded by the location of the two ends of a ship after it has been berthed cannot intersect with any of the prohibited interval.

$$(b_i, b_i + l_i) \cap (p, q) = \emptyset \qquad \begin{array}{l}\text{if constraint applies to } S_i\text{, where} \\ (p, q) \text{ is some forbidden interval}\end{array}$$

The minimization version of the problem is to determine L_o the minimum length of the wharf line needed to berth all the ships subject to the given constraints.

The decision version of the problem is to determine whether berthing is possible, given a fixed value of L.

The density D is defined as the maximum total length of berthed ships at any one time [1]:

$$D = \max_{t \in (-\infty, +\infty)} \left(\sum_{i \in \{i : t_i \leq t < t_i + d_i\}} l_i \right)$$

It is easy to see that D is a tight lower bound on L.

In this paper, we also define a measure F, which we call the fragmentation, defined as:

$$F = 1 - \frac{\sum d_i l_i}{\left(\max(t_i + d_i) - \min(t_i)\right) D}$$

The berthing scenario can be visualized as a 2-D plane where the x-axis represents time, the y-axis represents space (along the wharf line), and each ship

[1] For convienience, this definition ignores minimum end-berth and inter-ship clearance

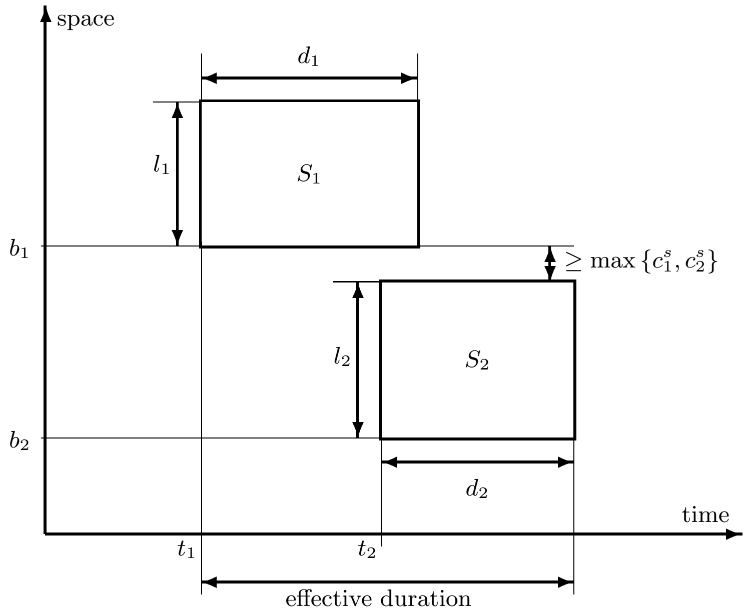


Fig. 1. A Sample Berthing Scenario

corresponds to a rectangular block whose x-extent is the time extent and the y-extent is the space extent of the ship. Figure 1 provides an example of this setup. Given an optimum solution of cost D, F is the percentage total area of regions not covered by a block, within the effective duration of the problem.

By having c_i^s and c_i^b set to zero and removing the last two constraints, the SBP becomes the dynamic storage allocation problem (DSA), which is known to be NP-complete and for which there is an 5-approximation algorithm (with a cost upper-bound of five times optimum)[8]. Since SBP is a generalization of DSA, it is also NP-complete. [7] provides a reduction of the NP-complete partition problem to the SBP, which is another way to show that SBP is NP-complete.

While simple to understand, a direct, geometric, representation for the SBP does not lend itself to an effective solution strategy. For this reason, a graph representation for the SBP was proposed in [7], together with an algorithm that uses this representation for finding good solutions.

In the new representation, each vertex v_i corresponds to the ship S_i and has weight l_i. Two distinct vertices v_i and v_j are connected by an (undirected) edge of weight $\max\left\{c_i^s, c_j^s\right\}$ iff $(t_i, t_i + d_i)$ and $(t_j, t_j + d_j)$ intersect. There are also zero-weight vertices v_l and v_r, and *arcs* $\langle v_l, v_i\rangle$ and $\langle v_i, v_r\rangle$ of weights c_i^b for each vertex v_i that corresponds to a ship S_i. The constraints related to prohibited and fixed berthing positions can be represented using a combination of auxilliary vertices

$\{v_{n+1}, v_{n+2}, \ldots\}$ acting as imaginary ships, auxilliary edges and auxilliary arcs. Details of this representation can be found in the original paper [7].

A feasible solution is any DAG, G, resulting from setting the direction of all edges, i. e., converting each edge to an arc of either direction. The cost of the solution, cost(G), is equal to both the length of the longest path in it and the value of L corresponding to that solution. The objective, therefore, is to find a DAG with as short a longest path as possible.

3 Solution as a Vertex Permutation

Not all edge direction assignments lead to a feasible solution as some lead to circuits, which do not exist in a DAG. Rather than searching all $2^{|E|}$ possible digraphs, many of which may not be DAGs, we can map each possible DAG to a vertex permutation and perform the search over the space of possible permutations. Each permutation $\pi = \begin{pmatrix} 1 & 2 & \cdots & n \\ \pi_1 & \pi_2 & & \pi_n \end{pmatrix} = (\pi_1 \pi_2 \ldots \pi_n)$ maps each vertex i to an integer label $\pi_i \in \{1, 2, ..., n\}$, where $n = |V|$. An edge $\langle u, v \rangle$ is set to arc $\langle u, v \rangle$ iff $\pi_u < \pi_v$ and $\langle v, u \rangle$ iff $\pi_v < \pi_u$. In this way, the digraph induced by a permutation is always a DAG, and each DAG has at least one corresponding permutation. A given permutation can always be normalized w. r. t. a given graph by first computing the DAG it induces and then performing a deterministic DAG labeling to obtain the normalized permutation.

In this paper, we use the terms 'labeling' and 'position' both to mean 'permutation'. 'Position' carries the meaning that vertices can be ordered in a sequence s. t. the first vertex in the sequence has label 1, the second vertex in the sequence has label 2 and so on, and the position of a vertex, which is equivalent to its labeling, is its position in that sequence.

The permutation representation or solutions seems to lend itself to a direct solution strategy similar to that used in [5], where the solution is also represented as a permutation. However, there is one important difference that makes the permutation representation problematic for the SBP. In the QAP, individual labels contribute piecewise additively to the final objective function. In the SBP, individual labels alone do not determine the cost of the longest path. This value is a function of the collective interaction between the vertex positions within the graph and there is no known straightforward relationship between the cost and the labeling over a subset of the vertices— over all possible DAGs, the cost of the longest path is not predicted or determined by individual vertex labels, or even arc directions, as the following two examples illustrate: The mere flip of a single arc can drastically change the cost; the reverse of a DAG G (all arcs are flipped; π_i is swapped with π_{n-i+1}) has the same cost as G.

Therefore, both the graph representation and the permutation representation seem unlikely to support ACO algorithms. For this reason, we adapted from the permutation representation a more indirect solution approach, which is explained in the next section.

4 First-Fit Approaches

In the standard first-fit algorithm (FF), ships are berthed (packed) one at a time in some predetermined order. When a ship S_i is to be packed, it is position as near to the front as possible without overlap with other ships S_j which have been berthed and whose time intervals $(t_j, t_j + d_j)$ intersect with the time interval of S_i, $(t_i, t_i + d_i)$. Of course, the packing of each ship is also done so that none of the other problem constraints are violated.

This can visualized on the geometric representation as positioning each block (ship), one at a time, with the x-coordinate fixed, minimizing the y-coordinate while not letting the current block overlap (or come too close) with over blocks which have been packed, or any other constraint to be violated.

When all the ships have been packed, the cost of the solution can be easily determined from the y-coordinate, length and minimum end-berth clearance of each ship.

The input the the FF algorithm, therefore, are the SBP problem itself and the order in which to pack the ships, represented as a permutation π, where ship i is the ith ship to be packed. It should be clear that for a given SBP problem, some permutations yield better solutions than others. In fact, it can be shown that there always exists a permutation that yields an optimum solution. However, for a given problem, it is difficult to perform a quick analysis to obtain an optimum, or even a good permutation. A straightforward approach, therefore, is to actually try different permutations.

4.1 Randomized First-Fit Algorithm

This algorithm simply generates a random permutation and calls FF with it. This is repeated for as many times as required to obtain better and better solutions. Improvements become increasing rare as the cost the the best discovered solution approaches the optimum.

4.2 ACO-First-Fit Algorithm

Using an ACO approach, we would like to obtain good permutations for doing FF.

Each permutation can be assigned a cost, which is the cost of the solution obtained by calling FF with it.

Unlike in DAG labeling, permutations could be related to cost in a straightforward way. In other words, for a given problem, there could be some easily represented, hidden, rules by which FF packing could be ordered so as to obtain a reasonable solution. For example, the complex nature of SBP seems not as strong in this representation, as can be seen in that in general, a permutation does not have the same cost as its reverse, and that displacing one ship in the packing sequence does not drastically change the cost, as are the case in DAG labeling.

Intuitively, it is the relative order in which packing is done that affects the cost, rather than the absolute position of each ship in the packing sequence—we are concerned with whether "S_i is packed before S_j" rather than whether "S_i is the jth ship to be packed". To support this mode of representation, given a permutation π of length n, we define a $n \times n$ boolean, lower-triangular matrix (only entries strictly below the diagonal are defined):

$$\mathbf{R} = \left(r_{ij} = \begin{cases} \text{true if } \pi_i < \pi_j \\ \text{false if } \pi_i > \pi_j \end{cases} \right)$$

It can be easily seen that there is a one-to-one function that maps π to $\mathbf{R}$, and that the reverse is not true, i. e., some values of $\mathbf{R}$ are invalid. A permutation π can be constructed from constraints encoded in (a valid) $\mathbf{R}$ as follows:

1. Set $\pi_1 = 1$
2. Determine (from the 2nd row of $\mathbf{R}$) whether $\pi_2 < \pi_1$ (then $\pi_2 = 1$ and π_1 should be shifted right, i. e., set to 2) or $\pi_2 > \pi_1$ (then $\pi_2 = 2$).
3. Determine (from the 3rd row of $\mathbf{R}$) whether $\pi_3 = 1, 2$ or 3 and set it to that value. $\forall i \neq 3$ s. t. $\pi_i \geq \pi_3$, increment π_i by 1.
4. In the same fashion determine and update π_i for $i \in \{4, \ldots, n\}$.

This above definition provides the basis for the pheromone design of FF-AS-SBP.

5 Ant Colony Optimization for the SBP

A vital factor in the effectiveness of any ACO algorithm is the design of the pheromone trail data structure—the information contained therein and how it is used. The FF-AS-SBP algorithm presented below uses the design we have experimentally found to give the best performance.

The algorithms in this section are not presented in a computationally optimized form, or the form in which we implemented them, but only to describe their computational effects.

The FF-AS-SBP algorithm takes as input the problem in the graph representation and the density D of the problem, which is used as a lower bound. It stores solutions as FF permutations and returns the cost of the best permutation found. The cost, L_π of a permutation π is defined as the cost of the packing obtained from performing FF with π. The parameters to the algorithm are α, N, $K$and γ. α and γ are real numbers in the interval $[0, 1)$. N and K are positive integers.

The non-scalar variables used are $\mathcal{B}$ and $\mathbf{T}$. $\mathcal{B}$ is a set of permutations — the set of the best K solutions discovered. $\mathbf{T} = (\tau_{ij})$ is a $n \times n$ matrix of real values each representing the intensity of a pheromone trail. τ_{ij} represents the desirability of setting $\pi_i < \pi_j$, in the spirit of matrix $\mathbf{R}$ of Section 4.2 (unlike in $\mathbf{R}$, the matrix is not lower-triangular but diagonal elements are irrelevant).

FF-AS-SBP Algorithm

```
1. Let $\tau_0 = (nD)^{-1}$. Let $L_{\min}$ denote $\min_{\pi \in \mathcal{B}} L_\pi$.
2. $\mathcal{B} \leftarrow \emptyset$;   $\mathbf{T} \leftarrow \tau_0$
3. Populate $\mathcal{B}$ with a random permutation for $2K$ times
4. Delete from $\mathcal{B}$ all but the best $K$ permutations
5. $\mathbf{T} \leftarrow (1-\alpha)\mathbf{T}$
6. For each $\pi \in \mathcal{B}$, update the pheromone matrix $\mathbf{T}$ according to the Global
   Update Algorithm using a reinforcement value of $L_\pi^{-1}$
7. For each ant $a = 1, 2, \ldots, N$, build a solution permutation according to
   the Ant Solution Algorithm and add the solution to $\mathcal{B}$ while keeping
   $|\mathcal{B}| \leq K$ by deleting worst permutations as necessary
8. If $L_{\min} > D$ and neither the iteration limit nor time limit has been exceeded
   then goto 5
9. Return $L_{\min}$
```

Steps 1–4 perform initialization. Steps 5–6 perform global pheromone update using the K best solutions. Step 7 performs ant exploration and local pheromone update.

We now explain the role of each input parameter. α represents the rate of pheromone evaporation and determines conversely, the persistence of memory from earlier iterations. N is the number of ants used in step 7 for constructing ant solutions. K is the number of globally best solutions to remember for the purpose of the global update in step 6. γ represents the preference for exploitation over exploration when ants construct their solutions in the Ant Solution Algorithm.

We now describe the sub-algorithms in detail.

Global Update Algorithm

This simple algorithm reinforces pheromone trails associated with a given solution π by a given reinforcement value Δ.

```
For each $u \in \{1, 2, \ldots, n-1\}$
  For each $v \in \{u+1, \ldots, n\}$
    If $\pi_u < \pi_v$
      $\tau_{uv} \leftarrow \tau_{uv} + \Delta$
    else
      $\tau_{vu} \leftarrow \tau_{vu} + \Delta$
```

Ant Solution Algorithm

This is the algorithm that each ant performs in building a solution π. The solution (permutation) is incrementally constructed by setting π_i in order, starting on π_1 and ending on π_n, similar to the method described in Section 4.2. The difference is that at each step, instead of being determined by a boolean matrix of constraints, each π_i is set to a value probabilistically chosen from all possible values based on their desirability.

At each step, the desirability measure of each candidate value, a function of different pheromone strengths, is evaluated and either exploitation or exploration is chosen probabilistically. Exploitation is chosen with probability γ. In exploitation, the candidate value with the highest desirability is chosen. In exploration, these values are probabilistically chosen based on their desirability. We now define the desirability measure d_p of a given value p when choosing a value for π_c.

d_p is defined as

$$\prod_{v \in \{1,2,\ldots,c\}} \begin{cases} \tau_{vc} \text{ if } \pi_v < p \\ \tau_{cv} \text{ otherwise} \end{cases}$$

With d computed, exploitation chooses the value with highest d, while exploration chooses each candidate value p with probability proportionate to d_p. After a value p has been chosen and π has been updated accordingly, all pheromone trails τ_{ij} associated with the p, i. e., that appear as terms in the product in the above equation, are diminished:

$$\tau_{ij} \leftarrow (1-\alpha)\tau_{ij} + \alpha\tau_0$$

The above is known as the local (pheromone) update.

6 Experiment

In this section we discuss the design, execution and results of our experiment to study the performance of the FF-AS-SBP algorithm, as well as our interpretation of the results and possible future work.

For simplicity in the design of the experiments, we considered only problem instances with zero edge weights. Also, the constraints related to clearance distances and fixed and forbidden positions are absent from the test cases. In other words, only problems that are also DSA problems are used. Nevertheless, the results should extend to the general case.

While we have not mapped out the characteristics of the parameter space as it is too vast, we have empirically found that setting $\alpha = 0.1$, $N = n$, $K = \lceil 0.1m \rceil$ ($m = |E|$) and $\gamma = 0.9$ seems to yield reasonably good solutions. This is the parameter setting adopted in all the test runs recorded in this section.

Each of the 6 test cases consist of one connected component–i. e., there is no way to cut the geometric representation into two parts with a vertical line without also cutting a block. This is because a problem of more than one component can be divided into these components to be solved individually.

The test cases were generated by a block cutting algorithm that generates blocks of varying height and width from an initial rectangular block, which is recursively divided into smaller and smaller blocks (some intermediate blocks are L-shaped blocks). Hence, all the problems generated are compact, i. e., have zero fragmentation, a property that is expected to make them difficult to solve optimally.

The randomized FF algorithm and FF-AS-SBP are run 10 times on each case, and each run is given $3|V|$ seconds to live. For each algorithm applied to each of the 6 test cases, the costs from each of the 10 runs are reported in ascending

Table 1. Randomized First-Fit Algorithm Results

Case	D	$\lvert V\rvert$	$\lvert E\rvert$	Runs									
1	80	20	119	80	80	80	80	80	80	80	80	80	80
2	120	30	338	120	120	120	121	121	121	125	125	126	126
3	160	40	457	175	186	187	187	188	189	190	190	190	193
4	200	50	633	245	246	247	248	249	251	251	252	254	254
5	240	60	783	298	300	300	301	301	305	306	307	308	312
6	320	80	1641	416	417	417	419	420	422	424	425	425	427

Table 2. FF-AS-SBP Results

Case	D	$\lvert V\rvert$	$\lvert E\rvert$	Runs									
1	80	20	119	80	80	80	80	80	80	80	80	80	80
2	120	30	338	120	120	120	120	120	120	120	120	121	121
3	160	40	457	173	173	175	176	176	177	177	182	182	184
4	200	50	633	205	225	231	232	233	235	238	239	244	247
5	240	60	783	254	254	255	258	258	258	258	258	258	258
6	320	80	1641	368	369	369	369	370	375	376	379	384	387

order. These results are shown in Table 1 and Table 2 and summarized in Table 3.

The randomized FF algorithm acts as a control for checking whether pheromone information does make a real difference in the quality of obtained solutions. For this reason, time-to-live of the runs is expressed in terms of actual time taken rather than number of iterations.

FF-AS-SBP performed consistently and significantly better than randomized FF, especially for larger cases, except in case 1, where both algorithms always returned an optimum solution.

Our interpretation of the experiment results is that pheromone information did help to improve the quality of the solution when applied to an FF-based approach to the SBP. We also note that it was necessary for us to adopt this indirect approach to avoid building an algorithm that would futilely explore the rough terrain of the more direct approach based on DAG labeling.

Table 3. Experiment Summary

				Randomized FF			FF-AS-SBP		
Case	D	$\lvert V\rvert$	$\lvert E\rvert$	Min	Mean	Max	Min	Mean	Max
1	80	20	119	80	80	80	80	80	80
2	120	30	338	120	122.5	126	120	120.2	121
3	160	40	457	175	187.5	193	173	177.5	184
4	200	50	633	245	249.7	254	205	232.9	247
5	240	60	783	298	303.8	312	254	256.9	258
6	320	80	1641	416	421.2	427	368	374.6	387

In view of the current results, the following areas merit further investigation:

- Heuristic measure for (partial) FF permutations while they are being constructed. An obvious choice is the cost of the partial solution.
- Alternative pheromone matrix design.
- The full characteristics of the parameter space in affecting the dynamics of the ant exploration and pheromone matrix. This could provide insight on how to optimize convergence and quality of solution, by setting parameters appropriately, or even dynamically varying them during the search through the use of appropriately designed meta-heuristics.
- Employing heterogeneous ants in the search. Ants could differ by quantitative parameters or qualitatively by the search strategy they use. The diversity introduced by having heterogeneous ants could contribute to overall algorithm performance. If investigation demonstrates that employing heterogeneous ants is indeed a profitable approach, the result could extend to other fields of distributed computing and lead to interesting investigation on the use of heterogeneous agents in those fields as well.
- How performance can be improved by increasing the number of ants. If increasing the number of ants can significantly improve performance, then there exists the possibility of distributing the work of many ants to multiple processors to achieve significant speedup, since individual ants operate independently of one another.
- How local search techniques can be combined with the basic ACO technique to improve performance. An example of such a technique is the famous tabu search method[9].

7 Conclusions

In this paper, we have formulated a pheromone information design and coupled it with the FF algorithm to produce an ACO algorithm that obtains reasonably good solutions for the SBP, thus demonstrating that the ACO paradigm can be applied effectively to the SBP. Morever, we have demonstrated that pheromone structure does not need to be directly related to the physical structure with which a target problem naturally is expressed.

The major ingredients of our algorithm, FF-AS-SBP, are the FF algorithm, permutation construction and the pheromone information that supports this construction. FF-AS-SBP compares favorably against the randomized FF algorithm.

We have also proposed a few key areas for further investigation to obtain even better performance and deeper insight. Of special interest is the possibility of using heterogeneous agents in FF-AS-SBP and in distributed computing in general.

8 Acknowledgement

This research was supported in part by the NUS Research Grant RP3972679.

References

[1] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: An autocatalytic optimizing process. Technical Report 91-016 Revised, Dipartimento Elettronica e Informazione, Politecnico di Milano, Italy, 1991.

[2] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, B(26):29–41, 1996.

[3] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992. In Italian.

[4] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learing approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[5] L. M. Gambardella, È. D. Taillard, and M. Dorigo. Ant colony for the QAP. Technical Report IDSIA 97-4, IDSIA, Lugano, Switzerland, 1997.

[6] K. Heng, C. Khoong, and A. Lim. A forward algorithm strategy for large scale resource allocation. In *First Asia-Pacific DSI Conference*, pages 109–117, 1996.

[7] Andrew Lim. An effective ship berthing algorithm. In *IJCAI '99*, volume 1, pages 594–605, 1999.

[8] Jordan Gergov. Approximation algorithms for dynamic storage allocation. In *ESA '96: Fourth Annual European Symposium*, pages 52–61, 1996.

[9] F. Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

Security Modeling and Evaluation for the Mobile Code Paradigm

Anthony H.W. Chan and Michael R. Lyu

Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong
{hwchan1, lyu}@cse.cuhk.edu.hk

There is no well-know model for mobile agent security. One of the few attempts so far is given by [1]. The model is, however, a qualitative model that does not have direct numerical measures. It would be great if there is a quantitative model that can give user an intuitive sense of "how secure an agent is".

Software reliability modeling is a successful attempt to give quantitative measures of software systems. In the broadest sense, security is one of the aspects of reliability. A system is likely to be more reliable if it is more secure. One of the pioneering efforts to integrate security and reliability is [2]. In this paper, these similarities between security and reliability were observed.

Security	**Reliability**
Vulnerabilities	Faults
Breach	Failure
Fail upon attack effort spent	Fail upon usage time elapsed

Fig. 1. Analogy between Reliability and Security

Thus, we have *security function, effort to next breach distribution,* and *security hazard rate* like the *reliability function, time to next failure distribution,* and *reliability hazard rate* respectively as in reliability theory. One of the works to fit system security into a mathematical model is [3], which presents an experiment to model the attacker behavior. The results show that during the "standard attack phase", assuming breaches are independent and stochastically identical, the period of working time of a single attacker between successive breaches is found to be exponentially distributed.

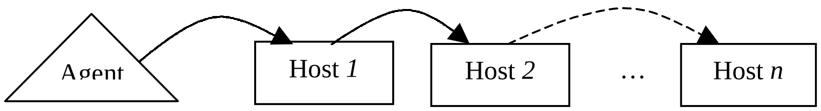


Fig. 2. A Mobile Agent Travelling on a Network

Now, let us consider a mobile agent travelling through *n* hosts on the network, as illustrated in Figure 2. Each host, and the agent itself, is modeled as an abstract machine as in [1]. We consider only the standard attack phase described in [3] by malicious hosts. On arrival at a malicious host, the mobile agent is subject to an attack effort from the host. Because the host is modeled as a machine, it is reasonable to estimate the attack effort by the number of instructions for the attack to carry out, which would be linearly increasing with time. On arrival at a non-malicious host, the effort would be constant zero. Let the agent arrive at host *i* at time T_i, for $i = 1, 2, \ldots, n$. Then the effort of host *i* at total time *t* would be described by the *time-to-effort function*:

$$E_i(t) = k_I(t-T_i) \text{ , where k is a constant}$$

We may call the constant *k* the *coefficient of malice*. The larger the k_i, the more malicious host *i* is ($k_i = 0$ if host *i* is non-malicious). Furthermore, let the agent stay on host *i* for an amount of time t_i, then there would be breach to the agent if and only if the following breach condition holds:

$$
\begin{array}{ll}
 & E_i(t_i+T_i) > \text{effort to next breach by host } i \\
\text{i.e.,} & k_it_i \quad > \text{effort to next breach by host } i
\end{array}
$$

As seen from [32], it is reasonable to assume exponential distribution of the effort to next breach, so we have the *probability of breach at host i*,

$$
\begin{aligned}
\text{P(breach at host } i) &= \text{P(breach at time } t_i+T_I) \\
&= \text{P(breach at effort } k_it_i) \\
&= 1 - exp(-vk_it_i) && ,v \text{ is a constant} \\
&= 1 - exp(-\lambda_it_i) && ,\lambda_i = vk_i
\end{aligned}
$$

We may call *v* the *coefficient of vulnerability* of the agent. The higher the *v*, the higher is the probability of breach to the agent. Therefore, the *agent security E* would be the probability of no breach at all hosts, i.e.,

$$E = \prod_{i=1}^{n} e^{-\lambda_i t_i} = e^{-\sum_{i=1}^{n} \lambda_i t_i}$$

Suppose that we can estimate the coefficients of malice k_i's for hosts based on trust records of hosts, and also estimate the coefficient of vulnerability *v* of the agent based on testing and experiments, then we can calculate the desired time limits T_i's to achieve a certain level of security *E*. Conversely, if users specify some task must be carried out on a particular host for a fixed period of time, we can calculate the agent security *E* for the users based on the coefficients of malice and vulnerability estimates.

References

1. Fritz Hohl. “A Model of Attacks of Malicious Hosts Against Mobile Agents”. In *Fourth Workshop on Mobile Object Systems (MOS’98): Secure Internet Mobile Computation*, 1998.
2. Sarah Brocklehurst, Bev Littlewood, Tomas Olovsson and Erland Jonsson. “On Measurement of Operational Security”. In Proceedings of the Ninth Conference on Computer Assurance (COMPASS’94): Safety, Reliability, Fault Tolerance and Real Time, Security, p.257-266.
3. Erland Jonsson. “A Quantitative Model of the Security Intrusion Process Based on Attacker Behavior”. In *IEEE Transactions on Software Engineering, Vol. 23, No. 4*. IEEE, April 1997.

CASA - Structured Design of a Specification Language for Intelligent Agents

Stephan Flake and Christian Geiger

C-LAB VIS (Visual Interactive Systems), Paderborn, Germany
{flake,chris}@c-lab.de, http://www.c-lab.de/vis

1 Introduction

Agent based technologies in the sense of distributed computing becomes increasingly relevant in academic and industrial research and development. Our multi agent system CASA focuses on the specification of complex plans that describe the behavior of agents. The design of CASA is based on concepts from concurrent logic programming that were used to extend the well known BDI agent approach AgentSpeak(L) [2]. Rao's AgentSpeak(L) is a specification language similar to horn clauses and can be viewed as an abstraction of an implemented BDI system. Our work is based on the assumption that AgentSpeak(L) demonstrates a successful reengineering approach of an implemented multi agent system that is now given a formal specification. By extending this specification with new features for complex plans it was possible to derive an efficient implementation that supports these new features.

2 Design of the CASA Specification Language

Specification: CASA agents perceive events/messages and select *relevant plans* with individual weights for handling these events/messages. Plans are modeled as clauses with additional guard predicates for testing the applicability of the clause (*applicable plans*). Such guard predicates may be arbitrary complex, i.e., the reduction of a guard may include the evaluation of another plan (deep guards). Such complex speculative computations may also include communicative acts with other agents. Based on the contextual conditions in guard predicates different plan types can be distinguished: *reactive plans* only have simple tests as guards, *deliberative plans* allow speculative computations within an agent and *communicative plans* allow to communicate with other agents. CASA uses a hierarchy for plan selection (reactive $>$ deliberative $>$ communicative). An agent can execute several plans at a time and elements of a single plan can be processed sequentially or in parallel. Additionally, plans can be suspended by other plans and have a special exception section if an applicable executed plan fails. The features of a CASA agent are formally defined based on extended guarded horn clauses and the cycle of operation is specified by an abstract interpreter. For ease of use we developed a simple textual format that allows an efficient modeling.
Modeling: The textual CASA definition format defines the initial agent state and is divided into four sections: First, functions for selecting events, plans, and intentions at run time execution have to be declared. Initial goals of the agent are declared in the

second section. Each of these goals will be instantiated together with an applicable plan. For parallel plan execution a multistack data structure is used to handle the instantiated plans (named as intentions) as separate entities. Initial facts and plans are listed in the third and fourth section.

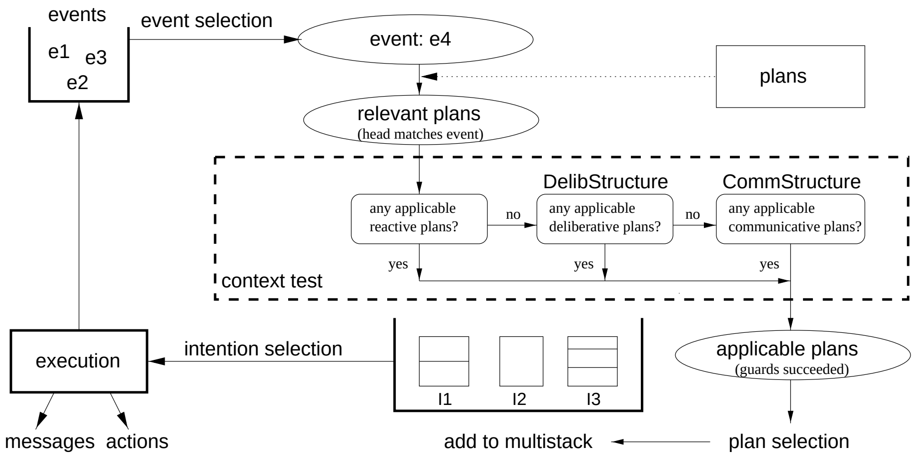


Fig. 1. CASA Interpreter Cycle

Handling speculative computations is one of the major aspects of the abstract CASA interpreter. Speculative computations appear in the context test for relevant plans whenever guards of deliberative or communicative plans have to be checked. Two independent additional components are introduced in order to manage speculative computations: the *DelibStructure* (resp. *CommStructure*) is holding elements composed of a goal and all relevant deliberative (resp. communicative) plans to check. For each of these elements a new instance of the interpreter is generated and executed in parallel to the other cycles. The operational semantics of the CASA interpreter are best described by means of the interpreter cycle shown in Figure 1.

Implementation: CASA is implemented with JDK 1.1.8, integrating modules of M. Huber's JAM library. A parser written in JavaCC reads the textual CASA specification, sets the initial state of CASA agents and starts the execution on the CASA interpreter. CASA agents are integrated into the MECCA framework, an agent management system that implements the FIPA ACL standard.

Validation: As a first case study we presented a simple application taken from holonic manufacturing [1]. Future work will concentrate on the development of visual tools for the design of CASA agents and the application in the area of flexible manufacturing systems and intelligent user interfaces.

References

1. S. Flake, Ch. Geiger, G. Lehrenfeld, W. Mueller, V. Paelke. Agent-Based Modeling for Holonic Manufacturing Systems with Fuzzy Control. *NAFIPS'99*, New York, 1999.
2. A.S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. *7th European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

An Efficient Location Management by Optimal Location Area Partitioning in PCS Networks

Joon-Min Gil and Chong-Sun Hwang

Dept. of Computer Science and Engineering, Korea Univ.
1, 5-Ga, Anam-Dong, SeongBuk-Gu, Seoul 136-701, KOREA
{jmgil, hwang}@disys.korea.ac.kr

1 Introduction

In PCS (personal communication systems) networks, location management deals with the problem of tracking down a mobile user. It involves two kinds of activities: one is location updating and the other is paging [1]. Each cost for location updating and paging is significantly associated with each other and therefore, has a trade-off relationship to total signaling cost. Location management strategies, as implemented in current PCS networks and as presented in this literature, are based on spatially static location areas that are built by heuristics or aggregate statistics [1,2]. The static location areas are used identically for all mobile users, even though the mobility pattern and call arrival rate of each mobile user are different spatially and temporally. Thus, the location management applied to same location areas for all mobile users suffers from the various mobile properties of users. Consequently, these strategies can not efficiently reduce the total signaling cost.

In this paper, we propose a location area partitioning strategy that can partition the whole cells into optimal location area. Our strategy considers per-user mobility pattern and call arrival rates on the zone-based approach. Also, the paging and location updating cost is simultaneously considered in determining the location area due to the intimate correlation on the total signaling cost.

2 Proposed Strategy

The proposed strategy consists of the following three steps:
1) Collect individual mobile user's mobility pattern and call arrival rate. Update user profile for every day or every week: We make an attempt to determine the location areas on per-user basis, *personal location area*. The information to create the personal location areas is derived from the user profile, P that contains a record of previous transitions from cell to cell and the number of calls requested in given time interval. P is updated for every or week, using the information accumulated whenever the user responds to a paging message, originates a call, or sends a update message.
2) Create the mobility graph from user profile: In order to represent and maintain the use profile efficiently, we introduce *mobility graph*, $G = (V, E)$,

where V is set of vertices and E is the set of edges. In G, each vertex has the weight that represents the average call arrival rate. The weight assigned to each vertex is used as the paging cost. The edge between two vertices has the weight which represent the average updating rate when a user moves across cells. The weight assigned to the edge is used as location updating cost. The total signaling cost per mobile user based on the mobility graph is following as: $C_T(m, \mu_a, U_m) = m \cdot \mu_a \cdot C_p + U_m \cdot C_u$, where m is the number of cells in a location area, u_a is the call arrival rate, and U_m is the location updating rate. C_p and C_u are the paging cost per cell and location updating cost, respectively.
3) Partition the vertices of the mobility graph into optimal location areas by genetic algorithms: We consider the determination of location areas as a class of partitioning problems: a partition of N_v vertices into N_{LA} location areas. Since location area partitioning problem is a well-known NP-complete problem, an exact search for optimal location areas in given time is impractical. Genetic algorithms is used for finding near-optimum location location areas. In mobility graph, the active vertex represents the cells which a user visited a lease once or in which a user received more than one incoming call. Only active vertices are encoded into individuals. Structural crossover and structural mutation operator are used as genetic operators.

In our simulation, we use mobility data collected for ten days from a mobile user in a hexagonal cell environment. We assume that the mobile user moves through six intervals, in which different mobility pattern (velocity and direction) and call arrival rate are used. After 1000 generations, the produced number of location areas is 7, 5, 4, and 3 when $1/\gamma$ (the ratio of C_u to C_p) is 2, 3, 5, and 10, respectively. It is shown that our strategy has the capability of optimizing the number of location areas as well as the location area size according to the mobility pattern and call arrival rate of a mobile host. Also, our strategy has smaller cost than zone-based strategy with the fixed location area, regardless of the variance of the $1/\gamma$.

3 Conclusions

In this paper, we proposed a location area partitioning strategy using genetic algorithms for mobile location tracking under the fact that the size and shape of location areas affect the total signaling cost.

References

1. Pitoura E. and Samaras G.: Data Management for Mobile Computing. Kluwer Academic, (1998)
2. Plassman D.: Location management strategy for mobile cellular networks of 3rd genetation. Proc. of IEEE Vehicular Technology Conf. (1995) 649-653

Programming with Explicit Regions

Koji Kagawa

R.I.S.E., Kagawa University
1-1 Saiwai-cho, Takamatsu 760-8526, JAPAN
kagawa@eng.kagawa-u.ac.jp

Most functional languages such as ML and Haskell, and many object-oriented languages such as Smalltalk and Java use garbage collection to reclaim memories automatically. Since programmers are freed from worry about memory management, they can enjoy more safety and productivity than in conventional languages such as C. On the other hand, garbage collection causes several difficulties in real-time systems and very small systems, since it is difficult for us to predict execution time of particular parts of programs and since garbage collection requires certain amount of memories in order to be performed efficiently.

Tofte and Talpin's region and effect system for ML [1] is an alternative to run-time garbage collection. It *infers* lifetime of objects at compile time and makes it possible to reclaim memories safely without using garbage collectors at run time. Then, objects can be allocated in a stack (to be exact, in a stack of regions). The system uses an ordinary ML program as input, and annotates it with *region* operators such as `letregion` and `at`. Programmers do not have to provide any explicit directives about lifetime of objects. It is sometimes necessary, however, for programmers to transform programs specially in order to make the system infer as they intend. Therefore, programmers have to know, to some extent, the region inference rules.

However, rather than a system which infers all about regions automatically, sometimes it might be convenient to have a semi-automatic system which lets programmers delimit lifetime of objects directly and checks safety of such directives. This is what we want to propose here. In principle, such a direct style would be possible in Tofte and Talpin's system by allowing programmers to use region annotations explicitly. In practice, however, it would be difficult to do so since region annotations tend to be lengthy and complex. Therefore, we want to make region annotations simple and readable.

Our proposal is also based on Launchbury and Peyton Jones's proposal of `runST` [2] for Haskell, which encapsulate *stateful computations* in pure computations. "State" in Launchbury and Peyton Jones's terminology roughly corresponds to "region" in Tofte and Talpin's. Though Haskell is a lazy language and therefore memory management is not a target of their research, it seems possible to apply their proposal to memory management in eager languages such as ML. In fact, Launchbury and Peyton Jones's `runST` and Tofte and Talpin's `letregion` seem to be closely related. They both use polymorphism to delimit regions and effects. In Launchbury and Peyton Jones's proposal, by using the *monad of state transformers* to manipulate state, state is not explicitly handled in programs, while types show information about state and the extent of state is given explicitly by the operator `runST`.

In our system, in most cases, programmers do not have to specify regions. Then, the current default region which is passed around implicitly is used. However, sometimes we want to access older regions instead of the current region. In such cases, programmers should be able to explicitly specify the region. *Compositional references*, which the author proposed in [3] in order to manipulate general mutable data structures in Haskell, are used for this purpose. Our region actually corresponds to a set of regions of Tofte and Talpin's. A newer region contains older regions and there is a kind of a subtyping relation between old and new regions. This fact is helpful to keep type expressions simple. When our region primitive named `extendST` allocates a new region, it passes to the inner expression a reference to the old default region. This is contrary to `letregion`, which passes to the expression a reference to the newly created region.

One of the important points of Tofte and Talpin's system is region inference of recursive functions [4]. Recursive functions must be region polymorphic in order for the system to be practical. In general, however, ML-style type inference algorithm cannot infer the most general type for polymorphic recursion. Tofte and Talpin's system treats region parameters in a special way to deal with polymorphic recursion and uses iteration to find an appropriate region assignment. We will present an alternative way of inferring regions of recursive functions, which does not use iteration.

Acknowledgment

This research is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Encouragement of Young Scientists, 10780196, 1999.

References

1. Mads Tofte and Jean-Pierre Talpin. Implementation of Typed Call-by-Value λ-calculus using a Stack of Regions. In *21st ACM Symposium on Principles of Programming Languages,* pages 188–201, January 1994.
2. John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation,* 8(4):293–341, 1995
3. Koji Kagawa. Compositional References for Stateful Functional Programming. *Proc. of the International Conference on Functional Programming 1997.* June 1997.
4. Mads Tofte and Lars Birkedal. A Region Inference Algorithm. *ACM Transactions on Programming Languages and Systems,* 20(5):724–767, July 1998.

A Verification Technique Based on Syntactic Action Refinement in a TCSP-like Process Algebra and the Hennessy-Milner-Logic

Mila Majster-Cederbaum and Frank Salger

Universität Mannheim
Fakultät für Mathematik und Informatik,
D7, 27, 68131 Mannheim, Germany
{mcb, fsalger}@pi2.informatik.uni-mannheim.de

Abstract. We investigate the conceptual link between action refinement for a process algebra which contains the TCSP-parallel composition operator and recursion on the one hand and action refinement for the Hennessy-Milner-Logic on the other. We show that the assertion $P \models \varphi \Leftrightarrow P[a \rightsquigarrow Q] \models \varphi[a \rightsquigarrow Q]$ where $\cdot[a \rightsquigarrow Q]$ denotes the refinement operator both on process terms and formulas holds in the considered framework under weak and reasonable restrictions.

TCSP (see e.g. [BHR84]) can be used to develop reactive systems in a formal way. This process calculus provides operators which are used to compose process expressions to more complex systems. The basic building blocks in such calculi are uninterpreted *atomic actions* which can be seen as the conceptual entities at a chosen level of abstraction. The concept of *(syntactic) action refinement* supplies the possibility to provide a more concrete structure for an action in a later system design step: Given a process expression P one refines an atomic action a of P by a more complex process expression Q, obtaining a more detailed process expression $P[a \rightsquigarrow Q]$. The Hennessy-Milner-Logic HML [Mil80] provides the ability to formalize properties of processes.

Whereas the interplay between action refinement and various notions of bisimulation is well understood (see e.g. [AH91, GGR94]), little attention has been given so far to the interplay of action refinement and specification logics. In particular knowing $P \models \varphi$ does not provide information which formulas ψ are satisfied by $P[a \rightsquigarrow Q]$ or vice versa. Such knowledge however, could be used to facilitate the task of verification in a stepwise refinement of systems. We enrich HML by an operator for action refinement and provide the link between syntactic action refinement for TCSP (without hiding and internal nondeterminism) and action refinement in HML by showing that the assertion

$$P \models \varphi \Leftrightarrow P[a \rightsquigarrow Q] \models \varphi[a \rightsquigarrow Q] \tag{1}$$

holds under reasonable and weak restrictions.

Assertion (1) can be interpreted in various ways. Firstly we may interpret (1) as a simplification of the verification task, as we may check $P \models \varphi$ instead of $P[a \rightsquigarrow Q] \models \varphi[a \rightsquigarrow Q]$. Secondly we may read (1) from left to right, which embodies two views of verification: Firstly we can focus on the refinement of the specification φ. Given $P \models \varphi$, a refinement $\cdot[\alpha \rightsquigarrow Q]$ on φ automatically supplies a refined process term which satisfies the specification $\varphi[\alpha \rightsquigarrow Q]$. This can be seen as a concept of 'a-priori'-verification. Secondly we can focus on the refinement of process terms, i.e. we consider the refinement $P[\alpha \rightsquigarrow Q]$ of P, where $P \models \varphi$ and obtain automatically $P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$. One might argue that we could determine $P[\alpha \rightsquigarrow Q]$, search a specification ψ which reflects the refinement $\cdot[\alpha \rightsquigarrow Q]$ in the logic and then apply a *model checker* to establish $P[\alpha \rightsquigarrow Q] \models \psi$. However no hint is available which formula ψ reflects the refinement of P in an appropriate way. Hence we would have to test a sequence of formulas (of which we think that they reflect the refinement) $\psi_1, \psi_2, \ldots, \psi_n$ with the model checker where in general a application of the model checker might need exponential time. The formula $\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$ which is obtained from $\varphi[\alpha \rightsquigarrow Q]$ by removing all occurrences of the refinement operator via reduction may be of size exponential in the size of Q. However only one such exponential reduction invoked by the application of assertion (1) is necessary to obtain a formula which reflects the refinement appropriatly. Hence (1) can be used as a more efficient indicator of the logical consequences induced by refinements on process terms than model checking. The validity of (1) has another interesting implication: The assertion can be used by a system designer, who is not interested in full equality of refined process expressions modulo bisimulation equivalence, but in the fact, that two refined processes both satisfy a refined property (or a set of refined properties) of special interest. A preliminary (full) version of this extended abstract can be found in [MCS99a]. Work is in progress ([MCS99b]) which extends our approach to the modal mu-calculus of [Koz83].

References

[AH91] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Lecture Notes in Computer Science*, 510:506–519, 1991.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[GGR94] U. Goltz, R. Gorrieri, and A. Rensink. On syntactic and semantic action refinement. *Lecture Notes in Computer Science*, 789:385–404, 1994.

[Koz83] D. Kozen. Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[MCS99a] M. Majster-Cederbaum and F. Salger. On syntactic action refinement and logic. Technical Report 7/99, Fakultät für Mathematik und Informatik, Universität Mannheim, D7.27, Germany, 1999.

[MCS99b] M. Majster-Cederbaum and F. Salger. Syntactic action refinement in the modal mu-calculus and its applications to the verification of reactive systems. Technical Report (in preparation), Fakultät für Mathematik und Informatik, Universität Mannheim, D7.27, Germany, 1999.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, Berlin, 1 edition, 1980.

Formal Verification of Secret Sharing Protocol Using Coq

Kok Meng Yew, M. Zahidur Rahman, and Sai Peck Lee

Faculty of Computer Sc. & Information Technology
University Malaya, Kuala Lumpur, Malaysia

1 Introduction

Different cryptographic protocols have been formally proved by different researchers. But in the case of secret sharing (SS) protocol there is no attempt yet of proving. We show with an example of SS modeling, how SS protocol can be formally verified using Coq, a general theorem prover. In modeling our SS protocol we follow the approach of Dominique. The approach is based on the use of state-based general purpose formal methods , and on a clear separation between the modeling of reliable agents and that of intruders. The formalization for the intruder knowledge, axioms for manipulating them, as well as the protocol description can be transposed quite directly using Coq.

Secret sharing protocol The secret share distribution protocol can be described in the following steps:

i) Share preparation : Dealer prepares share of the given secret and also prepares auxiliary shares i.e. $\alpha_i = f(i)$ and $\rho_i = r(i)$ where f and r are two random polynomial whose $f(0) = s$ i.e. secret; ii)Distribution of share: Dealer distributes each share to the Players via private channel, i.e. secretly (DistS); iii)Broadcast of commitment: Dealer broadcasts commitment by one-way hash of all shares and auxiliary shares (Bcast); iv) Broadcast of verification: Each player compares his share and commitment and broadcasts result (i.e. either true or false) (Comt); v) If any verification result is false then: Dealer broadcasts secret share, auxiliary share and hash value of that player (BcastD).

Modeling of intruder knowledge The data that can be communicated between principals are basic data, data obtained by composition and data obtained by combining valid number of shares.

The domain of secret shares is noted by SH. If the domain of symmetric key is noted KS, while that asymmetric keys is noted by KA, then K is thus the union of KA and KS. The pair operator is used to represent reversible constructors. Thus the domain C of data components is defined inductively starting from the infinite and disjoint sets KS, KS, D and SH
$C = C_K|(C, C)|B \quad B = K|D|SH \quad K = KA|KS|K^{-1}$

For secret share, new pair of operations σ and σ' are also introduced which represent sharing a secret to participants and reconstructing participant share back to secret.

$s \overset{\sigma}{\Longrightarrow} s' \overset{def}{=} \{(s, s')|(\exists s'', c, t, n, c_1, c_2, \ldots, c_n \cdot s = c \cup s'' \wedge s' = s \cup c_1 \cup c_2 \cup \ldots \cup c_n\} \wedge t \leq n)$

$s \overset{\sigma'}{\Longrightarrow} s' \overset{def}{=} \{(s, s')|(\exists s'', c, t, n, c_1, c_2, \ldots, c_n \cdot s = c_1 \cup c_2 \cup \ldots \cup c_n \cup s'' \wedge s' = s \cup c\} \wedge t \leq n)$

where n is number of shares, t is threshold of the sharing scheme, c is share where as $c_1, \ldots c_n$ are the shares.

Knowledge manipulation For the secret sharing in addition to the rules introduced by Dominique [1], some new set of rules are introduced. Some extension of consistency and completeness rules are:

$card(c_{sr}) \geq t \wedge c_{sr}\, known\_in\, s \Longrightarrow c\, known\_in\, s$
$card(c_{sr}) < t \wedge c_{sr}\, known\_in\, s \Longrightarrow \neg(c\, known\_in\, s)$
$card(c_{sr}) \geq t \wedge c_{sr}\, comp\_of\ s \Longrightarrow c\, comp\_of\ s$

Formalization of the protocol In the state model of *Dealer*, α and ρ are secret share and support share for Players. The protocol is then modeled using the 8 basic actions. Such as

$action(s_d.at, 1d, s_a^{'}.at) \wedge s_i^{'} = s_i \cup ((s_d^{'}.\alpha_i, s_d^{'}.\rho_i)_{k'_{dP_i}} \wedge new(s_d^{'}.\alpha_i, s_i)$
$action(s_{P_i}.at, 1P_i, s_{P_i}^{'}.at) \wedge (s_d^{'}.C_1)\ known\_in\ s_I$
$action(s_d.at, 2d, s_{P_i}^{'}.at) \wedge s_i^{'} = s_i \cup (s_d^{'}.Hash(\alpha_i, \rho_i)) \wedge new(s_d^{'}.Hash(\alpha_i, \rho_i))$

where primed variables are used for representing values of state variables after the application of the operation and where by convention static variables that are not explicitly changed are supposed to be unchanged: the first action which only mentions new values for $s_d.at$ and s_i thus assumes by default that $s_d = s_d^{'}$ and so on. Predicate *action* is used to specify sequencing constraints, i.e. control structure for each role. Here we assume that D is repeating an infinite loop [1d, 2d, 4d] and $At_{D1}, \ldots, At_{D4}$ are for example the respective control points before each of the three actions. In the same way P_i is supposed to repeat [$1p_i$, $2p_i$, $3p_i$, $4p_i$]. The predicate action (at, act, at') can in this case be defined as the union of 8 relations $(at = At_{D1} \wedge act = 1d \wedge at' = At_{D2}) \vee \ldots \vee (at = At_{D4} \wedge act = 4d \wedge at' = At_{D1})$.

2 Conclusions

The use of general theorem proving verification methods to help the verification of cryptographic protocols looks promising. If a protocol has been analyzed using general theorem proving verification methods, one should however not be fooled into thinking that the protocol is one hundred percent correct. The verification model should be seen as a debugging tool, which explores potential attacks on the protocol in an automated manner.

References

[1] Dominique Bolignano. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communications Security*, March 1996.

On Feasibility, Boundedness, and Redundancy of Systems of Linear Constraints over R^2-Plane

(An Extended Abstract)

Nguyen Duc Quang

Computer Science Program, School of Advance Technology
Asian Institute of Technology
dquang@cs.ait.ac.th

A system of linear constraints is represented as a set of half-planes $S = \{(a_jX + b_jY + c_j \leq 0)\ j = 1 \ldots N\}$. Therefore, in the context of linear constraints two terms "set" and "system" can be used interchangeably whenever one of two is suitable. A set of linear constraints represents a convex polygon that is the intersection of all half-planes in the set. The convex polygon represented by S is called the feasible polygon of S. Such sets of linear constraints can be used as a new way of represent spatial data. These sets need to be manipulated efficiently and stored using minimal storage. It is natural to store only sets of linear constraints which are feasible and in irredundant format. Therefore, it is very important to find out if a given system is feasible and/or bounded and to find the minimal (irredundant) set of linear constraint which have the same feasible area with the given one. LASSEZ and MAHER (1988) have investigated algorithms to check if a system of linear constraints over multidimensional R^d is feasible. LASSEZ et al (1989) have investigated algorithms to eliminate redundant constraints from a system of linear constraints over R^d. Their algorithms are based on the Fourier variable elimination (similar with Gaussian elimination in solving the linear system of equations) and therefore have the running time $O(N^2)$ where N is the number of constraints, and as such it is not efficient. DYER (1984) and MEGIDDO (1983) have independent proposed linear time algorithms to solve the linear programming problem in 2- and 3-dimension cases. In this paper, we consider sets of linear constraints as a new way of representing geographic data and therefore only set of linear constraints over R^2, which is enough for our purpose. DYER's and MEGIDDO's algorithms need to be investigated further for our purpose. Feasibility of a set of linear constraint is in fact one output of the algorithms by DYER (1984) and MEGIDDO (1983). We mention here the algorithm of checking feasibility of sets of linear constraints for the completeness. We proposed here a linear time algorithm to check boundedness of sets of linear constraints. Boundedness of a set of linear constraints S can be found by applying algorithm by DYER (1984) and MEGIDDO (1983) to determine four extremity points of the convex feasible polygon P of S: (i) $(X_0, Y_{\min}) : Y_{\min} = \min\{Y : \exists X_0(X_0, Y) \in P\}$, (ii) $(X_1, Y_{\max}) : Y_{\max} = \max\{Y : \exists X_1, (X_1, Y) \in P\}$, (iii) $(X_{\min}, Y_0) : X_{\min} = \min\{X : \exists Y_0(X, Y_0) \in P\}$ and (iv) $(X_{\max}, Y_1) : X_{\max} = \max\{X : \exists Y_1(X, Y_1) \in P\}$. If S is infeasible or if

S is feasible and all of $X_{\min}$ and $X_{\max}$ and Y_{min} and $Y_{\max}$ are finite then S is bounded. Otherwise S is unbounded.

The algorithm by DYER (1984) and MEGIDDO (1983) have been proved to have the running time of $O(N)$. Therefore, the algorithms to check the feasibility and boundedness of a set of linear constraint over R^2 will have the running time of $O(N)$ where N is the number of linear constraints in S.

In this paper, we also propose a linear time algorithm to eliminate the redundancy of a set of linear constraints provided set of linear constraints has been pre-sorted. A set of linear constraints S is said redundant if there exist a proper subset $S' \subset S(S' \subseteq S$ and $S' \neq S)$ so that feasible polygon of S' and feasible polygon of S are exactly the same. Given a set of linear constraint S over R^2 with feasible polygon P. From the convexity of polygon P, our algorithm of eliminating the redundancy of S is based on the following observations:

- Suppose l_1 and l_2 is two edges of a convex polygon P with slope a_1, a_2, respectively satisfying $a_1 \leq a_2$. Given line l with slope a satisfying $a_1 \leq a \leq a_2$, saying l is an edge of P means that intersection of l with polygon P is not empty. Therefore l is an edge of P if and only if the intersection point of l_1 and l_2 is external to the polygon P. If l is not an edge of P then l is redundant and can be eliminated.
- The intersection point of any two adjacent edges of a convex polygon is a vertex of the polygon and therefore belongs to this polygon. Suppose l_1 and l_2 is two lines derived from two constraints of S with slope a_1, a_2, respectively satisfying $a_1 \leq a2$ and there does not exist l from S so that slope a of l satisfying $a_1 \leq a \leq a_2$. If the intersection point of l_1 and l_2 does not belong to P then one of these two corresponding constraints are redundant and can be eliminated.

With preprocessing when sets of linear constraints are sorted in running time of $O(N \log N)$, our algorithm of eliminating the redundancy of a set of N linear constraints are of $O(N)$.

These above algorithms can be applied to detect and compute intersection of two sets of linear constraints in linear time provided the sets of linear constraints have been pre-sorted: Given two sets of linear constraints $S_1 = \{(a_{1j}X + b_1Y + c_{1j} \leq 0)\ j = 1 \ldots N_1\}$ and $S2 = \{(a_{2j}X + b_2jY + c_{2j} \leq 0)\ j = 1 \ldots N_2\}$. Two sets S_1 and S_2 are merged into one set S. In order to detect the intersection of S_1 and S_2, we just apply the algorithm to check feasibility of S, if S is feasible then S_1 intersects S_2. Otherwise S_1 does not intersects S_2. In order to compute the intersection of S_1 and S_2, we apply the algorithm of eliminating the redundancy to S. The result set will be the intersection of S_1 and S_2.

As shown above, with preprocessing when sets of linear constraints are sorted in running time of $O(N \log N)$, our algorithms are of $O(N)$ where $N = (N_1 + N_2)$.

Using our algorithm in context of spatial data, given two convex polygons represented using two sets of linear constraints with numbers of constraints N_1 and N_2 respectively, we can detect and compute the intersection of two given convex polygon in the running time of $O(N1 + N2)$ which is optimal provided the two sets of linear constraints have been pre-sorted.

References

1. Lassez, J.L. & Maher, M.J. (1988) On Fourier's Algorithm for Linear Arithmetic Constraints, IBM Research Report, IBM T.J. Watson Research Center.
2. Lassez, J.L., Huynh, T. and Mcalloon (1989), K. Simplification and Elimination of Redundant Arithmetic Constraints, Proc. of NACLP 89, MIT Press.
3. Dyer M.E. (1984), Linear Time Algorithms for Two- and Three-variable Linear Programs, SIAM J. Comp. 13(1), 31-45.
4. Megiddo N. (1983), Linear Time Algorithm for Linear Programming in R^3 and Related Problems, SIAM J. Comp. 12(4), 759-776.

Deduction in Logic of Association Rules

Poster Abstract

Jan Rauch

Laboratory of Intelligent Systems, Faculty of Informatics and Statistics,
University of Economics, W. Churchill Sq. 4, 13067 Prague, Czech Republic.
rauch@vse.cz

Knowledge discovery in database (KDD for short) is fast growing discipline of informatics. It aims at finding new, yet unknown and potentially useful patterns in large databases. Association rules [1] are very popular form of knowledge extracted from databases. The goal of KDD can be e.g. finding interesting patterns concerning dependency of quality of loans on various attributes which can be derived from a bank database. These attributes can be arranged e.g. in data matrix *LOANS* in Tab. 1. There are n loans, each loan corresponds to one row of data matrix. Each column corresponds to an attribute derived from the database. Column TOWN contains information about domicile of the owner of the loan, column AGE contains age of the owner (in years). Column LOAN describes the quality of the loan (OK or BAD). Let us emphasize that usually there are tens of attributes describing both static (TOWN, AGE) and dynamic characteristics (derived e.g. from the transactions) of owners of loans.

An example of association rule with *confidence* 0.9 and *support* 0.1 is the expression "TOWN[*Tokyo*] $\rightarrow_{0.9,0.1}$ LOAN[*OK*]". It means that (i) at least 90 per-cent of loans owners of which live in Tokyo are OK and (ii) there is at least 10 per-cent of loans such that their owners live in Tokyo. In other words it means that $\frac{a}{a+b} \geq 0.9$ and $\frac{a}{a+b+c+d} \geq 0.1$, where a, b, c, d are frequencies from the four-fold contingency table Tab. 2 of Boolean attributes TOWN[*Tokyo*] and LOAN[*OK*] for data matrix *LOANS*.

row	TOWN	AGE	...	LOAN
r_1	*Tokyo*	17	...	OK
⋮	⋮	⋮	⋮	⋮
r_n	*Prague*	17	...	BAD

Tab. 1 - Data matrix *LOANS*

	LOAN[*OK*]	¬ LOAN[*OK*]
TOWN[*Tokyo*]	a	b
¬ TOWN[*Tokyo*]	c	d

Tab. 2 - Four-fold contingency table

The frequency a is the number of rows (loans) in data matrix *LOANS* satisfying both Boolean attribute TOWN[*Tokyo*] (owner of the loan lives in Tokyo) and Boolean attribute LOAN[*OK*] (loan is OK). The frequency b is the number of rows satisfying TOWN[*Tokyo*] and not satisfying LOAN[*OK*], etc.

This poster deals with *generalized association rules*. Generalized association rule (GAR for short) is an expression of the form $\varphi \approx \psi$ where φ and ψ are Boolean attributes derived from the columns of the analysed data matrix. Data matrix *LOANS* with derived Boolean attributes φ and ψ is in Tab. 3. Examples of such attributes are φ = TOWN[*Tokyo, Prague*] $\wedge$ AGE[17, 18, 19] and ψ = TOWN[*Berlin, Paris, London*] $\vee$ AGE[77]. Boolean attribute φ is true in a row r_i, iff it is satisfied for the row r_i: a value of TOWN is Tokyo or Prague, and value of AGE is 17, 18 or 19, analogously for other Boolean attributes.

Symbol $\approx$ is called a *4FT quantifier*. GAR $\varphi \approx \psi$ corresponds to an association between φ and ψ, 4FT quantifier is a name of a kind of this association. A value of GAR $\varphi \approx \psi$ in a given data matrix can be true or false. These value is computed using an associated function $F_\approx$ of 4FT quantifier $\approx$. $F_\approx$ is a $\{0,1\}$-valued function defined for all four-fold contingency tables. It is applied to a four-fold contingency table Tab. 4 of φ and ψ for the given data matrix.

row	TOWN	AGE	...	LOAN	φ	ψ
r_1	*Tokyo*	17	...	OK	1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_n	*Prague*	17	...	BAD	0	0

Tab. 3 - Derived Boolean attributes

	ψ	$\neg\psi$
φ	a	b
$\neg\varphi$	c	d

Tab. 4 - Four-fold table of φ and ψ

Goals of this poster are:

- Mention various classes (*implication, double implication* and *equivalency*) of 4FT quantifiers. Each class contains both simple associations of Boolean attributes and tests of statistical hypotheses.
- Introduce 4FT logical calculi formulae of which corresponds to GAR.
- Argue for usefulness of correct deduction rules of 4FT calculi for KDD.
- Show that there is a simple condition concerning Boolean attributes φ, ψ, φ' and ψ' which is satisfied iff the deduction rule $\frac{\varphi \approx \psi}{\varphi' \approx \psi'}$ is correct. This condition depends on the class of 4FT quantifiers.

References

1. Aggraval, R. et al: Fast Discovery of Association Rules. In Fayyad, U. M. et al.: Advances in Knowledge Discovery and Data Mining. AAAI Press / The MIT Press, 1996. 307–328
2. Hájek, P., Havránek T.: Mechanising Hypothesis Formation - Mathematical Foundations for a General Theory. Berlin - Heidelberg - New York, Springer-Verlag, 1978, 396 p.
3. Rauch, J.: Logical Calculi for Knowledge Discovery in Databases. In Principles of Data Mining and Knowledge Discovery, (J. Komorowski and J. Zytkow, eds.), Springer Verlag, Berlin, 47-57, 1997.
4. Rauch, J.: Classes of Four-Fold Table Quantifiers. In Principles of Data Mining and Knowledge Discovery, (J. Zytkow and M. Quafafou, eds.), Springer Verlag, Berlin, 203-211, 1998.
5. Rauch, J.: Four-Fold Table Calculi and Missing Information. In JCIS'98 Proceedings, (Paul P. Wang, editor), Association for Intelligent Machinery, 375-378, 1998.

This work is supported by grant 47160008 of the Ministry of Education of the Czech Republic.

Asynchronous Migration in Parallel Genetic Programming

Shisanu Tongchim and Prabhas Chongstitvatana

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok 10330, Thailand Tel: (662) 218-6982 Fax: (662) 218-6955
g41stc@cp.eng.chula.ac.th
prabhas@chula.ac.th

Genetic Programming (GP) was used to generate robot control programs for an obstacle avoidance task [1]. The task was to control an autonomous mobile robot from a starting point to a target point in a simulated environment. The environment was filled with the obstacles which had several geometrical shapes. In order to improve the robustness of the program, each program was evaluated under many environments. As a result, the substantial processing time was required to evaluate the fitness of the population of the robot programs.

To reduce the processing time, this present study introduced a parallel implementation. In applying the parallel approach to the algorithm program by using a conventional coarse-grained model, the result achieved only linear speedup since the amount of work was fixed – the algorithm was terminated when it reached the maximum generation. Hence, the parallel algorithm did not exploit the probabilistic advantage that the answer may be obtained before the maximum generation.

We tried in this present study another method to further improve the speedup by dividing the *environments* among the processing nodes. After a specific number of generations, every subpopulation was migrated between processors using a fully connected topology. The parallel algorithm was implemented on the dedicated cluster of PC workstations with 350 MHz Pentium II processors, each with 32 Mb of RAM, and running Linux as an operating system. These machines were connected via 10 Mbs ethernet cabling. We extended the program used in [1] to run under the clustered computer by using MPI as a message passing library.

In the first stage of the implementation, the migration was synchronized. The synchronizing migration resulted in uneven work loads among the processors. This was due to the fact that the robot performed the task until either the robot achieved the target point or reached an iteration limit. Hence, this migration scheme caused the evolution to wait for the slowest node.

In the second stage of the implementation, we attempted to further improve the speedup of the parallel algorithm by the *asynchronous migration*. When the fastest node reached predetermined generation numbers, the migration request was sent to all subpopulations. Therefore, this scheme caused the evolution of all subpopulations to proceed according to the *fastest* node.

The widely used performance evaluation of the parallel algorithm is the parallel speedup. To make an adequate comparison between the serial algorithm and parallel algorithm, E. Cantú-Paz [2] suggested that the two must give the same quality of the solution. In this paper, the quality of the solution is defined in terms of the *robustness*. The robustness of the generated programs from the parallel algorithm was demonstrated to be better than the serial algorithm. Consequently, the amount of work from the parallel algorithm in this experiment was not less than the serial algorithm.

Figure 1 illustrates the speedup observed on the two implementations as a function of the number of processors used. Both implementations exhibit superlinear speedup. The speedup curves taper off for 10 processors and the performance of the asynchronous implementation is slightly better than the performance of the synchronous implementation.

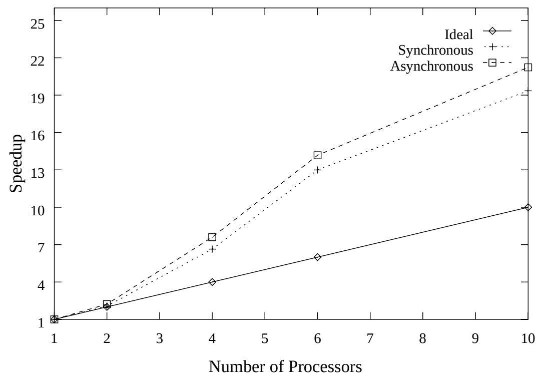


Fig. 1. Speedup

After obtaining some timing analyses, the results reveal the cause of the problem. The performance degradation in 10 processors is caused by the excessive communication time due to the broadcast function. Although the asynchronous migration reduces the barrier time effectively compared to the synchronous migration, the increase in the broadcast time in 10 processors obliterates this advantage. However, in case of a small number of processors (2,4,6), the reduction of the communication overhead from the asynchronous migration compared with the synchronous migration is considerable – i.e. the reduction in 2,4,6 nodes is 96.09%, 84.44% and 62.42% respectively. In terms of the wall-clock time, the asynchronous implementation in this work using 10 nodes is 21 times faster than the serial algorithm.

References

1. Chongstitvatana, P.: Improving Robustness of Robot Programs Generated by Genetic Programming for Dynamic Environments. In: Proc. of IEEE Asia-Pacific Conference on Circuits and Systems. (1998) 523–526
2. Cantú-Paz, E.: Designing Efficient and Accurate Parallel Genetic Algorithms. PhD thesis, University of Illinois at Urbana-Champaign (1999)

Verification Results for a Multimedia Protocol

Tony Tsang and Richard Lai

Department of Computer Science and Computer Engineering
La Trobe University
Victoria 3083, Australia.
{tsang, lai}@cs.latrobe.edu.au

Abstract. Verification of formal specifications for multimedia systems, with time taken into consideration, is still a subject of much research. Estelle, an internationally standardised Formal Description Technique (FDT) based on an extended finite state machine model, does not have enough expressive power to describe timing behaviours of distributed multimedia systems. To address this limitation, we have developed Time-Estelle, an extended Estelle which is capable of doing so. We also have developed a methodology for verifying Time-Estelle specification, which includes time properties. The verification method [1] involves translating a Time-Estelle specification to Communicating Time Petri Nets, which can then be verified using the tool, ORIS. In this paper, we describe the results obtained by applying our method to a real-life multimedia protocol, the Reliable Adaptive Multicast Protocol (RAMP).

1 The Verification Method

The verification of a complex and large multimedia system requires partitioning the system into components of manageable size. A reusable and time-dependent formal model is necessary for managing such a system. The primary stage consists of a formal time and functional description of the systems. The functional description is provided by the standard Estelle specification. The time description is provided by the extended Estelle specification. The intermediate stage focuses on translating Time-Estelle specification to Communicating Time Petri Nets (CmTPNs) [2]. In this stage, all the Time-Estelle module bodies and interaction points in the module header of any corresponding module body become *places* in a CmTPN, and transitions in the module body become *transitions* in CmTPN. *Tokens* in a place denote the fact that the message and state belonging to the module instance will be processed by one of the enabling transitions in the module body. The time information of Time-Estelle modules and transitions become the time constraint in the transitions and arcs of nets.

The final stage executes and analyses the system using the CmTPN based tool, ORIS. This approach highlights how the validation of CmTPN models relieves the state explosion problem by integration analysis. The integration analysis is the validation of a composed CmTPN system through the integration of the results of the unit analysis of its component modules. A merit of integration analysis is its flexible management of state space explosion by permitting a systematic concealment of local events.

2 Verification Results

The verification exercises revealed that the Burst mode of RAMP [3] does not have any deadlock or livelock in all the normal cases when collision does not occur. However, in the case of collision and overdue delay but without any functional operations in the "unexpected" states, deadlocks but no livelocks have been uncovered. The deadlocks are set out below:
a) A receiver issues the Ack message to the sender later than the expect time of the Sender, and the Receiver has changed to another state. The Sender has released a resent message simultaneously, but the Receiver does not functionally respond to the Resent message. As a result, a deadlock occurs.
b) A receiver issues a resent message to the Sender later than the idle time of the current burst period, and the Sender has changed to next burst state. The Sender cannot functionally resent the data messages of previous burst data stream after time-out; this state is unreachable. As a result, a deadlock occurs.

The following describes the results for verifying the Idle mode of RAMP. It is verified against the deadlock freeness property. The verification exercises revealed that there are several deadlocks when a collision or an unreliable situation occurs. The deadlocks are set out below:
a) If the Receiver does not receive an Idle message or a Data message within an adaptive time-out interval from the last Data message, the Receiver issues the resent message with the next sequence number. However, in the case of collision, the Sender does not receive the resent message so that the Sender does not resent the message. The Receiver loses this message. As a result, a deadlock occurs in the transfer of this message.
b) If after a time-out, the Receiver does not receive the missing message, the Receiver closes the control channel with the sender and this closes the RAMP flow to the Sender. However, the Sender does not receive the closed message from the Receiver so that the Sender continues sending the Idle message. As a result, a deadlock occurs in the Sender.
c) Receivers periodically unicast Idle messages to the Sender. Any Receiver's message to the Sender resets the Receiver Idle time-out period. If the Sender does not receive Idle message from a receiver, the Sender closes the connection to the receiver. However, the Sender loses the Idle messages periodically so that the Sender closes the connection also periodically. As a result, a deadlock occurs in the Sender.

References

1. T. Tsang and R. Lai, "Specification and Verification of Multimedia Synchronisation Scenarios Using Time-Estelle", Software Practice and Experience, John Wiley and Sons, Vol 28, No 11, pp.1185-1211, September 1998.

2. Giacomo Bucci and Enrico Vicario, “Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets”, IEEE Transactions on Software Engineering, December 1995.
3. Alex Koifman, Stephen Zabele, “RAMP: Reliable Adaptive Multicast Protocol”, Technical Assistance Support Center (TASC),1996.

Multipoint-to-Point ABR Service with Fair Intelligent Congestion Control in ATM Networks

Qing Yu and Doan B. Hoang

Basser Department of Computer Science, University of Sydney
doan@cs.usyd.edu.au

Abstract. Current standards of ATM can only support pt-pt (or unicast) connections and unidirectional point-to-multipoint (pt-mpt) connection and do not provide a scalable solution for truly multipoint-to-multipoint (mpt-mpt) communication. The main reason is that AAL5 does not provide multiplexing identification on a per cell basis. Cells from different packets on a single connection cannot be interleaved. To preserve AAL5 structure, additional mechanisms are needed at the merging point to differentiate packets and prevent cell mixing. In our previous study [1], we have proposed a Fair Intelligent Congestion Control (FICC) for ABR point-to-point traffic. It was demonstrated that FICC is simple, robust, efficient, scalable and fair relative to other proposed congestion control algorithms. In this paper we propose to apply FICC together with simple queueing and scheduling mechanism to provide efficient, fair bandwidth allocation and congestion control in a multipoint-to-point (mpt-pt) connection for heterogeneous service with different data rates. The simulation results show that FICC preserves all the desirable point-to-point properties, and performs equally well in multipoint-to-point connections.

FICC for Mpt-pt ABR Service in ATM Network

I. Description of the Fair Intelligent Congestion Control

The Fair Intelligent Congestion Control [1] treats rate allocation mechanisms for noncongestion and for congestion period in a similar and consistent manner. It aims for a target operating point where the switch queue length is at an optimum level for good throughput and low delay, and where the rate allocation is optimum for each connection. In order to estimate the current traffic generation rate of the network and allocate it among connections fairly, a Mean Allowed Cell Rate per output queue is kept at the switch. An explicit rate is then calculated as a function of the Mean Allowed Cell Rate, and a queue control function.

II. Queueing and Scheduling Mechanism of the Data

The Queueing and Scheduling mechanism employed is the same mechanism proposed in [2], Essentially, the mechanism is implemented as follows:

- To isolate cells from different source VCs and to prevent cells from different packets interleaving, a separate queue is employed to collect cells from a given source VC packet, cells from these queues are merged(packet-interleaving) into one single VC.

- To avoid cell-interleaving, Cells from each queue for the mpt-pt connection are scheduled for transmission on a per packet basis. A packet is fully transmitted before any cells from the next packet is transmitted.

III. Switch Algorithm for Multipoint-to-Point ABR

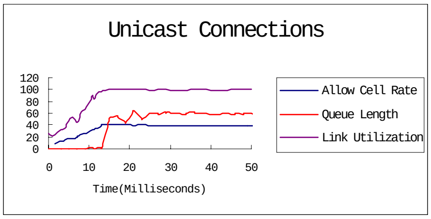
```
if received_backward_RM(ACR, ER, CI) {
Process this RM cell as in the unicast ABR algorithm,
in our case FICC algorithm;
Forward this RM cell to the corresponding link;
Return a backward RM(ACR, ER=MER, CI=MCI) cell to the
source;}
if received_forward_RM(ACR, ER, CI) {
Process this RM cell as in the unicast ABR algorithm,
in our case FICC;
Update the congestion condition: MCI=CI;
Update the Explicit Rate on the connection: MER=ER;
Discard this RM cell;}
```

Fig. 1. Switch Algorithm for Multipoint-to-Point ABR Service

The modified algorithm employed in this paper differs from its original [2] in the place where the unicast ABR algorithm is replaced by our Fair Intelligent Congestion Control in both forward and backward directions

IV. Simulation Results and Analysis

Simulation for various configurations has been performed. However, due to space limit, only simulation results for mpt-pt connection for parking-lot configuration are presented.



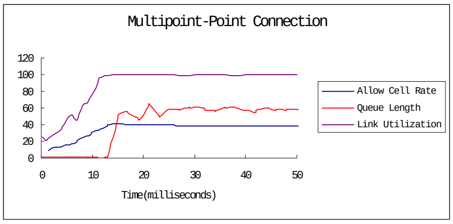


Fig. 2. FICC-Parking lot configuration. a) Unicast b) 4-to-1 Multipoint connection

Simulation results demonstrate that the Fair Intelligent Congestion Control scheme is effective in congestion control and allocate bandwidth fairly among VCs in mpt-pt connection. It preserves all its desirable properties from point-to-point application. These properties include simplicity in that it does not require per VC-accounting, robustness in that it does not depend critically to switch parameters, scalability in that the switch buffer requirements do not depend on the number of VC connections, fairness in that it can allocate bandwidth fairly among its connections. More importantly, FICC achieves high throughput and acceptable delay, and these two performance parameters can be specified in terms of target operating point.

References

1. Hoang, D. B., and Q. Yu, "Performance of the Fair Intelligent Congestion Control for TCP Applications over ATM Networks", to appear in *the Proceedings of the Second International Conference on ATM (ICATM'99)*, Colmar, France, June 1999, pp. 390-395.
2. Ren, W., Siu, K-Y, and Suzuki, H., "Multipoint-to-Point ABR Service in ATM Networks." *Computer Networks and ISDN systems*, Vol. 30, No. 19, Oct. 1998, pp. 1793-1810.

Author Index