Lecture Notes in Computer Science 2154
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Kim G. Larsen Mogens Nielsen (Eds.)

CONCUR 2001 – Concurrency Theory

12th International Conference
Aalborg, Denmark, August 20-25, 2001
Proceedings

Springer

Series Editors

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

Volume Editors

Kim G. Larsen
BRICS, Aalborg University, Department of Computer Science
Fredrik Bajersvej 7, 9220 Aalborg, Denmark
E-mail: kgl@cs.auc.dk

Mogens Nielsen
BRICS, Aarhus University, Department of Computer Science
Ny Munkegade, Bldg. 540, 8000 Aarhus C, Denmark
E-mail: mn@brics.dk

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Concurrency theory : 12th international conference ; proceedings / CONCUR 2001, Aalborg, Denmark, August 20 - 25, 2001. Kim G. Larsen ; Mogens Nielsen (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2154)
ISBN 3-540-42497-0

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

ISSN 0302-9743
ISBN 3-540-42497-0 Springer-Verlag Berlin Heidelberg New York

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

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

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10845533 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001) hosted by Basic Research in Computer Science (BRICS) and the Department of Computer Science at Aalborg University, Denmark, August 20–25, 2001.

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

The first two CONCUR conferences were held in Amsterdam (NL) in 1990 and 1991. The following ones in Stony Brook (USA), Hildesheim (D), Uppsala (S), Philadelphia (USA), Pisa (I), Warsaw (PL), Nice (F), Eindhoven (NL), and State College (USA). The proceedings have appeared in Springer LNCS, as volumes 458, 527, 630, 715, 836, 962, 1119, 1243, 1466, 1664, and 1877.

Of the 78 regular papers submitted this year, 32 were accepted for presentation and are included in the present volume. The conference also included talks by four invited speakers: Bengt Jonsson (Uppsala University, S), Robin Milner (University of Cambridge, GB), Shankar Sastry (University of California, Berkeley, USA), and Steve Schneider (Royal Holloway, University of London, GB). Additionally, there were two invited tutorials by Holger Hermanns and Joost-Pieter Katoen (Twente University, NL), and John Hatcliff (Kansas State University, USA). The conference had five satellite events:

- EXPRESS 2001 (Expressiveness in Concurrency), organized by Luca Aceto and Prakash Panangaden, held on 20 August 2001.
- GETCO 2001 (Geometric and Topological Methods in Concurrency), organized by Martin Raussen, held on 25 August 2001.
- RT-TOOLS (Workshop of Real-Time Tools), organized by Paul Pettersson, held on 20 August 2001.
- MTCS 2001 (Models for Time-Critical Systems), organized by Flavio Corradini and Walter Vogler, held on 25 August 2001.
- FATES 2001 (Formal Approaches to Testing of Software), organized by Jan Tretmans and Ed Brinksma, held on 25 August 2001.

We would like to thank all the Program Committee members and the subreferees who assisted in their work. Also, the Local Organization Chair, Anna Ingólfsdóttir, and the other members of the Local Organization, Luca Aceto and Arne Skou, and further members of BRICS deserve our gratitude for their contributions throughout the preparations. Thanks to the Workshop Chair, Hans Hüttel, and the workshop organizers. We would also like to thank the invited speakers and invited tutorial speakers, the authors of submitted papers, and all the participants of the conference. Special thanks to Brian Nielsen for installing and managing the START Conference system.

We gratefully acknowledge support from Det Obelske Familiefond, Thomas B. Thrige Foundation, Vaughan Pratt, Mindpass, the Department of Computer Science at Aalborg University, BRICS, and the City of Aalborg.

June 2001 Kim Guldstrand Larsen and Mogens Nielsen

CONCUR Steering Committee

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

Program Committee

Kim G. Larsen (Aalborg University, DK, Co-chair)
Mogens Nielsen (Aarhus University, DK, Co-chair)
Rajeev Alur (University of Pennsylvania, USA)
Frank de Boer (Utrecht University, NL)
Javier Esparza (Technische Universität München, D)
Wan Fokkink (CWI, NL)
Roberto Gorrieri (University of Bologna, I)
Petr Jancar (Technical University of Ostrava, CZ)
Orna Kupferman (Hebrew University, IL)
Marta Kwiatkowska (University of Birmingham, GB)
Oded Maler (Verimag, F)
Ugo Montanari (University of Pisa, I)
Uwe Nestmann (Ecole Polytechnique Fédérale de Lausanne, CH)
Ernst-Rüdiger Olderog (Universität Oldenburg, D)
Catuscia Palamidessi (Penn State, USA)
K.V.S. Prasad (Chalmers University, S)
Philippe Schnoebelen (ENS-Cachan, F)
Björn Victor (Uppsala University, S)
Walter Vogler (Universität Augsburg, D)
Igor Walukiewicz (Warsaw University, PL)
Alex Yakovlev (University of Newcastle, GB)

Referees

L. Aceto
T. Amnell
F. Arbab
L. de Alfaro
P. D'Argenio
E. Asarin
G. Auerbach
C. Baier
P. Baldan
R. Barbuti
G. Behrmann
M. Berger
M. Bernardo
K.J. Bernstein
E. Best
E. Bihler
S. Blom
C. Bodei
R. Bol
A. Bouajjani
M. Bozga
M. Bravetti
L. Brim
R. Bruni
G. Bruns
J. Burton
N. Busi
B. Caillaud
R. Cardell-Oliver
F. Cassez
I. Černá
H. Chockler
P. Ciancarini
B. Cook
Y. Cook
F. Corradini

V. Cremet
D. Dams
P. Darondeau
A. Dawar
J. Desel
R. Devillers
H. Dierks
S. Donatelli
D. Dubhashi
M. Duflot
G. Ferrari
I. Fischer
E. Fleury
N. De Francesco
S. Gay
R. van Glabbeek
S. Gnesi
G. Goessler
U. Goltz
J. Goubault-Larrecq
O. Grumberg
T. Henzinger
Y. Hirshfeld
J. Hoenicke
K. Honda
F. Honsell
M. Huth
A. Ingólfsdóttir
R. Janicki
B. Jeannet
A. Jeffrey
H.E. Jensen
P.K. Jensen
B. Jonsson
G. Juhas
M. Jurdzinski
J. Katoen
V. Khomenko
A. Kiehn
E. Kindler
H. Klaudel
J.W. Klop
B. Koenig
M. Köhler
M. Koutny
M. Křetínský
A. Kučera
Y. Lakhnech
C. Laneve
I. van Langevelde
F. Laroussinie
S. Lasota
J. Leifer
K. Lodaya
G.G.I. Lopez
G. Lowe
G. Lüttgen
B. Luttik
K. McMillan
M. Mendler
M. Merro
D. Miller
O. Moeller
F. Moller
R. Morin
M. Mueller-Olm
M. Mukund
R. De Nicola
P. Niebert
D. Niwinski
G. Norman
J. Nyström
A. Omicini
V. van Oostrom
K. Ostrovsky
P. Paczkowski
P. Panangaden
J. Parrow
P. Pettersson
A. Philippou
M. Pietkiewicz-Koutny
G.M. Pinna
M. Pistore
N. Piterman
J. van de Pol
F. Pommereau
E. Posse
W. Prasetya
C. Priami
A. Rabinovich
R. Ramanujam
J. Rathke
A. Ravara
A.P. Ravn
A. Rensink
M. Ribaudo
J. Riely
C. Roeckl
W.P. de Roever
M. de Rougemont
M. Ryan
D. Sands
L. Santocanale
V. Sassone
P. Savický
I. Scagnetto
A.M. Schettini
R. Segala
N. Soerensson
P. Sobocinski
D. D'Souza
J. Srba
J. Steggles
M. Steffen
J. Stříbrná
G. Sutre
M. Szreter
F. Tang
P. Tsigas
D. Turi
I. Ulidowski
Y. Usenko
F. Valencia
M. Vardi
V.T. Vasconcelos
H. Veith
E. de Vink
H. Wehrheim
M. Weichert
H. Wimmel
F. Xia
N. Yoshida
S. Yovine
H. Zantema
G. Zavattaro
J. Zwiers

Table of Contents

Model Checking

Process Algebra

Probabilistic Systems

Unfoldings and Prefixes

Model Checking

Logic and Compositionality

Games

Channel Representations in Protocol Verification
(Preliminary Version)

Parosh Aziz Abdulla and Bengt Jonsson

Dept. of Computer Systems, P.O. Box 325, S-751 05 Uppsala, Sweden
{parosh,bengt}@docs.uu.se

Abstract. In automated verification of protocols, one source of complications is that channels may have unbounded capacity, in which case a naive model of the protocol is no longer finite state. Symbolic techniques have therefore been developed for representing the contents of unbounded channels. In this paper, we survey some of these techniques and apply them to a simple leader election protocol. We consider protocols with entities modeled as finite state machines which communicate by sending messages from a finite alphabet over unbounded channels; this is a framework for which many techniques have been developed. We also consider a more general model in which messages may belong to an unbounded domain of values which may be compared according to a total ordering relation: the motivation is to study protocols with timestamps or priorities. We show how techniques from the previous setting can be extended to this more general model, but also show that reachability quickly becomes undecidable if channels preserve the ordering between messages.

1 Introduction

Protocol verification has, since 25 years, been a driving application for the development of automated verification techniques. State space exploration techniques were developed in this context [36,33], and it is one of the important application areas for current model checking tools such as SPIN [25] and UPPAAL [27].

Protocol verification involves the construction of a model of a protocol, which can be subject to analysis, e.g., by a model checking tool. The model should abstract from less relevant details of the protocol in order to facilitate the analysis. Typically, a protocol model consists of a number of processes, which communicate over channels of some kind. In many cases, the channels are a significant source of problems for the analysis. If communication channels are large or unbounded, naive model-checking cannot be performed exhaustively. When modeling a protocol, one must therefore be careful to model the channels in a way which suits subsequent analysis. Many model checkers do not support unbounded channels.

In general, unbounded channels have infinitely many states, and must therefore be represented symbolically in automated verification. Different representations have been proposed for different types of channels. In this paper, we

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 1–15, 2001.
© Springer-Verlag Berlin Heidelberg 2001

will survey some of these symbolic techniques and illustrate them on a simple example.

First, we consider a model of protocols with entities modeled as finite state machines which communicate by sending messages from a finite alphabet over unbounded channels. If channels are unordered, this model can be represented by Petri Nets. FIFO ordered channels has been considered rather extensively in protocol verification [8,23,15,24,30,31,34]. Since this model can simulate Turing machines [12], we will devote most attention to a weaker model in which the FIFO channels can spontaneously lose messages at any time. We will illustrate two types of symbolic representations and their use in checking whether some states are reachable. The first type represents *upward closed* sets. It is useful in backward reachability analysis, since it gives a lower bound on which messages must be in each channel for a certain set of states to be reachable. This representation was used in [5] to decide the reachability problem, and in [5,21] to decide the termination problem. The second type (called Simple Regular Expressions in [3]), dually represents *downward closed* sets. This representation is useful in forward reachability analysis, since it gives an upper bound on which messages *may* be in the channel (we can never be sure whether a message is in the channel, since it can be lost). Analogous symbolic representations of upward closed sets have also been used for unordered channels and related models like Petri nets and broadcast systems [20,4,22,19,17].

We also consider how the techniques illustrated in the first part may be extended to models with an infinite set of message values, on which a limited set operations can be performed. We focus on the case where the domain of message values is equipped with a total ordering, which may be used in guards. The motivation is to study protocols with timestamps or priorities. We present a negative result showing that with lossy FIFO channels one can simulate perfect FIFO channels already if tests for equality and inequality are allowed on messages. For unordered channels, however, the backward reachability analysis presented in the first part of the paper can be used as a decision procedure. Using ideas from our earlier work [6,7] we can show that the techniques carry over to handle messages on which an ordering relation is defined.

This small survey is organized as follows. In the next section, we define protocols with a finite message alphabet and present a simple leader election protocol as a running example. Forward and backward reachability analysis with associated symbolic representations are presented in Section 3. In Section 4, we extend the model to an infinite set of messages, and present an undecidability result for lossy FIFO channels, and an extension to messages with an ordering relation. Section 6 contains discussion and conclusion.

2 Protocols with a Finite Set of Messages

In this section, we present our first protocol model: finite-state processes which communicate over unbounded channels using a finite message alphabet. In this model, a program has two parts: a control part and a channel part. The con-

trol part represents the combined behavior of the processes that perform local computations and communicate over the channels. It is represented by a finite automaton, which is typically the cross-product of control parts of the finite-state processes in the system. To each transition there may be associated the transmission or reception of a message to or from a channel. The channel part consists of a set of channels, each of which contains a potentially unbounded number of messages from a finite alphabet. Each channel can be ordered or unordered, lossy or non-lossy.

Notation. For a set $\mathcal{M}$ we use $\mathcal{N}^{\mathcal{M}}$ to denote the set of multisets (bags) of elements in $\mathcal{M}$, i.e., the set of mappings from $\mathcal{M}$ to the natural numbers $\mathcal{N}$. The empty bag is denoted by $\emptyset$. For $x, x' \in \mathcal{N}^{\mathcal{M}}$ we let $x \bullet x'$ denote the multiset union of x and x'. Define the partial order $\preceq$ on $\mathcal{N}^{\mathcal{M}}$ by $x \preceq x'$ if x is a subbag of x', i.e., $x(m) \leq x'(m)$ for all $m \in \mathcal{M}$.

For a set $\mathcal{M}$ we use $\mathcal{M}^*$ to denote the set of finite strings of elements in $\mathcal{M}$. The empty string is denoted by ε. For $x, x' \in \mathcal{M}^*$ we let $x \bullet x'$ denote the concatenation of x and x'. Define the partial order $\preceq$ on $\mathcal{M}^*$ $x \preceq x'$ if and only if x is a (not necessarily contiguous) substring of x'.

Note that we have overloaded the notations $\bullet$ and $\preceq$ for both bags and strings. We allow any element $m \in \mathcal{M}$ to be interpreted as the bag or string with the only element m, and thus $x \bullet m$ denotes the addition of element m to the bag or string x. Note also that the relation $\preceq$ closely corresponds to the notion of losing messages: $x \preceq x'$ if x can be obtained by deleting (losing) elements from the bag or string x'.

Programs. A *program* consists of

- A *channel part*, given by a finite set C of *channels*, and a finite set $\mathcal{M}$ of *messages*. Each channel is either ordered or unordered, either lossy or nonlossy.
- A *control part*, defined by a finite automaton $\langle S, s_0, \delta \rangle$, where S is a finite set of *control states*, $s_0 \in S$ is the *initial control state*, and δ is a finite set of *transitions*. Each transition is a triple of the form $\langle s, op, s' \rangle$, where $s, s' \in S$, and op is either the empty label ϵ or an operation of form $c!m$ or $c?m$, where $c \in C$ and $m \in \mathcal{M}$, which denotes the transmission of message m to channel c, or reception of message m from channel c, respectively.

Semantics. A *global state* γ of a program is a pair $\langle s, w \rangle$, where $s \in S$, and w is a mapping from C to bags and strings over $\mathcal{M}$ (depending on whether each particular channel is ordered or unordered). The *initial global state* γ_0 is $\langle s_0, w_0 \rangle$, where w_0 maps each channel to the empty bag or string. The global state can be changed by performing transitions in δ. More precisely, the global state of a nonlossy system can change from γ to γ', denoted $\gamma \longrightarrow \gamma'$, as follows.

- If $\langle s, \epsilon, s' \rangle \in \delta$ then $\langle s, w \rangle \longrightarrow \langle s', w \rangle$ for any w.
- If $\langle s, c!m, s' \rangle \in \delta$ then $\langle s, w \rangle \longrightarrow \langle s', w[c := w(c) \bullet m] \rangle$ for any w.
- If $\langle s, c?m, s' \rangle \in \delta$ then $\langle s, w[c := m \bullet w(c)] \rangle \longrightarrow \langle s', w \rangle$ for any w.

Here $w[c := w(c) \bullet m]$ denotes the mapping which maps c to $w(c) \bullet m$ and any $c' \neq c$ to $w(c')$. Extend $\preceq$ in the natural way to global states of form $\langle s, w\rangle$ by $\langle s, w\rangle \preceq \langle s', w'\rangle$ is $s = s'$ and $w(c) \preceq w'(c)$ for $c \in C$. The global state of a lossy system can change from γ to γ', denoted $\gamma \longrightarrow \gamma'$ if the corresponding nonlossy system can perform $\gamma'' \longrightarrow \gamma'''$ for γ'', γ''' with $\gamma'' \preceq \gamma$ and $\gamma' \preceq \gamma'''$.

A set Γ of global states is *upward closed* (UC) if $\gamma \in \Gamma$ and $\gamma \preceq \gamma'$ imply $\gamma' \in \Gamma$. We define the notion of *downward closed* (DC) analogously.

The behavior of a program depends on whether the channels are ordered or unordered and on whether they are lossy or nonlossy. In the following, we will assume that all channels of a program are of the same kind, and use the terms OL (ordered lossy), ON (ordered nonlossy), UL (unordered lossy), and UN (unordered nonlossy) for the four program models.

Reachability. A global state γ' is said to be *reachable* from a global state γ if $\gamma \stackrel{*}{\longrightarrow} \gamma'$. A global state γ is said to be *reachable* if γ is reachable from the initial global state γ_0. For a program and a set Γ of global states, the *reachability problem* asks whether some state in Γ is reachable. Typically, the set Γ are undesirable states, which should not occur when the system executes. In the following, we will mostly consider the special problem of *control state reachability* for a set $T \subseteq S$ of control states, asking whether the set $\{\langle s, w\rangle \ : \ s \in T\}$ of global states is reachable.

The reachability problem is related to checking of *safety properties*. A safety property can be described by specifying a set of finite sequences of states or transitions that are allowed to occur when the system executes. If the set of allowed sequences is regular, then there is a standard procedure for transforming the problem of checking a safety property into the problem of checking for reachability [35].

Example. As a running example, we use a simplification and variation of a leader election protocol, due to LeLann [28]. Assume a set of process connected into a ring by channels. Each process has a unique identity. In the algorithm, each process starts the algorithm by transmitting its identity over its outgoing channel to the next neighbor. Thereafter, the process waits to receive messages on its incoming channel. Each arriving identity is forwarded to the outgoing channel, except when the received identity is that of the process itself: in this case the process terminates the algorithm without forwarding its identity again. Each process elects as leader the process with the least identity among the ones received (including its own identity) during the algorithm. Thus, a process need not store all received identities during the algorithm, but only maintain the minimum identity among its own and the ones received so far. A correctness property of the algorithm is that all processes should elect the same leader.

In order to illustrate the algorithm for all four channel models with unbounded channels, we will vary the algorithm: each process will initially transmit its identity an unbounded number of times instead of just once, and channels may belong to any of the four channel models. With this modification, the processes should report the same leader if *all* processes terminate. Note that if channels

are lossy, then some process may not terminate, in which case it is possible for two other terminating processes to disagree on the identity of the leader.

In this section, we specialize the protocol to two finite-state processes, with identities 1 and 2. They communicate over the channel c_1, which transfers messages from process 2 to process 1, and channel c_2, which transfers messages from process 1 to process 2. The set of messages $\mathcal{M}$ is simply the set of identities $\{1, 2\}$. Finite automata for the two processes are shown in Figure 1. In the figure, we have slightly extended the notation and let a transition labeled by $c?m/c!m$ denote two sequentially ordered transitions, the first of which is labeled $c?m$ and the second labeled $c!m$. The states of processes are labeled I (initial), W (waiting

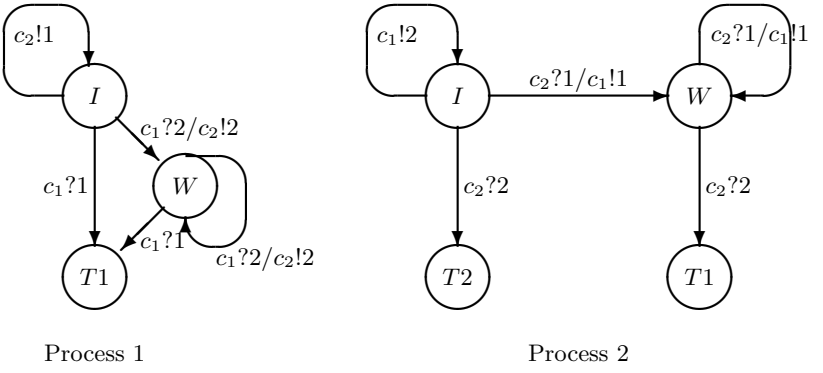


Fig. 1. Leader Election Protocol for Two Processes

for messages), and Ti for $i = 1, 2$ (terminated with i as elected leader). Initially, both processes are in state I, and the channels are empty. Process 1 initially transmits its identity an unbounded number of times. When receiving message 2, it compares that message with its current identity, and concludes that identity 1 is still the leading candidate for becoming a leader. Consequently, message 2 is simply forwarded to the outgoing channel. When receiving its own identity 1, the process terminates and announces that process 1 is the leader. Process 2 initially transmits its identity an unbounded number of times. When receiving message 1, it changes its internal state to reflect that 1 is the minimal identifier. Each received 1 is forwarded to the outgoing channel. When receiving message 2, the process terminates. In state $T2$ it announces 2 as the leader, and in state $T1$ it announces 1 as the leader.

According to our model, we represent control states as pairs of control states of each process. The above correctness property is expressed by requiring that the control state $\langle T1, T2\rangle$ is not reachable. This state represents an incorrect termination of the algorithm in which each process elects itself as leader.

3 Reachability Analysis

In this section, we illustrate existing techniques for checking reachability, applied to the system in Figure 1. In Section 3.1, we illustrate backward reachability analysis for the OL model [5] from our earlier work, and thereafter for the UL and UN models [4,22]. In Section 3.2, we illustrate forward reachability analysis for the OL model according to [3]; there are also other approaches [21]. We will also illustrate the approach of using standard finite automata (under the name QDDs) for the more general ON model [9,10]. For the UN and UL models, forward reachability analysis has a large literature (e.g., [26,20,18,38]).

3.1 Backward Symbolic Reachability Analysis

We first consider symbolic backward reachability analysis. Like standard symbolic model checking [32,16,13], it is based on symbolic calculation of pre-images, but with a specific representation of sets of states. The algorithm for checking reachability of a set Γ of global states consists in calculating the set of global states from which a state in Γ is reachable. For a set Γ of global states, let let $pre(\Gamma)$ denote the set $\{\gamma \ : \ \exists \gamma' \in \Gamma \ . \ \gamma \longrightarrow \gamma'\}$ of states from which a state in Γ can be reached by performing a transition. The naive version of the backward reachability algorithm for checking control state reachability of a set T of control states consists in generating a sequence $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ of sets of global states, where $\Gamma_0 = \{\langle s, w\rangle \ : \ s \in T\}$ and $\Gamma_{i+1} = \Gamma_i \cup pre(\Gamma_i)$, which stops when $\Gamma_{i+1} = \Gamma_i$. Then T is reachable if and only if $\gamma_0 \in \Gamma_i$.

In the UL, UN, and OL models, it is useful to represent sets of states as finite unions of atomic constraints. An *atomic constraint* is given by a global state $\langle s, w\rangle$ and denotes the UC set $[\![\langle s, w\rangle]\!] = \{\langle s, w'\rangle \ : \ w \preceq w'\}$. Programs in the UL, UN, and OL models now satisfy the following two important properties:

- Any program is *monotonic* with respect to $\preceq$, i.e., whenever $\gamma \longrightarrow \gamma'$ and $\gamma \preceq \gamma_1$ there is a γ_1' such that $\gamma_1 \longrightarrow \gamma_1'$ and $\gamma' \preceq \gamma_1'$. In other words, $\preceq$ is a simulation relation on global states.
- $\preceq$ is a *well quasi-ordering* (wqo) on the set of global states, i.e., in each infinite sequence $\gamma_0\ \gamma_1\ \gamma_2\ \gamma_3 \cdots$ of global states, there are indices $i < j$ such that $\gamma_i \preceq \gamma_j$.

Monotonicity implies that $pre(\Gamma)$ is UC for any UC set Γ. Well quasi-orderedness of $\preceq$ implies that any UC set is a *finite* union of atomic constraints, and that any increasing sequence $\Gamma_0 \subseteq \Gamma_1 \subseteq \Gamma_2 \subseteq \cdots$ of UC sets eventually converges so that there is an i with $\Gamma_j = \Gamma_i$ whenever $i \leq j$. Together, this means that the iterative calculation of $pre^*(\Gamma_0)$ can be carried out as above, using finite sets of atomic constraints as symbolic representation, and that it is guaranteed to terminate after a finite number of iterations [4,22].

Example. Let us use this technique to check whether the erroneous terminal state is reachable in the OL model. We represent global states as 4-tuples

$\langle s_1, s_2, w_1, w_2\rangle$ where s_i is the control state of process i, and w_i is the contents of channel c_i. We want to investigate whether the atomic constraint generated by $\langle T1, T2, \varepsilon, \varepsilon\rangle$ is reachable. Let $\Gamma_0 = [\![\langle T1, T2, \varepsilon, \varepsilon\rangle]\!]$. Then

$$\Gamma_1 = \Gamma_0 \cup [\![\langle I, T2, 1, \varepsilon\rangle]\!] \cup [\![\langle W, T2, 1, \varepsilon\rangle]\!] \cup [\![\langle T1, I, \varepsilon, 2\rangle]\!]$$
$$\Gamma_2 = \Gamma_1 \cup [\![\langle I, I, 1, 2\rangle]\!] \cup [\![\langle I, T2, 21, \varepsilon\rangle]\!] \cup [\![\langle W, T2, 21, \varepsilon\rangle]\!] \cup [\![\langle W, I, 1, 2\rangle]\!]$$
$$\Gamma_3 = \Gamma_2 \cup [\![\langle W, I, 21, \varepsilon\rangle]\!] \cup [\![\langle I, I, 21, \varepsilon\rangle]\!]$$

Here the procedure converges, without containing the initial state $\langle I, I, \varepsilon, \varepsilon\rangle$. Note that in set Γ_2, the sets $[\![\langle I, T2, 21, \varepsilon\rangle]\!]$ and $[\![\langle W, T2, 21, \varepsilon\rangle]\!]$ are contained in the the already generated sets $[\![\langle I, T2, 1, \varepsilon\rangle]\!]$ and $[\![\langle W, T2, 1, \varepsilon\rangle]\!]$. They are therefore redundant, and are discarded; they will not be used further to calculate predecessors.

In the UN and UL models, the analysis is very similar. In atomic constraints of form $\langle s_1, s_2, w_1, w_2\rangle$, the component w_i will be a bag rather than a sequence. The procedure looks very similar, except that constraints do not order messages in channels.

3.2 Forward Reachability Analysis

Backward reachability analysis, using calculation of preimages, is a suitable technique for checking that a certain invariant is satisfied. In fact, it generates the weakest inductive invariant which implies the invariant to be checked. In many cases, however, one would like to generate invariants which are as strong as possible. For instance, we could be interested in knowing the set of possible contents of a certain channel, or to generate a finite-state abstraction of the protocol, which will be useful in a subsequent analysis of a program with channels. To generate a representation of the set of reachable global states, forward reachability analysis is more appropriate. In this section, we will illustrate techniques for doing this in the OL and ON models.

Representing Sets in the OL Model. Consider what is a suitable symbolic representation of sets of global states in the OL model. Because of the possibility of losing messages, the set of reachable states will always be a downward closed (DC) set. Since any DC set is regular, the set of reachable global states of a program in the OL model be characterized as a regular set [5,14]. On the other hand, it follows from undecidability results by Mayr [29] that this characterization cannot be effectively constructed. Various techniques for approximating the set of reachable states have been developed [9,10,3,11], which are based on forward reachability analysis.

Let $post(\Gamma)$ denote the set $\{\gamma \ : \ \exists \gamma' \in \Gamma \ . \ \gamma' \longrightarrow \gamma\}$. One immediately observes that the naive forward reachability analysis, based on successive calculations of $post^i(\Gamma_0)$, where Γ_0 is the set of initial states, will not converge whenever the program contains a loop with retransmissions. Therefore, this technique has been extended with *acceleration* techniques for calculating the effect of certain kinds of loops. Below we will illustrate the approach of [3] for the OL model.

A *Simple Regular Expressions (SRE)* is a sum of products of regular expressions of form $(m + \varepsilon)$ and $(m_1 + \cdots + m_n)^*$. In [3], it is shown that *SREs* can express exactly the class of DC sets, and how to calculate the postcondition of an arbitrary number of iterations of a simple loop with respect to an SRE. We will illustrate this on the example.

Example. Let us apply forward reachability analysis to the example in Figure 1. We will represent sets of states as unions of 4-tuples $\langle s_1, s_2, r_1, r_2 \rangle$ where s_i is the control state of process i, and r_i is an *SRE* representing a set of contents of channel c_i. The set of initial states is the set $\langle I, I, \varepsilon, \varepsilon \rangle$. We first explore the effects of the simple self-loops in the states I, which is obviously to add an arbitrary sequence of messages to the corresponding channel. This means that the set of states $\langle I, I, 2^*, 1^* \rangle$ is reachable. The transitions from this set to control state W in each process add the sets represented by

$$\langle W, I, 2^*, 1^*(2 + \varepsilon) \rangle \quad \langle I, W, 2^*(1 + \varepsilon), 1^* \rangle \quad \langle W, W, 2^*(1 + \varepsilon), 1^*(2 + \varepsilon) \rangle \quad .$$

The last set represents the effects of both transitions to W. Note that all these sets account for the possibility that the transmitted message is lost, by appending, e.g., $(1 + \varepsilon)$ instead of just 1. Now comes the interesting step: to calculate the effects of the forwarding loops in states W. Consider the set $\langle W, W, 2^*(1 + \varepsilon), 1^*(2 + \varepsilon) \rangle$ and the self-loop of process 1 which receives message 2 and forwards it. Since there may be an unbounded number of 2's in the incoming channel, this loop may consume an arbitrary long initial string of 2's, and then transmit them to the outgoing channel. The effect is the set $\langle W, W, 2^*(1 + \varepsilon), 1^*2^* \rangle$. We can now perform a similar argument using the forwarding loop of Process 2, and arrive at the set $\langle W, W, 2^*1^*, 1^*2^* \rangle$. By applying the same technique to the other sets in the previous display, we obtain the sets

$$\langle W, I, 2^*, 1^*2^* \rangle \quad \langle I, W, 2^*1^*, 1^* \rangle \quad \langle W, W, 2^*1^*, 1^*2^* \rangle \quad .$$

Finally, we should consider the effect of transitions to final states. These transitions consume only one message, if it is present, and will to the already generated sets add the sets

$$\langle W, T2, 2^*, 2^* \rangle \quad \langle T1, W, 1^*, 1^*2^* \rangle \quad \langle W, T1, 2^*1^*, 2^* \rangle \quad \langle T1, T1, 1^*, 2^* \rangle \quad .$$

We see that the "dangerous" transition from control state $\langle W, T2 \rangle$ to $\langle T1, T2 \rangle$ is impossible since the incoming channel does not contain any message 1.

Abstract Transition Graph. The result of the preceding analysis can be summarized in the symbolic abstract state transition graph, as in Figure 2, which describes the possible transitions between different control states. To each control state, we associate the possible channel contents. An abstract transition graph may be used in further analysis of the protocol, e.g., as in [1] for the Bounded Retransmission Protocol.

We note that backward reachability analysis, as in Section 3.1, is less suitable as a basis for generating an abstraction of the program. For this particular

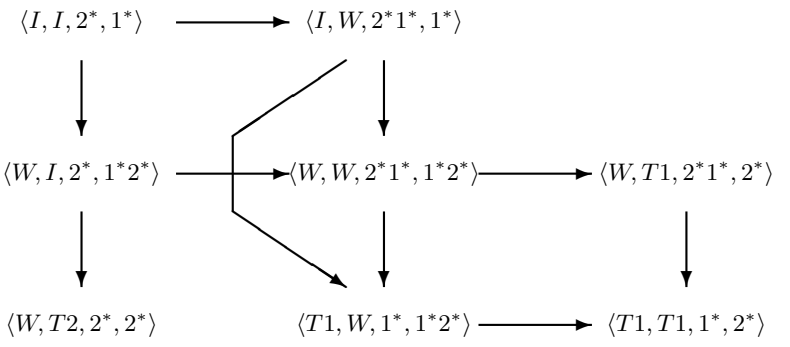


Fig. 2. Abstract Transition Diagram of Leader Election Protocol for Two Processes. All states except the bottom right have a self-loop.

example, the backward analysis gives no information about the channel contents when Process 2 is in control states W or $T1$, since the analysis investigates only the reachability of control state $\langle T1, T2\rangle$.

Representing Contents by Finite Automata. Sets of channel states can also be represented as ordinary regular sets, using finite automata. This approach is the basis for the *Queue Decision Diagram (QDD)* representation developed by Boigelot and Godefroid [9,10]. In this work, a global state is represented as a word, obtained by concatenation of the channel contents in a certain order. The approach is generally applicable in the ON model; for the OL model on can first modify the program (e.g., for each message transmission, a corresponding null transition is added). In [9] it is described how to calculate the effect of loops which first receive a (possibly empty) sequence from one channel and thereafter transmit a (possibly empty) sequence to another channel. The control loops of the processes in Figure 1 are of this form. In comparison to the techniques for *SRE*s described earlier, this approach is applicable to a more general model. On the other hand, using *SRE*s for the OL model, one can calculate the effect of a larger class of loops; a characterization of such loops is given in [2].

When applying the technique of [9] using QDDs to the example, we obtain essentially the same set of reachable states as that shown in Figure 2.

Forward reachability analysis in the UN and UL models can be performed analogously to the analysis for ordered channels. Techniques for calculating the effect of loops are well-developed (e.g., [26,20,18,38]).

4 Channels with an Unbounded Message Alphabet

In the previous section, we assumed that messages in channels are taken from a finite set. In this section, we will consider a slightly more general model of

programs that operate on a potentially infinite set of data, which can be stored in program variables, and transmitted as messages. We will consider program models in which very limited operations can be performed on data. In this section, we will consider a model where data values are equipped with a total order, which may be used in guards of transitions. We do not allow any other arithmetic on data values. One intended application is to consider protocols with time stamps, which may be compared with each other. In this section, we will also give a negative undecidability result, which holds in the more restricted model where we allow only tests for equality and inequality between values.

Programs. We extend the program model of the preceding section as follows. The finite set $\mathcal{M}$ of messages is replaced by an infinite set $\mathcal{D}$ of data values, which is equipped with a total order $<$. The control component is extended by a finite set $x_1, \ldots, x_m$ of *program variables*. Each transition in the control component may be labeled either by a receive statement of form

$$c?v; g(v, x_1, \ldots, x_m) \longrightarrow x_1, \ldots, x_m := p_1, \ldots, p_m$$

or a send statement of form

$$g(v, x_1, \ldots, x_m) \longrightarrow c!v; x_1, \ldots, x_m := p_1, \ldots, p_m$$

where v is a temporary variable and c is a channel in C; each parameter p_i is either v or among $x_1, \ldots, x_m$. The guard $g(v, x_1, \ldots, x_m)$ is a boolean combination of equalities ($=$) and comparisons (according to $<$) over v and $x_1, \ldots, x_m$. Intuitively, v is a free variable which may be bound to the received message in the receive statement, and which may be bound to any value in a send statement; in send statements it can represent the generation of new identifiers. We will sometimes use the abbreviation $\overline{x}$ for $x_1, \ldots, x_m$ and $\overline{p}$ for $p_1, \ldots, p_m$.

Semantics. A *global state* γ is a triple $\langle s, \sigma, w \rangle$, where $s \in S$, where σ is a mapping from $\{x_1, \ldots, x_m\}$ to $\mathcal{D}$, and where w maps each channel in C to a bag or string over $\mathcal{D}$. We use the notation $\sigma \models g$ to denote that the boolean expression g is satisfied by the mapping σ. The global state of a nonlossy system can change from γ to γ', denoted $\gamma \longrightarrow \gamma'$, as follows.

- For a transition from s to s' labeled by the receive statement $c?v; g(v, \overline{x}) \longrightarrow \overline{x} := \overline{p}$, we have $\langle s, \sigma, w[c := d \bullet w(c)] \rangle \longrightarrow \langle s', \sigma', w \rangle$ for any $d \in \mathcal{D}$ such that $\sigma \models g(d, \overline{x})$, where $\sigma'(x_i) = \sigma(p_i)$ if p_i is x_j for some j, otherwise (if p_i is v) $\sigma'(x_i)$ is d.
- For a transition from s to s' labeled by the send statement $g(v, \overline{x}) \longrightarrow c!v; \overline{x} := \overline{p}$, we have $\langle s, \sigma, w \rangle \longrightarrow \langle s', \sigma', w[c := w(c) \bullet d] \rangle$ for any $d \in \mathcal{D}$ such that $\sigma \models g(d, \overline{x})$, where $\sigma'(x_i)$ is $\sigma(p_i)$ if p_i is x_j for some j, otherwise (if p_i is v) $\sigma'(x_i)$ is d.

Definitions of transitions of lossy channels, reachable, are as in Section 2.

5 Reachability Analysis

We will in this section consider the control state reachability problem for programs defined in the model of Section 4. We will present two results. We first consider the OL model, where the control state reachability problem now becomes undecidable. In fact, it is undecidable also if only tests for equality and inequality between data values is allows. The undecidability result can be proven through a reduction from the control state reachability problem for perfect channel systems with a finite set of messages. Brand and Zafiropulo [12] showed that perfect channel systems can simulate Turing machines, and hence any nontrivial problem for perfect channel systems is undecidable. On the other hand, if guards may only test for equality (under an even number of negations) then the problem is decidable; the model is then similar to the model of data independent programs considered by Wolper [37].

5.1 Ordered Lossy Channels

In this section, we consider programs where guards are boolean combinations of equalities and inequalities. We then have the following negative result concerning decidability:

> The control state reachability problem is undecidable for OL programs where guards are boolean combinations of equalities and inequalities.

This can be shown by a reduction from the control state reachability problem for systems with perfect FIFO channels and a finite message alphabet, i.e., a program in the ON model of Section 2. Suppose we are given such a system $\mathcal{P}_{ON}$ and a control state s_f. We shall construct a program $\mathcal{P}_{OL}$ in the OL model over an infinite set $\mathcal{D}$ of values, which "simulates" $\mathcal{P}_{ON}$ in the sense that s_f is reachable in the lossy channel system if and only if s_f is reachable in $\mathcal{P}_{ON}$.

The main idea of the construction is to let $\mathcal{P}_{OL}$ be identical to $\mathcal{P}_{ON}$, but to add a protocol for transmission of messages, which makes it possible for the receiver at one channel to detect whether any message has been lost during transmission. If a loss is detected, the operation of the protocol is immediately aborted. Due to the availability of an infinite set of different messages, it is possible to assign "unique identifiers" to each message. This must be done in such a way that the receiver can detect whether some message has been lost. If the infinite set of different messages would have been the natural numbers, then if the $\mathcal{P}_{ON}$ transmits a sequence $m_1\ m_2\ m_3\ m_4 \ldots$ over a channel, then $\mathcal{P}_{OL}$ would transmit the sequence $0\ m_1\ 1\ m_2\ 2\ m_3\ 3\ m_4 \ldots$. It is easy to see that the receiver can detect whether any message has been lost.

In the current model, however, the set $\mathcal{D}$ is a set of "anonymous" values which can only be tested for equality and inequality. We therefore need a little trick to achieve the same effect. What is needed is to find for each message a "unique identifier" which is different from the previous identifiers, such that a receiver can detect when some identifier is missing. We can do this by using a predicate

fresh to generate new sequence numbers, and to tag each new sequence number with a copy of the previous sequence number. Thus, in the above example, if the sequence of generated "fresh unique identifiers" is 0 1 2 3..., then the sequence $m_1\ m_2\ m_3\ m_4\ldots$ will be equipped with sequence numbers as follows $0\ m_1\ 1\ 0\ m_2\ 2\ 1\ m_3\ 3\ 2\ m_4\ldots$. Also here, a receiver can easily detect whether a message is missing, even if the sequence of identifiers is not known in advance.

The predicate *fresh* can be implemented by guessing an arbitrary identifier, and thereafter checking whether it already exists in the channel (this can be done by receiving all messages and then retransmitting them again).

The previous undecidability result illustrates a limitation of the approach of [4,22] to deciding the reachability problem by using a preorder on global states which is monotonic and is a well quasi-ordering, as in Section 3.1. The limitation is that an unbounded data domain with two ordering relations usually cannot be equipped with a useful well quasi-ordering. In the current case, the data elements are ordered both in channels and by the equality/inequality relation. These two orderings create sufficiently much structure that the natural ordering between states is not a well quasi-ordering. Symbolic backward reachability analysis can still be performed, but it is not guaranteed to terminate.

5.2 Unordered Channels

Let us now consider the control state reachability problem for unordered (lossy or nonlossy) channels. We shall extend the symbolic backward reachability analysis of Section 3.1 to the program model in this section, using a symbolic representation introduced in [6] and [7].

We define an ordering $\preceq$ on global states, so that the conditions of monotonicity and well quasi-orderedness are preserved. The key property of $\preceq$ is that if $\langle s,\sigma,w\rangle \preceq \langle s',\sigma',w'\rangle$, then all guards which are satisfied in state $\langle s,\sigma,w\rangle$ are also satisfied in $\langle s',\sigma',w'\rangle$. Intuitively, this means that $\langle s,\sigma,w\rangle$ is "isomorphic" to a state obtained by deleting zero or more messages from the channels in $\langle s',\sigma',w'\rangle$. Formally, we say that $\langle s,\sigma,w\rangle \preceq \langle s',\sigma',w'\rangle$ if $s = s'$ and if there is an injection $h : \mathcal{D} \mapsto \mathcal{D}$ on the domain of data values which preserves the ordering $<$ such that $\sigma' = h \circ \sigma$ and such that $w(c)(d) \leq w'(c)(h(d))$ for each channel c and data element d (i.e., if in $w(c)$ we replace each occurrence of d by $h(d)$ then we obtain a subbag of $w'(c)$). One can prove that any program is monotonic with respect to $\preceq$ and that $\preceq$ is a well quasi-ordering. Hence for this class of programs, control state reachability can be decided using symbolic backward reachability analysis.

Example. As an example of a protocol in this model, we consider again the leader election protocol. Let us model the algorithm for N processes, numbered in order from 0 to $N-1$. There are N channels $c_0,\ldots,c_{N-1}$, where channel c_i transmits messages from process $i-1$ to process i (arithmetic modulo N). Each process i as two local variables id_i and min_i, and will run the protocol in Figure 3. Correctness of the algorithm states that if all process reach the control state T, then min_i will have the same value in all processes. By instantiating

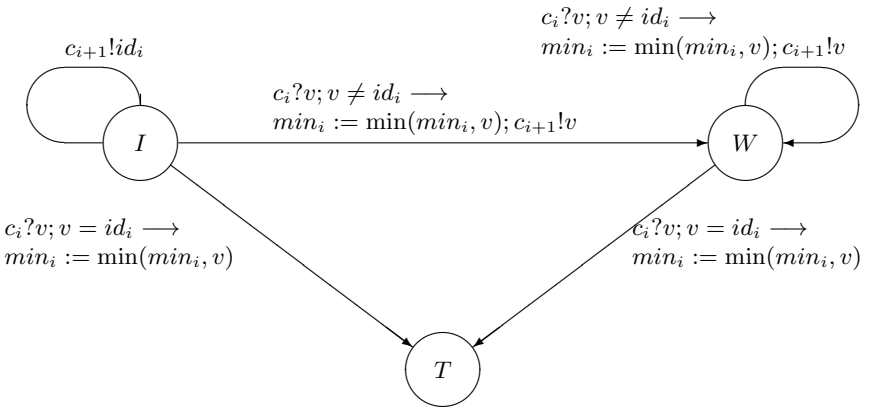


Fig. 3. Leader Election Protocol for Any Number of Processes

the above model to any particular number N of processes, we obtain a program in the model of this section. For each value of N, correctness can be expressed by requiring that the set of global states where all processes are in control state T and two processes disagree on the value of min_i is unreachable. For the case $N = 4$, the set can be represented symbolically as

$$\langle T, T, T, T, \phi \rangle$$

where ϕ is a formula stating that there are two processes that disagree on the value of min_i.

6 Conclusion

In this paper, we have surveyed some existing techniques for symbolic representation of unbounded channels in reachability analysis. We first considered the case where channels carry messages from a finite alphabet. We also briefly considered the case where messages are from an infinite set. It was shown that decidability may get lost in some models, but that in other frameworks it appears possible to incorporate this case into automated verification. More work seems necessary in order to understand whether and how messages from "interesting" data domains can be handled in general. Furthermore, it would be interesting to know whether protocols, such as the one in Section 5.2, can be verified parametrically for all values of N.

Acknowledgments. This work draws upon past and present collaboration, in particular with Aurore Annichini, Ahmed Bouajjani, Karlis Čerāns, Elena Fersman, Purushothaman Iyer, Mats Kindahl, Marcus Nilsson, Aletta Nylén, Yih-Kuen Tsay, and Wang Yi. Thanks to Pritha Mahata for valuable comments on drafts of the paper.

References

1. Parosh Aziz Abdulla, Aurore Annichini, and Ahmed Bouajjani. Algorithmic verification of lossy channel systems: An application to the bounded retransmission protocol. In *Proc. TACAS '99*, volume 1579 of *LNCS*, 1999.
2. Parosh Aziz Abdulla, Luc Boasson, and Ahmed Bouajjani. Effective lossy queue languages, 2001. To appear in Proc. ICALP '2001: 28^{th} International Colloquium on Automata, Lnaguages, and Programming.
3. Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 305–318, 1998.
4. Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. LICS' 96* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
5. Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
6. Parosh Aziz Abdulla and Bengt Jonsson. Verifying networks of timed processes. In Bernhard Steffen, editor, *Proc. TACAS '98*, volume 1384 of *LNCS*, pages 298–312, 1998.
7. Parosh Aziz Abdulla and Aletta Nylén. Timed Petri nets and BQOs, 2001. To appear in Proc. ICATPN'2001: 22nd Int. Conf. on application and theory of Petri nets.
8. T.P. Blumer and D.P. Sidhu. Mechanical verification of communication protocols. *IEEE Trans. on Software Engineering*, SE-12(8):827–843, Aug. 1986.
9. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Alur and Henzinger, editors, *Proc.* 8^{th} *Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer Verlag, 1996.
10. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the Fourth International Static Analysis Symposium*, LNCS. Springer Verlag, 1997.
11. A. Bouajjani and P. Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. In *Proc. ICALP '97*, volume 1256 of *Lecture Notes in Computer Science*, 1997.
12. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 2(5):323–342, April 1983.
13. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. LICS' 90,* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1990.
14. Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 10 January 1996.
15. A. Choquet and A. Finkel. Simulation of linear FIFO nets having a structured set of terminal markings. In *Proc.* 8^{th} *European Workshop on Applications and Theory of Petri Nets*, 1987.
16. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.
17. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Proc. CSL'99*, 1999.

18. E.A. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS' 98* 13^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.
19. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS' 99* 14^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1999.
20. A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89:144–179, 1990.
21. A. Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3), 1994.
22. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere. Technical Report LSV-98-4, Ecole Normale Supérieure de Cachan, April 1998.
23. M. Gouda. Closed covers: to verify progress for communicating finite state machines. *IEEE Trans. on Software Engineering*, SE-10(6):846–855, Nov. 1984.
24. M.G. Gouda, E.M. Gurari, T.-H. Lai, and L.E. Rosier. On deadlock detection in systems of communicating finite state machines. *Computers and Artificial Intelligence*, 6(3):209–228, 1987.
25. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
26. R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and Systems Sciences*, 3(2):147–195, May 1969.
27. K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1-2), 1997.
28. G. LeLann. Distributed systems – towards a formal approach. In B. Gilchrist, editor, *IFIP77*, pages 155–160. North-Holland, 1977.
29. R. Mayr. Undecidable problems in unreliable computations. In *Theoretical Informatics (LATIN'2000)*, number 1776 in Lecture Notes in Computer Science, 2000.
30. J.K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Protocol Specification, Testing, and Verification VII*, May 1987.
31. W. Peng and S. Purushothaman. Data flow analysis of communicating finite state machines. *ACM Trans. on Programming Languages and Systems*, 13(3):399–442, July 1991.
32. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *LNCS*, pages 337–352. Springer Verlag, 1982.
33. H. Rudin, C. West, and P. Zafiropulo. Automated protocol validation - one chain of development. In *Proc. Computer Network Protocols Symposium Liege*, 1978.
34. A.P. Sistla and L.D. Zuck. Automatic temporal verification of buffer systems. In Larsen and Skou, editors, *Proc. Workshop on Computer Aided Verification*, volume 575 of *LNCS*. Springer Verlag, 1991.
35. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.
36. C.H. West. Automated validation of a communications protocol: the ccitt x.21 recommendation. *IBM Journal on Research and Development*, 22(1), Jan. 1978.
37. Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.
38. Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *LNCS*, pages 88–97, Vancouver, July 1998. Springer Verlag.

Bigraphical Reactive Systems

Robin Milner

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

Abstract A notion of *bigraph* is introduced as a model of mobile interaction. A bigraph consists of two independent structures: a *topograph* representing locality and an *edge net* representing connectivity. Bigraphs are equipped with reaction rules to form *bigraphical reactive systems* (BRSs), which include versions of the π-calculus and the ambient calculus. A behavioural theory is established, using the categorical notion of *relative pushout*; it allows labelled transition systems to be derived uniformly for a wide variety of BRSs, in such a way that familiar behavioural preorders and equivalences, in particular bisimilarity, are congruential. An example of the derivation is discussed.

1 Introduction

It is nearly forty years since Petri produced the first substantial model of concurrent computation, and it was a graphical model [14]. Since that time a great many models have been studied. They are not always graphical, but the spatial metaphor is never far away; we often use terms like linkage, location, mobility, and so on. As it was for Petri, it remains a challenge for us to deploy spatial intuition but to retain rigour. This challenge grows as mobility grows in importance.

About ten years ago the author began the study of action calculi [10], an algebraic theory which allows graphs to reconfigure themselves, and which serves as a common frame for a variety of concurrency models, including Petri nets and the π-calculus; each model can be specified in the frame by a signature (a set of node types) and a set of reaction rules. Such a general framework yields understanding of the family of models, and of their differences. What it must also do is to provide a central theory which can be specialised to each individual model. The author now believes that such a theory is best achieved by treating graphs as the primary mathematical objects, not only as a means to visualise or present an algebraic or other theory. That belief lies behind the work reported here. The present model advances from action calculi in two ways: in graphical structure, and in behavioural theory. It will be presented in full detail in a forthcoming technical report (Milner [11]).

Graphical structure The first advance is in the form of graph on which the model is based. The model introduced here treats the *locality* and *connectivity* of agents – their nesting and wiring – as orthogonal, coordinated only by the

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 16–35, 2001.
© Springer-Verlag Berlin Heidelberg 2001

nodes of the graph. Hence the term *bigraph*; each graph is the superposition of two graphical structures. Moreover, the linkage of nodes is undirected and unconstrained.

The model inherits many features from action calculi. In particular, a node of a graph represents a variety of things, for example a π-calculus input term $x(y).P$, a "solution" of the Chemical Abstract Machine [1], an administrative region, or a host machine running several processes. A node not only carries ports, linked by arcs other ports, but also may enclose other nodes similarly linked. However, in action calculi there are constraints both on the arcs themselves (arcs are directed, and an output port carries exactly one outgoing arc) and on the relation between arcs and nesting (a node may be linked to siblings, parents, aunts and uncles but not to cousins in the nesting hierarchy). These constraints in action calculi make sense from the algebraic viewpoint adopted there. But in bigraphs, by removing the constraints, we not only to model a broader class of systems but also achieve a tractable behavioural theory.

Several influences have led to this treatment.

(1) The mobile ambients of Cardelli and Gordon [2] have emphasized the value of dealing with mobility in terms of (physical) location, rather than coding it in terms of wiring (as can be done to some extent in the π-calculus). Original action calculi can model ambients as originally defined, but not certain natural developments of the ambient notion.

(2) The fusion systems of Gardner [4], which evolved from action calculi, represent a process-calculus framework whose graphical form is implicit in its process-algebraic formulation. Two of its innovations are adopted in the present work; the unconstrained connectivity of the kind mentioned above, and the explicit fusions – here called aliases – of Gardner and Wischik [5] (developed from the fusion calculus of Parrow and Victor [13]). These authors are further developing a calculus of fusion graphs.

(3) It may be argued that to allow arcs to link nodes which are distant cousins, i.e. enclosed within distinct parent nodes arbitrarily far apart in the nesting structure, is contrary to reality. But we wish to model not only the *reality*, e.g. how communication is implemented, but also the *fiction* – e.g. "action-at-a-distance" – which the world wide web permits us to adopt. By embracing both views in the same model, one can hope – for example – to validate complex communication protocols in a mobile environment. This is well-argued by Wojciechowski and Sewell in their Nomadic Pict [16].

(4) A strong motivation has been to model fully-fledged mobile interaction, as increasingly found on the internet. A certain complexity is essential in a model which will support the analysis of such real systems. By treating locality and connectivity as independently as possible, it appears that the model can nevertheless remain mathematically tractable.

Behavioural theory In process calculi, activity is often first expressed as *reactions* of the form $a \longrightarrow\!\triangleright\, a'$, where a and a' are agents, and then refined or extended somehow to *labelled transitions* of the form $a \xrightarrow{\lambda}\!\triangleright\, a'$, where the label

λ is drawn from some vocabulary expressing the interactions which an agent may perform with its environment. Transitions (we henceforth omit "labelled") have the advantage that they support the definition of behavioural preorders and equivalences, such as traces, failures and bisimilarity. This kind of behavioural theory has been successful; but for each calculus one may ask where the labels come from. Typically they are simple, but not always atomic; in the π-calculus we have transitions like $a \xrightarrow{\nu x.\overline{y}\langle x\rangle} a'$, meaning that a can send a new private name x along the channel y. For action calculi, but more generally for any *reactive system* (under a precise definition of this concept), we may ask whether somehow these labels can be *derived* uniformly from any given set of reaction rules $r \longrightarrow r'$. A natural approach is to take the labels to be (a special class of) contexts; then $a \xrightarrow{F} a'$ implies the reaction $F \circ a \longrightarrow a'$, i.e. in the context F, a can react to become a'. But we don't want *all* contexts as labels. How to find a suitably minimal set has been open for some six years. Sewell [15] was able uniformly to derive satisfactory context-labelled transitions for parametric term rewrite systems with parallel composition and blocking, and showed bisimilarity to be a congruence. It remained a problem to do it for reactive systems dealing with connectivity. In recent work we have achieved a uniform derivation for all reactive systems satisfying a certain condition. We arrive at quite tractable transition systems for which, moreover, bisimilarity and other familiar behavioural preorders and equivalences are *guaranteed* to be congruences. The first results of this kind appear in Leifer and Milner [9]; these were extended and refined in Leifer's PhD Dissertation [8], which is the most comprehensive reference.

The notion of *reactive system* is defined categorically, and the condition required is that, in the appropriate precategory, sufficient *relative pushouts* (RPOs) exist. In a pilot study, Cattani *et al* [3], we showed that this condition holds for certain action calculi involving no nesting of nodes. We believe that the result extends to calculi with nesting, but verifying the condition becomes uncomfortably hard. The pilot study led us to expect our RPO condition to be met in interesting cases, but also led us to believe that it would be hard to verify for nested systems in which locality and connectivity are interdependent.

We come now to the second advance represented by bigraphs: their behavioural theory – being simpler – has been further developed. In fact, as reported here, the RPO condition holds for a wide class of bigraphical reactive systems. The paper includes a simple example of a derived congruential transition system.

Notation We use corresponding lower case letters for members of sets: $v \in V$, $x \in X$, $p \in P$ and so on. We write $X \uplus Y$ for the union of sets known or assumed to be disjoint. We assume a fixed representation of disjoint sums; for example, $X + P + Y$ means $(\{0\} \times X) \uplus (\{1\} \times P) \uplus (\{2\} \times Y)$, and $\sum_{v \in V} P_v$ means $\bigcup_{v \in V} (\{v\} \times P_v)$. We write $R^{\equiv}$ for the smallest equivalence containing a relation R, and $\equiv_0 \sqcup \equiv_1$ for the smallest equivalence including $\equiv_0$ and $\equiv_1$. If the equivalence $\equiv$ is over $X \uplus S$ then $\equiv \backslash X$ is its restriction to S.

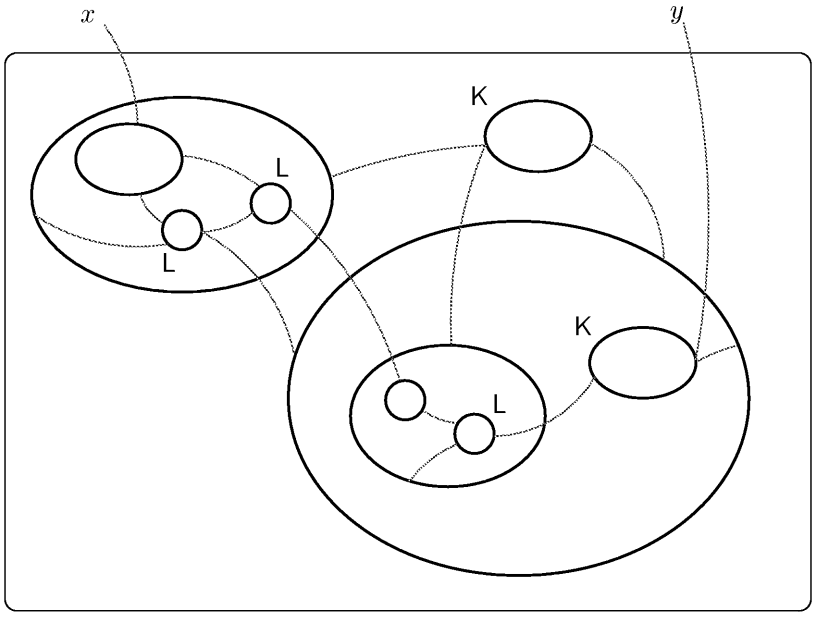


Figure 1: An example of a bigraph

2 Bigraphs in action

In this section we illustrate bigraphs informally, with examples which show the kinds of system they can represent, and the kind of mobility that they can model. Along the way we introduce, again informally, a simple term language for describing bigraphs. At the end of this section the reader will find a brief summary of the remainder of the paper.

Figure 1 shows an example of a bigraph. It has *nodes* (the ovals and circles) which support two kinds of structure; hence the term "bigraph". First, nodes may occur inside other nodes, so a bigraph has depth; since a node represents locality, in either a concrete or an abstract sense, we call this nesting structure of a bigraph its *topograph*. Second, nodes may be linked by *edges*, represented here by thin lines which may fork; we call this linkage structure of a bigraph its *edge net*. To each node is assigned a *control*, such as K or L, which tells us what kind of node it is. Each control has an *arity* ≥ 0; for example, L has arity three, so each L-node has an ordered set of three *ports*, at each of which an edge may impinge. It does not matter whether they impinge from inside or from outside the node. The diagram also shows the use of *names* x and y; such names allow a bigraph to be linked into larger bigraph.

The topograph and the edge net of any bigraph are coordinated by a node set, but are otherwise independent structures. But the *dynamics* of bigraphs, i.e. the reactions which may occur, depend upon both structural components; for

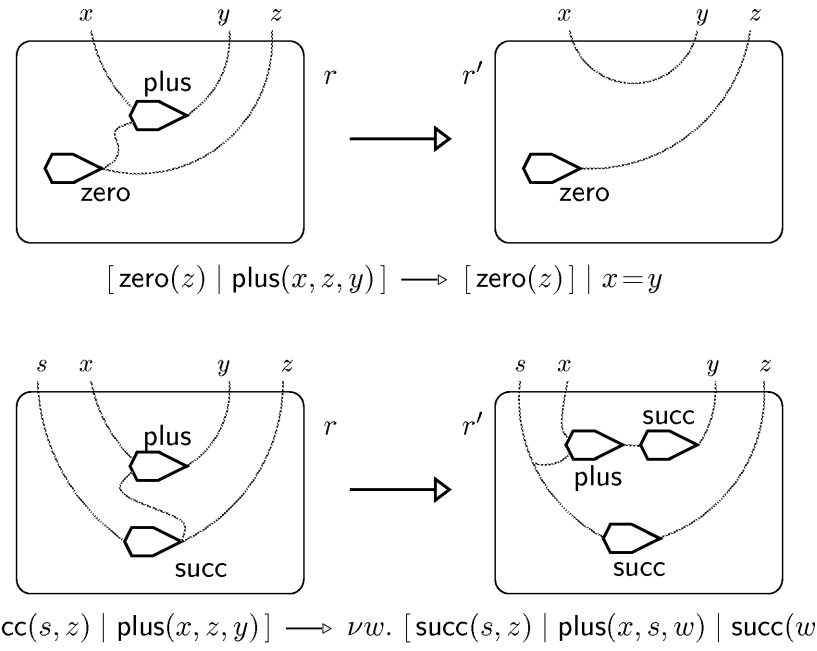


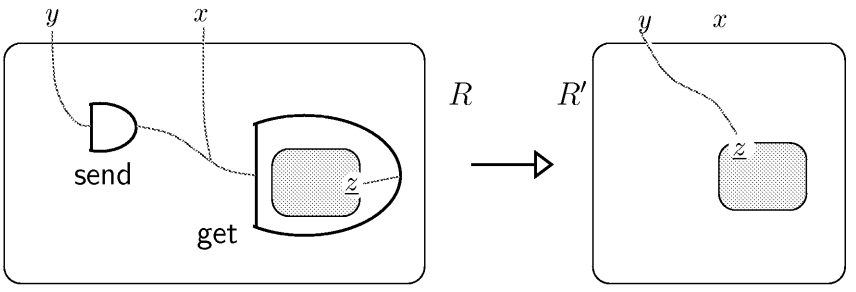
Figure 2: Two reaction rules for arithmetic

the precondition for any reaction is the presence of a certain pattern of nesting and linkage. A node in a bigraph may represent a great variety of computational objects: a physical location, an administrative region, a data constructor, a π-calculus input guard, an ambient, a cryptographic key, a message, a replicator, and so on. Its behaviour in any such role is determined by one or more *reaction rules*. Any control K can be specified as either *reactive* or *non-reactive*; in the latter case reactions cannot occur inside a K-node. (The term "reactive" pertains to what happens inside the node, not how the node itself may react.)

We now give some typical reaction rules. Each rule consists of a precondition and a postcondition, which we call a *redex* and *reactum*. A *reaction* consists of the replacement of a redex occurring in a bigraph by the corresponding reactum; we shall make precise what we mean by "occurrence" and "replacement".

Example 1 (reactions for arithmetic) Our first example, shown in Figure 2, involves no nesting. Note that the graphs r, r' (redex and reactum) have a name-set X as outer interface with their environment; here X takes the respective values $\{x, y, z\}$ and $\{s, x, y, z\}$. The shape of nodes is immaterial, except that if a control has arity more than 1 then its shape should not have rotational symmetry, so that the order of ports is unambiguous.

The two reaction rules use controls zero, succ and plus. They say, roughly, $0 + x \longrightarrow x$ and $s' + x \longrightarrow (s + x)'$ (where a prime here means successor). This strongly resembles the interaction nets of Lafont [7], but there is "sharing" in



$$[\,\mathsf{send}(y,x) \mid \mathsf{get}(x,\underline{z})[-]\,] \longrightarrow\!\!\triangleright\ [\,-\,] \mid y{=}\underline{z}$$

Figure 3: Reaction rule for the π-calculus

the sense of Hasegawa's sharing graphs [6]. In both cases one argument of plus is shared via the name z; we envisage a context around the redex which uses the argument for some other purpose.

In the first rule, the reactum makes y an *alias* for x. This is essentially an "explicit fusion" in the sense of Gardner and Wischik [5]; their calculus of explicit fusions was developed from the fusion calculus of Parrow and Victor [13] and from action calculi [10], and has guided the present development. The reactum in the second rule illustrates the use of a closed (i.e. unnamed) edge – between the plus and succ nodes.

In both cases, terms which define the redex and reactum are shown. Such terms can be used to define any bigraph. They reveal how to think of building a bigraph; its body consists of a parallel composition of *molecules* (one for each node) and "holes", with square brackets for grouping. The subterm for a molecule indicates the edges impinging on it by suitable identifiers. The term language is described at the end of this section. The notion of molecule stems from action calculi [10]; but the use here is more accurately that of fusion systems (Gardner [4]), in that edges are undirected hyperedges. As in that work, the only name-binding mechanism is the restriction operator ν appearing in the term language. Here it is used in the second arithmetic rule to describe a closed edge. ∎

Our next three examples illustrate more subtle uses of bigraphs in defining distributed reaction in mobile systems. Parametric redexes are used, following Sewell's study [15] of deriving labelled transition rules from reaction rules.

Example 2 (reaction in the π-calculus) Our second example (Figure 3) represents the familiar reaction rule of the asynchronous π-calculus (without summation)

$$x\langle y\rangle \mid x(z).P \longrightarrow\!\!\triangleright\ \{y/z\}P\ .$$

To present this version of π-calculus in terms of bigraphs we need two controls send and get, both of arity 2. Since the rule is parametric, the redex bigraph

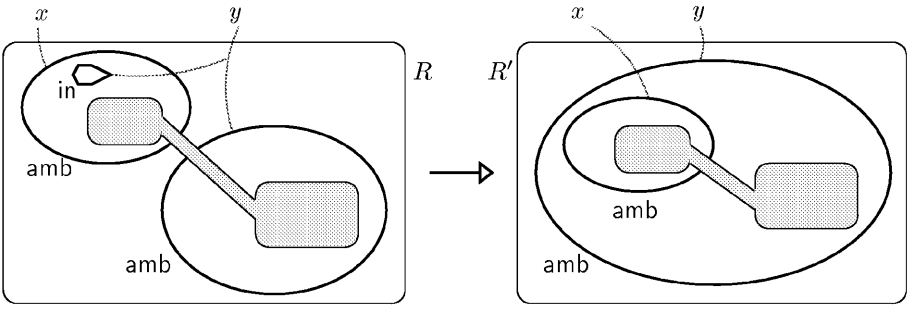


Figure 4: Reaction rule for the ambient calculus

is now a *context* R with both an *inner* and an *outer* interface as shown. Both interfaces have a multiplicity 1 (in this case – we discuss multiplicities later) and a name-set. For consistency with the π-calculus, which prevents reaction inside an input guard, we declare the get control to be non-reactive.

The name-set of the inner interface, at the hole inside the redex's get-node, contains the name z; this is to say that the outer interface of the agent parameter must contain z, and R will link it to the get-node. Our rule is polymorphic in this interface; it may contain any set W of other names distinct from x, y and z, which are understood to be exported by R and R' to their outer interface – i.e. the reaction rule is transparent to W. Names of the inner interface are underlined to distinguish them from those of the outer, since the two name-sets may not be disjoint. We shall refer to the names of inner interface as *conames*.

In applying the rule, one must be able to choose arbitrary instances for the names x, y and W. In fact, an arbitrary substitution for these names can be effected by the surrounding context with the help of *co-aliases*, dual to the notion of alias we have mentioned. We say more about them later. ∎

Example 3 (reaction in the ambient calculus) In the ambient calculus of Cardelli and Gordon [2], one of the primitive forms of reaction is the movement of one ambient inside another.

Figure 4 shows how bigraphs may represent such a rule. We use two controls: amb for an ambient, and in for a "command" to move its parent ambient somewhere else. In contrast with get in the π-calculus we declare amb to be reactive, since ambients are intended only to localize activity, not to inhibit it. In the redex R, the intent of the in command is to move the ambient named x inside the one named y. The rule has two parameters – the other contents of the x-ambient and the contents of the y-ambient – so the redex has two holes. These parameters, though occupying distinct sites, may be linked by edges. A conversation between these "two" agents, active at the time our reaction occurs,

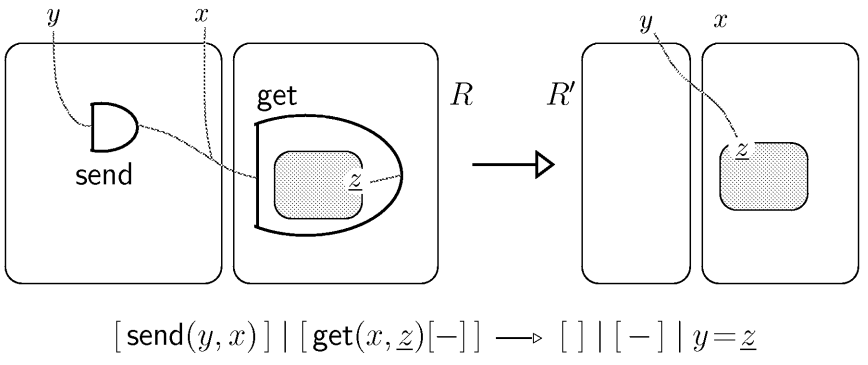


Figure 5: Reaction rule for the "elastic" π-calculus

should not be interrupted by the movement.

Thus we think of R as having a multiple site, and we expect the inhabitant to be a bigraph of multiplicity 2. As in Example 2, this bigraph may have an arbitrary name set Z, and since locality and linkage are independent these names pertain to the whole (multiple) site, not to one or other component site. In the terms describing the redex and reactum, these component sites are denoted by $-_0$, $-_1$. (Our use of square brackets is not meant to match how the ambient calculus uses them.)

Note that the redex and reactum of this rule have multiplicity 1. Our next example illustrates a bigraph with multiplicity 2, which may indeed fit into the multiple hole of the present example. ∎

Example 4 (reaction in the "elastic" π-calculus) In the reaction rule of Example 2 the redex and reactum have multiplicity 1; this means that the rule applies only when the send and get molecules are co-located. To put it another way, any context in which we place the redex will have these two nodes topographically adjacent. To allow a context to site them apart, we just change the multiplicity of the redex and reactum, and make them linked bigraph-pairs, in which we may locate nodes and holes in whichever component we please.

This rule is interesting in the presence of one or more reactive controls, since such a control can be used to separate the components of our elastic redex but still allow it to react. We have already introduced amb as an example of such a control. By choosing Z in Example 3 (Figure 4) to be $\{x, y\} \uplus W$, we may compose the ambient redex with the elastic π-redex; we then obtain two interwoven but independent redexes, for which neither reaction precludes the other. This is not an unlikely occurrence in the internet, modelled at a suitable level of abstraction. ∎

In the course of examples we have introduced a term language for describing bigraphs. Later we shall give mathematical meaning to these terms. Here is how they are built:

- A *term* describing a bigraph $G : (m, X) \to (n, Y)$ consists of a set of local names written after ν, an ordered sequence of n regions, and an equivalence relation on $X + Y$. Only the sites $-_0, \ldots, -_{m-1}$ may occur in the term, each just once. ν may be omitted if there are no local names. The equivalence relation may be given as equations between names and/or conames.

- A *molecule* such as $\mathsf{amb}(x)[\mathsf{in}(y) \mid -_1]$ describes a node and its contents; it consists of a control, followed in parentheses by an ordered list of identifiers denoting the edge impinging on each port, followed by a region describing its contents, if any.

- A *region* is a parallel composition (in any order) of molecules and sites, in square brackets $[\cdot]$. A *site* is $-_i$ $(i \geq 0)$. One may write $-$ for $-_0$.

- An *identifier* is either a name x, or a coname $\underline{y}$, or a local name.

- The *local names* are distinct from X and Y, and can be re-named by alpha-conversion. If two names or conames are equivalent they denote the same edge, so either may be used in a molecule to denote that edge.

The remainder of the paper is organised as follows: Section 3 contains some mathematical background on reactive systems in general; Section 4 gives the main definitions of the structure of bigraphs; Section 5 develops their dynamic theory, based upon the two preceding sections; Section 6 makes a preliminary investigation of how the theory may be applied in practice; Section 7 concludes by listing a number of outstanding problems and tasks.

3 Theoretical background

In this section we define a general notion of reactive system; we also define our main technical device, the relative pushout.

Definition 1 (precategories) A *precategory* $\mathbf{A}$ is defined exactly as a category, except that composition $(\circ)$ is not always defined. Composition with the identities id is always defined, and $\mathsf{id} \circ A = A = A \circ \mathsf{id}$. For associativity, if $A_2 \circ A_1$ and $A_1 \circ A_0$ are defined then either both $A_2 \circ (A_1 \circ A_0)$ and $(A_2 \circ A_1) \circ A_0$ are undefined or both are defined and equal.

A *subprecategory* $\mathbf{B}$ of $\mathbf{A}$ is defined just as a subcategory; we take $A_1 \circ A_0$ to be defined in $\mathbf{B}$ exactly when it is defined in $\mathbf{A}$. We shall say that $\mathbf{B}$ is *closed under decomposition* if, whenever $A_1 \circ A_0$ is in $\mathbf{B}$ then so are A_0 and A_1.

An *agent (pre)category* $\mathbf{A}$ is a (pre)category with a distinguished object called the *origin*, which we shall denote by ϵ. Its objects $I, J, \ldots$ are called *interfaces*, and its arrows $A, B, \ldots$ are called *contexts*. Contexts with domain ϵ are called *agents*, and denoted by $a, b, \ldots$. ∎

We now introduce a kind of dynamical system.

Definition 2 (reactive system) A *reactive system* $\mathsf{R}_{\mathbf{A}}$ over an agent precategory $\mathbf{A}$ has, in addition,

- a set $\mathsf{Reacts} \subseteq \bigcup_I \mathbf{A}(0,I)^2$ of *reaction rules*;
- a subprecategory $\mathbf{D}$, the *reactive contexts*, closed under decomposition.

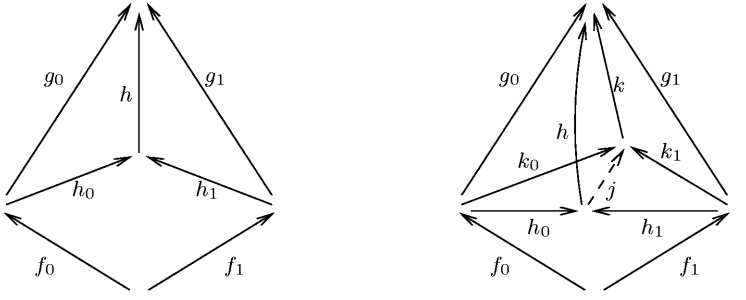
The *reaction relation* $\longrightarrow\!\!\triangleright$ over agents is defined as the least relation such that $D \circ r \longrightarrow\!\!\triangleright D \circ r'$ for all reaction rules (r, r') and reactive contexts D for which the compositions are defined. ∎

We now turn to relative pushouts.

Notation We shall frequently use $\vec{f}$ to denote a pair f_0, f_1 of arrows in a precategory. If, for example, the two arrows share a domain W and have codomains X_0, X_1 we write $\vec{f} : W \to \vec{X}$.

Definition 3 (bound, consistent) Given two pairs of arrows $\vec{f} : W \to \vec{X}$ and $\vec{g} : \vec{X} \to Z$ sharing domain and codomain respectively, if $g_0 \circ f_0 = g_1 \circ f_1$ then $\vec{g}$ is a *bound* for $\vec{f}$. If such a pair $\vec{f}$ has any bound, then it is *consistent*. ∎

Definition 4 (relative pushout) In a precategory, let $\vec{g} : \vec{X} \to Z$ be a bound for $\vec{f} : W \to \vec{X}$. An *RPO-candidate* for $\vec{f}$ w.r.t. $\vec{g}$ is a triple $(\vec{h}, h)$ of arrows such that $h_0 \circ f_0 = h_1 \circ f_1$ and $h \circ h_i = g_i$ $(i = 0, 1)$. A *relative pushout* (*RPO*) for $\vec{f}$ w.r.t. $\vec{g}$ is a candidate $(\vec{h}, h)$ such that for any candidate $(\vec{k}, k)$ there is a unique arrow j such that $j \circ h_i = k_i$ $(i = 0, 1)$ and $k \circ j = h$. ∎



The more familiar notion, a pushout, is just an RPO for $\vec{f}$ which is independent of the bound $\vec{g}$. (In bigraphs we shall find that RPOs exist in cases where there in no pushout.) Thus, a pushout is a *minimum* bound, while an RPO provides a *minimal* bound w.r.t. a given bound $\vec{g}$. The following shows that the notion of minimality can be defined without prior mention of a bound $\vec{g}$:

Definition 5 (idem pushout) In a precategory, if $\vec{f} : W \to \vec{X}$ is a pair of arrows with common domain, then a pair $\vec{h} : \vec{X} \to Y$ is an *idem pushout* (*IPO*) for $\vec{f}$ if $(\vec{h}, \mathsf{id}_Y)$ is an RPO for $\vec{f}$ w.r.t. $\vec{h}$. ∎

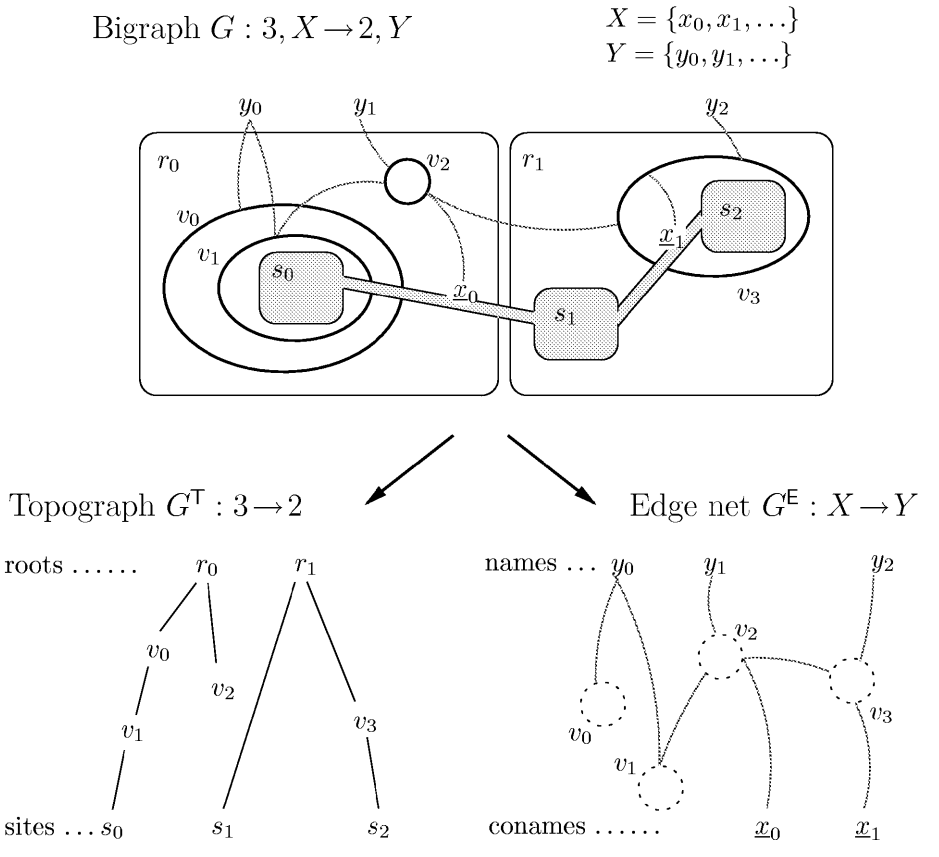


Figure 6: Resolving a bigraph into a topograph and an edge net

In a later section, in defining a transition $a \xrightarrow{F} a'$, we shall require that there be a reaction rule (r, r') and a context D such that F, D is an IPO for a, r.

RPOs and IPOs in categories have a few vital properties on which our work relies. These are established in Leifer [8] or Leifer and Milner [9]; they will be recapitulated in Milner [11].

4 Defining bigraphs

We are now ready to embark on the theory of bigraphs. First we need a repertoire of controls, as illustrated in our examples:

Definition 6 (signature) A *signature* $\mathcal{K}$ is a set of *controls*, together with an assignment to each control $K \in \mathcal{K}$ of a natural number $ar(K)$, its *arity*. ■

Assume a fixed but unspecified signature. We shall define bigraphs top-down; that is, we first define how a bigraph is built from its two structural components, and then define those components themselves. We thus obtain a precategory of bigraphs, built upon those for topographs and edge nets. In each case it is routine to verify that composition has the right properties.

Definition 7 (bigraph) A *bigraph* over the signature $\mathcal{K}$ takes the form $G = (V, ctrl, G^{\mathsf{T}}, G^{\mathsf{E}}) : I \to J$ where: $I = (m, X)$ and $J = (n, Y)$ are its *inner* and *outer interfaces*, each consisting of an *multiplicity* (a natural number) and a finite *name set*; V is a set of *nodes*; $ctrl : V \to \mathcal{K}$ is the *control function* assigning a control to each node; $G^{\mathsf{T}} = (V, ctrl, prt) : m \to n$ is a *topograph* (Definition 8) which inherits G's control function; and $G^{\mathsf{E}} = (P, \equiv) : X \to Y$ is an *edge net* (Definition 10) whose inner ports are $P \triangleq \sum_{v \in V} ar(ctrl(v))$. ∎

Thus each interface is decomposed into its components, one for the topograph and the other for the edge net. The control function *ctrl* defines the inner ports of the edge net, since the ports of a node are indexed by the arity of its control. Figure 6 shows how a bigraph is resolved into its components. As we shall see below, this resolution bases bigraphs upon two well-studied mathematical structures: trees for topographs and equivalence relations for edge nets.

Definition 8 (topograph) A *topograph* $A = (V, ctrl, prt) : m \to n$ has an *inner multiplicity* m, and an *outer multiplicity* n, both natural numbers; a finite set V of *nodes*; a control function $ctrl : V \to \mathcal{K}$ as for a bigraph; and a *parent function* $prt : m \uplus V \to V \uplus n$ which is *acyclic*, i.e. such that $prt^k(v) \neq v$ for all $k > 0$ and $v \in V$. ∎

Acyclicity ensures that the parent function *prt* represents a forest of n unordered trees; n indexes the *roots* of the trees. Other vertices are either nodes $v \in V$, or *sites* indexed by m; a site may only occur as a leaf. Sites and roots provide the means of composing the forests of two topographs; each root of the first is planted in a distinct site of the second. Formally:

Definition 9 (precategory of topographs) The agent precategory **Top** has natural numbers as interfaces, with origin 0, and topographs $A = (V, ctrl, prt) : m \to n$ as contexts. The composition $A_1 \circ A_0 : m_0 \to m_2$ of two topographs $A_i = (V_i, ctrl_i, prt_i) : m_i \to m_{i+1}$ $(i = 0, 1)$ is defined when the two node sets are disjoint; then $A_1 \circ A_0 = (V, ctrl, prt) : m_0 \to m_2$ where $V = V_0 \uplus V_1$, $ctrl = ctrl_0 \uplus ctrl_1$ and $prt = (\mathsf{Id}_{V_0} \uplus prt_1) \circ (prt_0 \uplus \mathsf{Id}_{V_1})$. The identity topograph at m is $\mathsf{id}_m \triangleq (\emptyset, \emptyset_{\mathcal{K}} \mathsf{Id}_m) : m \to m$. ∎

We now turn to edge nets.

Definition 10 (edge net) An *edge net* $A = (P, \equiv) : X \to Y$ has four components. Three are sets of *ports*: the *conames* X, the *names* Y and the *inner ports* P. The fourth component is an equivalence $\equiv$ upon $X + P + Y$. ∎

We rely upon our naming convention to allow us to write, for example, $x \equiv_A p$ instead of $\langle 0, x\rangle \equiv_A \langle 1, p\rangle$. However, when X and Y are not disjoint we may write $\underline{x}$ for a coname x. We shall abuse notation by writing just A to denote its own equivalence; for example xAp means $\langle 0, x\rangle \equiv_A \langle 1, p\rangle$.

Definition 11 (precategory of edge nets) The agent precategory **Edg** has name sets as interfaces, with origin $\emptyset$, and edge nets $A : (P, \equiv_A) : X \to Y$ as contexts. The composition $B \circ A : X \to Z$ of two edge nets $A = (P, \equiv_A) : X \to Y$ and $B = (Q, \equiv_B) : Y \to Z$ is defined when $P \cap Q = \emptyset$; its inner ports are then $P \uplus Q$, and its equivalence[1] is $(\equiv_B \sqcup \equiv_A)\backslash Y$. The identity on X is $\mathsf{id}_X : X \to X$ whose set of inner ports is empty, with the equivalence $\{(\underline{x}, x) \mid x \in X\}^{\equiv}$. ∎

We shall need a few properties of ports; recall that a port r may be a name, a coname or an inner port.

Definition 12 (port properties) A port r in A is *open* (*at* y) if rAy for some $y \in Y$; otherwise *closed*. A name $y \in Y$ is *idle* if $rAy \Rightarrow r \in Y$ for all ports r in A. y is an *alias for* $y' \in Y$ if $y \neq y'$ and yAy'. There are dual notions *co-open*, *co-idle* and *co-alias* involving the conames X. ∎

Aliases played a role in the arithmetic example of Section 2, where a reaction rule creates an alias in the reactum. They have no analogue for topographs. Although they are needed for some calculi, we shall here find it useful to study bigraphs without aliases (though we keep co-aliases). To this end, we now define

Definition 13 (alias-free) Denote by $\mathbf{Edg}_{\bar{\mathsf{a}}}$ the sub-precategory of **Edg** including only *alias-free* bigraphs – those whose edge nets contain no aliases. It is easily shown that alias-freedom is preserved by composition. ∎

We are now ready to define our main precategory.

Definition 14 (precategory of bigraphs) The agent precategories **Big** and $\mathbf{Big}_{\bar{\mathsf{a}}}$ of bigraphs have pairs $I = (m, X)$ as interfaces, with origin $\epsilon = (0, \emptyset)$, and bigraphs $G : (U, ctrl_G, G^{\mathsf{T}}, G^{\mathsf{E}}) : I \to J$ as contexts, where G^{E} has no aliases in the case of $\mathbf{Big}_{\bar{\mathsf{a}}}$. If $H : J \to K$ is another bigraph with node set V disjoint from U, then their composition is defined directly in terms of the composition of the components as follows:

$$H \circ G : I \to K \;\triangleq\; (V \uplus U, ctrl_H \uplus ctrl_G, H^{\mathsf{T}} \circ G^{\mathsf{T}}, H^{\mathsf{E}} \circ G^{\mathsf{E}}) : I \to K \;.$$

The identities are $\mathsf{id}_I \triangleq (\emptyset, \emptyset_{\mathcal{K}}, \mathsf{id}_m, \mathsf{id}_X) : I \to I$. ∎

We sometimes need to change the identity of nodes in a bigraph. For example, we cannot compose H with G unless their node sets V and U are disjoint. To make them so, we may convert H into H' by applying a bijection $V \rightleftarrows V'$; if we choose V' disjoint from V we can then form $H' \circ G$.

[1] This definition is slightly inaccurate, for the sake of brevity. Since the sets involved are not always disjoint, the fully accurate definition of $\equiv_{B \circ A}$ is as follows: first take the lub of the two equivalences on $X + P + Y + Q + Z$ induced by $\equiv_A$ and $\equiv_B$; then take the equivalence thus induced on $X + (P \uplus Q) + Z$.

Definition 15 (support translation) For a bigraph G and any bijection ρ whose domain includes the node set of G, the bigraph which results from renaming the nodes of G by ρ is denoted by $\rho \cdot G$, and called a *support translation* of G. Two bigraphs G and G' are *support equivalent*, $G \simeq G'$, if one is a support translation of the other. ∎

If we divide **Big** or $\mathbf{Big}_{\bar{\mathsf{a}}}$ by support equivalence we obtain a category, which we may call its *support quotient*. We shall later define dynamic behaviours in a way which is invariant under support translation, thus yielding dynamics also for the quotient category.

A crucial question for dynamics is whether RPOs exist in these precategories. The question remains open for **Big**; but for $\mathbf{Big}_{\bar{\mathsf{a}}}$ there is a positive answer. In fact, we answer the question independently for topographs and edge nets. For topographs RPOs always, exist; for edge nets, not quite always. The results and proofs will appear in full detail in Milner [11]. Putting them together, we get

Theorem 16 *In* $\mathbf{Big}_{\bar{\mathsf{a}}}$, *Let* $\vec{G} : W \to \vec{X}$ *be a pair of bigraphs such that no name* $x \in X_1$ *is idle in* G_1. *Let* $\vec{E}$ *be a bound for* $\vec{G}$. *Then there exists an RPO for* $\vec{G}$ *w.r.t.* $\vec{E}$.

A second important property is that (RPOs and) IPOs are preserved by support translation.

Proposition 17 (IPO sliding) *In* **Big** *or* $\mathbf{Big}_{\bar{\mathsf{a}}}$, *if* $\vec{H}$ *is an IPO for* $\vec{G}$ *and* ρ *is any bijection whose domain includes the node sets of* $\vec{G}$ *and* $\vec{H}$, *then* $\rho \cdot \vec{H}$ *is an IPO for* $\rho \cdot \vec{G}$.

5 Bigraphical reactive systems

We are now almost ready to define bigraphical reactive systems and their dynamics. First, as well as composition of bigraphs, we need a (partial) tensor product $\otimes$ for placing them side-by-side:

Definition 18 (tensor product) Given two interfaces $I_i = (m_i, X_i)$ $(i = 0, 1)$ with X_0, X_1 disjoint, we define their *tensor product*: $I_0 \otimes I_1 \triangleq (m_0 + m_1, X_0 \uplus X_1)$. When $I_0 \otimes I_1$ and $J_0 \otimes J_1$ are defined and the bigraphs $G_i : I_i \to J_i$ $(i = 0, 1)$ have disjoint node sets, then we define their *tensor product* $G_0 \otimes G1 : I_0 \otimes I_1 \to J_0 \otimes J_1$ by taking the unions of their control functions, parent functions and equivalences respectively. ∎

Subject to definedness, the tensor product satisfies the equations for a strict symmetric monoidal category.

We now specialise the definition of a reactive system (Definition 2) to bigraphs.

Definition 19 (bigraphical reactive system) A *bigraphical (reactive) system* (BRS) is a reactive system based upon **Big** or $\mathbf{Big}_{\bar{\mathsf{a}}}$ with some signature $\mathcal{K}$, such that

- the reaction rules are *closed under support translation*, i.e. if $(r, r') \in \mathsf{Reacts}$ then $(\rho \bullet r, \rho' \bullet r') \in \mathsf{Reacts}$;
- the reactive contexts are *closed under factorisation*, i.e. if $D \otimes D' \in \mathbf{D}$ then $D, D' \in \mathbf{D}$. ∎

We can now define transitions for these systems. A definition can be given for any reactive system, but we need to deploy the notion of support equivalence enjoyed by BRSs.

Definition 20 (transition) In any BRS, a (*labelled*) *transition* is a triple a, F, a', written $a \xrightarrow{F} a'$, such that there exist a reaction rule (r, r') and a reactive context D for which F, D is an IPO for a, r and $a' \simeq D \circ r'$. ∎

It follows from IPO sliding, and the fact that support translation preserves reaction, that it also preserves transition. This is important if we wish to define transitions in the support quotient category. To be precise:

Proposition 21 *In any BRS, let $a \simeq b$, $F \simeq G$ and $a' \simeq b'$. If $a \xrightarrow{F} a'$ and $G \circ b$ is defined, then $b \xrightarrow{G} b'$.*

Having defined transitions in a BRS, we can define behavioural equivalences in familiar ways. In particular:

Definition 22 (strong bisimilarity) In any BRS, *strong bisimilarity* $\sim$ is the largest symmetric relation such that if $a \sim b$ and $a \xrightarrow{F} a'$, then whenever $F \circ b$ is defined there exists b' such that $b \xrightarrow{F} b'$ and $a' \sim b'$. ∎

The only slight departure from the standard definition is the condition that $F \circ b$ should be defined (since we are working in a precategory). In previous work [9, 8] we have shown that, if a BRS has sufficient RPOs, then many behavioural preorders and equivalences based upon transitions are congruential. We now define "sufficient RPOs" and state the theorem for strong bisimilarity:

Definition 23 (redex RPOs) A BRS *has all redex-RPOs* if any pair of agents a, r, where r is the redex of a reaction rule, has an RPO w.r.t. any bound F, D. ∎

Theorem 24 *If any BRS has all redex-RPOs, then strong bisimilarity is a congruence.*

Now it can be argued that in **Big** or $\mathbf{Big}_{\bar{\mathsf{a}}}$, a reaction rule whose redex has an idle name leads to rather strange behaviour, unlikely to be met in applications. (Note that no redex in our examples has an idle name.) So, regarding a BRS

as unreasonable if any redex has an idle name, from Theorems 24 and 16 we deduce the corollary: *In any reasonable bigraphical reactive system based upon* $\mathbf{Big}_{\bar{\mathsf{a}}}$, *bisimilarity for the derived transition system is a congruence.*

These positive results for BRSs rest upon the existence of RPOs, Theorem 16. In practice, we want to know what these derived transitions are. To this end, the forthcoming technical report [11] gives a full characterisation of the IPOs for $\mathbf{Big}_{\bar{\mathsf{a}}}$, and this directly supplies the transitions. In this paper we confine ourselves to a simple example of a derived transition system; this is presented in the following section.

Discussion The work of this section has been in a *precategory* of bigraphs. But we have ensured that both reactions and transitions are respected by support translation ($\doteq$), and can deduce that bisimilarity is preserved by support translation. Thus, if we quotient $\mathbf{Big}_{\bar{\mathsf{a}}}$ by $\doteq$ we obtain a reactive system based upon a *category*, and we understand its dynamic theory too. This quotient, with fully defined composition, is amenable to *algebraic* theory. But the work of this section has been in the unquotiented BRS, because that is where we find the necessary categorical structure to develop the *dynamic* theory.

6 Applications

In this section we give some preliminary results of applying the theory explained above. First, we define some useful constructions; then we identify a general class of derived transitions which are superfluous; finally we study a simple example of derived transitions for a specific reaction rule.

We begin with some definitions, which explain the term language described and used in Section 2.

Definition 25 (molecule) A *molecular bigraph* $K_v(\xi_0, \ldots, \xi_{r-1})_m : (m, X) \to (1, Y)$ has one root, m sites and a single node v – parent of all the sites – whose control is K with arity r. The i^{th} inner port of v is linked to ξ_i, which is either a coname $\underline{x}$ ($x \in X$) or a name $y \in Y$. We call $K(\vec{\xi})_m \circ a$ a *molecule*; the hole is occupied by an agent. We call $K(\vec{\xi})_0$ an *atom*; the hole is null. ∎

Definition 26 (parallel composition) If two edge nets $\vec{A} : \vec{X} \to \vec{Y}$ have disjoint sets $\vec{V}$ of inner ports then their *parallel composition* $A_0 \mid A_1 : X_0 \cup X_1 \to Y_0 \cup Y_1$, with inner port set $V_0 \uplus V_1$ and equivalence $\equiv_{A_0} \sqcup \equiv_{A_1}$. We extend it to two bigraphs $G_i = (V_i, ctrl_i, G_i^{\mathsf{T}}, G_i^{\mathsf{E}}) : (m_i, X_i) \to (n_i, Y_i)$ with disjoint node sets as follows:

$$G_0 \mid G_1 \triangleq (V_0 \uplus V_1, ctrl_0 \uplus ctrl_1, G_0^{\mathsf{T}} \otimes G_1^{\mathsf{T}},\ G_0^{\mathsf{E}} \mid G_1^{\mathsf{E}}) : (m_0 + m_1, X_0 \cup X_1) \to (n_0 + n_1, Y_0 \cup Y_1)\ .$$ ∎

Parallel composition resembles the tensor product $\otimes$, but takes the *union* of domains and codomains; thus it may merge edges from G_0 and G_1. It is commutative and associative, with identity id_ϵ.

Definition 27 (merge) Define $1_{(m,X)} : (m, X) \to (1, X)$ to be the only bigraph with the given interfaces whose edge net is id_X. If G has codomain (m, X), we define $[G] \triangleq 1_{(m,X)} \circ G$, the *merge* of G; it has codomain $(1, X)$ and results from merging the m regions of G into one. ∎

We abbreviate $K_v(\vec{\xi})_m \circ G$ to $K_v(\vec{\xi})[G]$, and $K_v(\vec{\xi})_0$ to $K_v(\vec{\xi})$.

Definition 28 (name,coname,restrict) Define the bigraphs $x : \epsilon \to (0, \{x\})$ and $\underline{x} : (0, \{x\}) \to \epsilon$ to consist of a single name or coname. Then define the *restriction* $\nu x \triangleq \underline{x} \otimes \mathsf{id}_I$ for any interface I not involving x. ∎

We now ask: How essential are the transitions which we have derived? This paper has no space to characterise them; but we shall say what it means for a transition to be superfluous. Moreover we identify a class of superfluous transitions; it supports the intuition that each proper transition of an agent a should represent some contribution which a makes to a reaction.

Definition 29 (adequate transitions) Let T be a set of transitions. Define $\sim_T$ to be the largest symmetric relation between agents with like codomain such that if $a \sim_T b$, then for every transition $a \xrightarrow{F}\!\triangleright\, a'$ in T there exists b' such that $b \xrightarrow{F}\!\triangleright\, b'$ and $a' \sim_T b'$. Call T *adequate* if $\sim_T = \sim$. ∎

(Note especially that the transition of b, which matches the T-transition of a, need not itself be in T. This means that $\sim_T$ is in general not transitive.)

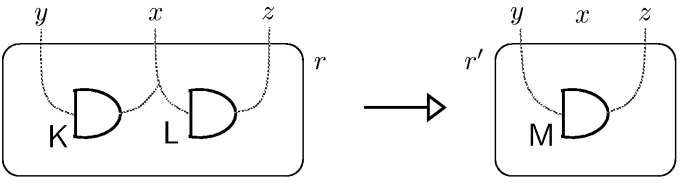
We now look at an important case. We may expect a transition $a \xrightarrow{F}\!\triangleright\, a'$ to be superfluous if "a contributes nothing" to the redex on which the transition is based; for then the entire redex is "provided by F". The tensor product allows us to make this intuition precise and justify it. First, our IPO characterisation shows that "tensorial" transitions exist; further analysis shows them to be superfluous:

Proposition 30 *In* $\mathbf{Big}_{\bar{\mathsf{a}}}$ *let* $a : \epsilon \to I$ *be an agent and* (r, r') *a reaction rule not involving names in* I. *Then* $a \xrightarrow{\mathsf{id}_I \otimes r}\!\triangleright\, a \otimes r'$ *is a transition.*

Definition 31 (tensorial transition) A transition from $a : \epsilon \to I$ is *tensorial* if, up to isomorphism, it takes the form $a \xrightarrow{\mathsf{id}_I \otimes r}\!\triangleright\, a \otimes r'$, where (r, r') is a reaction rule. ∎

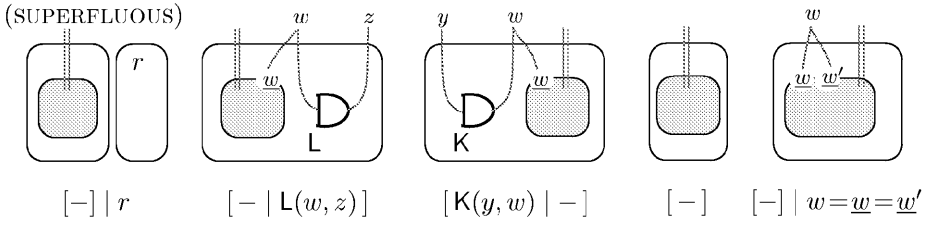
Proposition 32 *The set of non-tensorial transitions is adequate.*

Finally, let us exhibit the derived transitions for a simple specific BRS, motivated by the π-calculus. Take the control signature $\mathcal{K} \triangleq \{\mathsf{K}, \mathsf{L}, \mathsf{M}\}$, all with arity two; take all contexts to be reactive, and let there be a single reaction rule:



$$[\,\mathsf{K}(y,x) \mid \mathsf{L}(x,z)\,] \longrightarrow\!\triangleright\, \mathsf{M}(y,z)$$

We shall now classify the transitions of any agent $a : \epsilon \to I$ with non-empty node set and multiplicity 1; without loss of generality we assume $x, y, z \notin I$. The classification is based upon the extent of node-sharing between a and r. (Note that our term language does not specify node-identity. Working in $\mathbf{Big}_{\bar{\mathsf{a}}}$, we have to take care of it; this is analogous to identifying *occurrences* of subterms in the λ-calculus, where they are important in examining properties of term reduction.) This yields five kinds of transitions for a; four of them constitute an adequate set. The transition labels – proper and improper – are shown in the following diagrams. Any names in I not shown in the diagrams are exported to the outer interface; we have indicated this by a double line.



The transitions arise as follows:

No nodes shared There is one transition: $a \xrightarrow{[-]|r}\!\triangleright\, a \mid r'$. It may also be written $a \xrightarrow{\mathsf{id}_I \otimes r}\!\triangleright\, a \otimes r'$, which we recognise as tensorial – hence superfluous. Note especially that this is the only label with two regions; the reaction $r \longrightarrow\!\triangleright\, r'$ occurs "alongside" a without merging with it.

One node shared If the K-node is shared, then its right-hand port must be open a, at w say, and the unique IPO of a, r equates w with x and contains just an L-atom. The transition takes the form $a \xrightarrow{[-|\mathsf{L}(w,z)]}\!\triangleright\, a'$. Similarly, if the L-node is shared there is a transition $a \xrightarrow{[\mathsf{K}(y,w)|-]}\!\triangleright\, a'$.

Both nodes shared There are two cases. If the K- and L-nodes are already linked in a, then there is a transition of the form $a \xrightarrow{[-]}\!\triangleright\, a'$; this corresponds to a τ transition in CCS. Otherwise the two nodes are open in a, say at w and w', and there is a transition of form $a \xrightarrow{[-]|w=\underline{w}=\underline{w}'}\!\triangleright\, a'$; in explicit fusions [5] this is a fusion supplied by the context.

The situation is more complex with a parametric rule. For example, instead of our "ground" rule (r, r') we may have a parametric rule (R, R') in which, say, the L-node contains a site. (Our rules for the π-calculus and the ambient calculus are also examples.) We then have to specify what set of ground rules is signified by the parametric rule (R, R'). A natural candidate is the set of pairs $((R \otimes \mathsf{id}_I) \circ d, (R' \otimes \mathsf{id}_I) \circ d)$ for any interface I and agent d; this achieves the polymorphism mentioned in Example 2. If in our example we allow the L-node to contain a site, then all the above kinds of transition still appear. Other transitions also arise, but may be superfluous; this is under present research – in which, following Sewell [15], we shall look in greater detail at how best to handle parametric rules in general. For now, at least in a simple non-parametric example, our derived transitions turn out to be close to what we would expect.

7 Conclusion and further work

There is much work to be done.

(1) Using the results so far obtained, existing calculi (π-calculus, ambients, etc.) must be examined to see how our derived transition systems and equivalences match up to those already known. This will entail dealing sensibly with parametric reaction rules. We expect significant differences to appear. For example the π-calculus is not a graphical model; we expect that the richer contexts the bigraph model will affect the nature of congruential equivalences.

(2) At present we only know about the existence of RPOs in $\mathbf{Big}_{\bar{\mathsf{a}}}$, which has no aliases. I conjecture that RPOs exist under manageable conditions, in the general case; but I found the proofs – though not simple! – to be simpler without aliases, and this restriction also makes examples simpler. The RPO requirement appears natural; therefore, besides applications, there is intrinsic interest in discovering the conditions under which it can be met. Also, in the course of further work it is likely that simpler proofs will be found.

(3) The definition of bigraphs represents a deliberate choice to keep the structures as simple as possible, consistent with the challenge to find a model for mobile interaction that is both rigorous and useful; there are no restrictions upon wiring in relation to locality, arcs are undirected and may branch without constraint, ports are untyped, and so on. We expect to find submodels (subprecategories) or enriched models which embody such constraints or extra structure, but in a way that preserves RPO theory.

(4) While this RPO-based theory appears to be new, there is much work on graph-rewriting which may be relevant to it, especially in relation to graphs with extra structure. Links with that research field need to be explored.

Acknowledgements James Leifer has contributed much of the basis for this work in his PhD Dissertation. Philippa Gardner and Peter Sewell have had a strong influence on different parts of the work. I thank them, as well as Ole Jensen, Alistair Turnbull and Lucian Wischik, for valuable discussions.

References

[1] Berry, G. and Boudol, G. (1992), The chemical abstract machine. Journal of Theoretical Computer Science, Vol 96, pp217–248.

[2] Cardelli, L. and Gordon, A.D. (2000), Mobile ambients. Foundations of System Specification and Computational Structures, LNCS 1378, pp140–155.

[3] Cattani, G.L., Leifer, J.J. and Milner, R. (2000), Contexts and Embeddings for closed shallow action graphs. University of Cambridge Computer Laboratory, Technical Report 496. [Submitted for publication.]
Available at `http://www.cam.cl.ac.uk/users/jjl21` .

[4] Gardner, P.A. (2000), From process calculi to process frameworks. Proc. CONCUR 2000, 11th International Conference on Concurrency theory, pp69–88.

[5] Gardner, P.A. and Wischik, L. (2000), Explicit fusions. Proc. MFCS 2000. LNCS 1893, pp373–383.

[6] Hasegawa, M. (1999) Models of sharing graphs. Distinguished Dissertation Series, Springer-Verlag.

[7] Lafont, Y. (1990), Interaction nets. Proc. 17th ACM Symposium on Principles of Programming Languages (POPL 90), pp95–108.

[8] Leifer, J.J. (2001), Operational congruences for reactive systems. PhD Dissertation, University of Cambridge Computer Laboratory.

[9] Leifer, J.J. and Milner, R. (2000), Deriving bisimulation congruences for reactive systems. Proc. CONCUR 2000, 11th International Conference on Concurrency theory, pp243–258. Available at `http://www.cam.cl.ac.uk/users/jjl21` .

[10] Milner, R. (1996), Calculi for interaction. Acta Informatica 33, pp707–737.

[11] Milner, R. (2001), Bigraphs. Forthcoming Technical Report, University of Cambridge Computer Laboratory.

[12] Milner, R., Parrow, J. and Walker D. (1992), A calculus of mobile processes, Parts I and II. Journal of Information and Computation, Vol 100, pp1–40 and pp41–77.

[13] Parrow, J. and Victor, B. (1998), The fusion calculus: expressiveness and symmetry in mobile processes. Proc. LICS'98, IEEE Computer Society Press.

[14] Petri, C.A. (1962), Fundamentals of a theory of asynchronous information flow. Proc. IFIP Congress '62, North Holland, pp386–390.

[15] Sewell, P. (1998), From rewrite rules to bisimulation congruences. Proc CONCUR'98, LNCS 1466, pp269–284. Full version to appear in Theoretical Computer Science, Vol 272/1–2.

[16] Wojciechowski, P.T. and Sewell, P. (1999), Nomadic Pict: Language and infrastructure design for mobile agents. Proc. ASA/MA '99, Palm Springs, California.

Control of Networks of Unmanned Vehicles

Shankar Sastry

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley CA 94720

At Berkeley we have been interested in design schemes for network of complex networks of semi-autonomous agents. These networks are characterized by interaction between discrete decision making and continuous control. The control of such systems is often frequently organized in hierarchical fashion to obtain a logarithmic decrease in complexity associated with the design, We have used as examples three classes of systems to motivate the design approach:

1. Intelligent Vehicle Highway Systems (IVHS)
2. Air Traffic Management Systems (ATMS)
3. Unmanned Aerial Vehicles

Over the last five years or so, a group of us have developed a set of design approaches which are aimed at designing control schemes which are live, deadlock free, and "safe". Our design methodology is to be considered an alternative to the verification based approaches to hybrid control systems design, and is an interesting blend of game theoretic ideas, planning and fault handling in a probabilistic framework, mathematical and temporal logic and planning ideas from robotics. In today's talk, I will focus on design problems involved in coordinating groups of Unmanned Aerial Vehicles (UAVs). Problems to be addressed include:

1. Design of embedded software for real-time control.
2. Vision based landing and navigation.
3. Pursuit Evasion problems for multi-UAV missions.

The last set of issues touches on issues of decentralized map making, computationally tractable solutions of pursuit evasion games with partial information and probabilistic verification. The work on UAVs is joint with (in alphabetical order) Joao Hespanha, Hyoun Jin Kim, John Koo, Maria Prandini, Omid Shakernia, David Shim, and Claire Tomlin.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, p. 36, 2001.
© Springer-Verlag Berlin Heidelberg 2001

Process Algebra and Security (Abstract)

Steve Schneider

Royal Holloway, University of London
Egham, Surrey, TW20 0EX, UK

Abstract. Over the past decade, techniques from concurrency theory have been applied to problem areas in security, sometimes with extremely successful results. This talk discusses the contribution made by concurrency theory to the analysis of security protocols, and to the characterisation of non-interference properties.

Process algebras provide a rich and mature theory for modelling, developing, and understanding concurrent systems, and they provide a wide variety of techniques for analysing and verifying such systems.

Computer security appears to be a natural application area for concurrency theory. One facet of security is secure communication, which makes use of security protocols [NS78] employing cryptographic mechanisms to achieve properties such as authentication and secrecy, between participants on a (generally insecure) network. Such protocols have much in common with the traditional process algebra application area of communication protocols over unreliable media, and experience from that field is transferable.

This talk will argue that process algebra has been extremely successful in the analysis of security protocols for a number of reasons, including:

- the expressiveness of the languages for describing both protocols and the range of insecure environments they are designed for, as well as the security properties they are intended to provide [RSG+00];
- mature semantic models with well-established verification techniques and powerful tool support;
- experience of language design in the construction of new security-oriented process calculi (for example [AG99]).

Another aspect of security concerns the flow of information between different security levels within a multi-user system. Typically, what is required here is *non-interference*, that activity at higher security levels does not have any impact on activity at lower levels. Natural definitions were given in the early 1980's for deterministic systems (see for example [GM82]), requiring that low-level possibilities do not change during high-level activity. However, this seemingly simple property has been notoriously difficult to generalise to the nondeterministic case, with a bewildering variety of definitions which vary according to their model of computation and their notion of equivalence, as well as the motivation of their proponents. The theme common to the definitions is that the behaviour on a

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 37–38, 2001.
© Springer-Verlag Berlin Heidelberg 2001

particular low-level interface of a process should not be affected by activity at higher levels. Characterisations of non-interference have been expressed for example as properties of sets of traces [McL94]. In the process-algebraic world there have been many characterisations, including those in terms of unwinding rules [Rya90], in terms of process equations [FG95] (that the process is the same in different contexts), or in terms of the low-level processes view being deterministic (see [RWW94], generalised in [Low99, For99]).

A number of benefits have been gained by the application of concurrency theory to the formulation of non-interference as for security protocols, including the deployment of well-established semantic models and mature techniques and tools to analyse systems, and to establish general results about non-interference properties. It has also provided insights such as the observation that the variety of definitions of non-interference in the literature reflects the variety of equivalences within concurrency theory [RS01]. However, the requirements of the security community for non-interference properties which are preserved under composition and refinement, and remain practical and not too restrictive, have not yet been fully met, and they still raise challenges for concurrency theory.

References

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[FG95] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1), 1995.

[For99] R. Forster. *Non-interference properties for nondeterministic processes.* D.Phil, Oxford University, 1999.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Research in Security and Privacy.* IEEE Press, 1982.

[Low99] G. Lowe. Defining information flow. Technical Report 1999/3, Leicester University, 1999.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Research in Security and Privacy.* IEEE Press, 1994.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[RS01] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. *Journal of Computer Security*, 9(1/2), 2001.

[RSG+00] P. Y. A. Ryan, S. A. Schneider, M. H. Goldsmith, G. Lowe, and A. W. Roscoe. *Modelling and Analysis of Security Protocols.* Addison-Wesley, 2000.

[RWW94] A. W. Roscoe, J. Woodcock, and L. Wulf. Non-interference through determinism. In *European Symposium on Research in Computer Security*, number 875 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[Rya90] P. Y. A. Ryan. A CSP formulation of non-interference and unwinding. In *3rd IEEE Computer Security Foundations Workshop*, 1990.

Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software

John Hatcliff and Matthew Dwyer

SAnToS Laboratory, Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA.
{hatcliff,dwyer}@cis.ksu.edu

Abstract. The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin [16], dSpin [6], SMV [3], and JPF [2]). Both program slicing and user extensible abstract interpretation components are applied to customize the program model to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

In this tutorial paper, we use a simple concurrent Java program to illustrate the functionality of the main components of Bandera and how to interact the tool set using its graphical user interface.

1 Introduction

Modern computing applications increasingly require concurrent/distributed software systems that are extremely reliable. Unfortunately, current software validation techniques, such as inspections and testing, are failing to provide high levels of assurance of correctness for these systems due to system size and complexity as well as the fundamental difficulties of reasoning about state/event sequences in concurrent behavior.

Model-checking techniques (now widely used for hardware verification) hold promise for establishing crucial behavioral properties of complex software because they can automatically check to see if an abstract finite-state transition system model of the software conforms to a given state/event sequence property. If the model fails to satisfy the property, the model-checker gives a counterexample — a path through the model's transitions that violates the property. This can be used to locate and correct the corresponding software defect.

Although it holds great promise, we believe that there are four problems that are currently preventing model-checking technology from being successfully applied to software.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 39–58, 2001.
© Springer-Verlag Berlin Heidelberg 2001

The state explosion problem: the exponential increase in the size of a finite-state model as the number of system components grows. A variety of methods exist for curbing the state explosion when analyzing certain types of systems, and these methods have proven sufficient to make analysis of many hardware designs tractable. Unfortunately, software systems tend to have more complex state than hardware components and thus must be more aggressively abstracted to produce tractable models.

The model construction problem: bridging the semantic gap between the artifacts produced by software developers and those accepted by current verification tools. Most development is done with general-purpose programming languages (e.g., C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g., process algebras, state machines). In order to use a verification tool on a real program, a developer must extract an abstract mathematical model of the program's salient behavior and specify this model in the input language of the verification tool. This process is both error-prone and time-consuming.

The requirement specification problem: the difficulty of expressing software requirements in the temporal specification languages of existing model-checking tools. Although model-checker property specification languages are built on theoretically elegant temporal logics, practitioners and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug.

Moreover, model-checker specification languages are designed to state properties of mathematical models rather than software source code. Most software specifications include references to program features such as control-points (e.g., method entry/exit), local and instance variables, array access, nested object dereferences. However, current tools provide little or no support for the intricate mappings that are often required to bridge the gap between source code features and their corresponding model realizations. This means that the user is often forced to state the specifications in terms of the *model's representation of program features* such as coded *e.g.*, in Spin's Promela input language, instead of in terms of the source code itself. Thus, the user must understand these typically highly optimized representations to accurately render the specifications. This is somewhat analogous to asking a programmer to state assertions in terms of the compiler's intermediate representation. Moreover, the representations may change depending on which optimizations were used when generating the model. Even greater challenges arise when modeling the *dynamism* found in typical object-oriented software: components corresponding to dynamically created objects/threads are dynamically added to the state-space during execution. These components are *anonymous* in the sense that they are often not bound directly to variables appearing in the source program. The lack of fixed source-level component names makes it difficult to write specifications describing dynamic component properties: such properties have to be expressed in terms of the model's representation of the heap.

The output interpretation problem: When a property fails when checking large models (and software systems typically produce very large models), the counterexample traces produced by the checker can be hundreds or even thousands of steps long.[1] Manually matching up these counterexamples with source code is extremely tedious for several reasons. First, the length is quite long and it may require hours to walk through the trace. Second, the error trace is expressed in terms of the low-level, possibly highly optimized model representations. Thus, one has the reverse of the "representation gap" issue mentioned in the property specification problem: the analyst must understand the model's representation of complex program features in order to accurately project the model error trace back to the source level. Typically, one "step" in the source program may correspond to as many as ten steps in the low-level model representation.

1.1 Goals and Context of the Bandera Project

Bandera provides multiple forms of tool support to address the problems above. To address the model construction problem, Bandera automatically compiles Java programs to existing model-checking engine back-ends (e.g., Spin, SMV, dSpin, and JPF) which can be inserted into the tool set as pluggable components. To address the requirement specification problem, Bandera provides a temporal specification language that allows user to express properties at the source code level using temporal specification patterns. These specifications are then compiled automatically to the input language of the selected back-end model-checker. To address the state-explosion problem, Bandera provides program slicing and abstract interpretation components, as well as a number of static analyses similar to what one would find in an optimizing compiler — these are used to customize the generated models with respect to the specification to be checked. To address the output interpretation problem, Bandera automatically maps model-checker counterexample traces back to the source-level. A GUI provides debugger-like facilities – it allows the user to navigate a trace both forwards and backwards and to display the contents of the program state (variable and heap values, threads waiting or blocked on locks, etc).

Bandera is the second generation of tools that we have built for model-checking software properties (a previous effort led by Corbett, Dwyer, and Avrunin [7] provided a framework for checking concurrent Ada programs). Work on Bandera began in the fall of 1998, and a number of people have participated in its development. Jay Corbett (faculty, University of Hawai'i) designed and coded the back-end infrastructure which compiles an intermediate representation of Java to several existing model-checkers. In addition to the authors, a good-sized group of research associates and graduate students from Kansas State University including Radu Iosif, Roby Joehanes, Shawn Laubach, Corina Pasareanu, Venkatesh Ranganath, Robby, Oksana Tkachuk, and Hongjun Zheng have contributed to design and implementation of the system.

[1] For example, in a software model-checking experiment performed by researchers at NASA Ames and Honeywell, checking properties of the DEOS real-time operating system scheduler produced SPIN counterexamples over 2700 steps long [19].

Even though a good deal of effort have been devoted to developing Bandera, we believe that each of the previously mentioned barriers to applying model checking to software is significant and it is unclear at present exactly which technologies and forms of tool support will ultimately be *best* suited for overcoming them. Thus, the primary aim of the Bandera project is not to provide "silver bullet" solutions to these problems, but, instead, to provide several different forms of tool support along with an open infrastructure that allows for easy experimentation with new techniques.

The project website `http://www.cis.ksu.edu/santos/bandera` includes a number of papers related to the project, a collection of slides for technical talks and tutorials, and downloadable distribution of the tools. The distribution includes a much larger tutorial and a repository of simple examples.

Outside of our own research group, the primary users of Bandera have been researchers at NASA Ames's Automated Software Engineering and Honeywell Technology Center who have used Bandera in conjunction with NASA Ames's JPF model-checker to check properties of avionics software. It should be noted that these users were already quite familiar with model-checking concepts. Even though Bandera provides many forms of automated support, effective use of it at the present time requires a basic knowledge of temporal logic, explicit state model-checking, and abstract interpretation. Introduction of Bandera in our graduate courses is typically preceded by 2-3 weeks of high-level background material on model-checking and abstraction.

This work was supported in part by the US National Science Foundation under grants CCR-9703094, CCR-9701418, CCR-9708184, CCR-9896354 and CCR-9901605, by the US National Aeronautics and Space Agency (NASA) under grant NAG-02-1209, by Sun Microsystems under grant EDUD-7824-00130-US, by US Department of Defense Advanced Research Projects Agency (DARPA/ITO)'s PCES program through AFRL Contract F33615-00-C-3044, by Honeywell Technology Center and NASA Langley Research Center under Formal Verification of Integrated Modular Avionics Software Cooperative Agreement, NCC-1-399, and by the US Army Research Office under agreement DAAD190110564.

2 Tool Architecture and Use

Figure 1 presents the internal architecture of Bandera, and below we summarize the functionality of the components.

Java infrastructure and intermediate representation: Bandera is built on top of the *Soot* Java compiler framework developed by Laurie Hendren's Sable group at the University of McGill [21]. In the Soot framework, Java programs are translated to an intermediate language called *Jimple*. Jimple was originally developed to be the target language of a Java decompiler (*i.e.*, the Soot tools provide a component for decompiling Java class files (byte code) to Jimple). Thus, Jimple is essentially a language of control-flow graphs where (a) statements appear in three-address-code form (the explicit stack manipulation inherent in JVM instructions has been removed by introducing temporary variables),

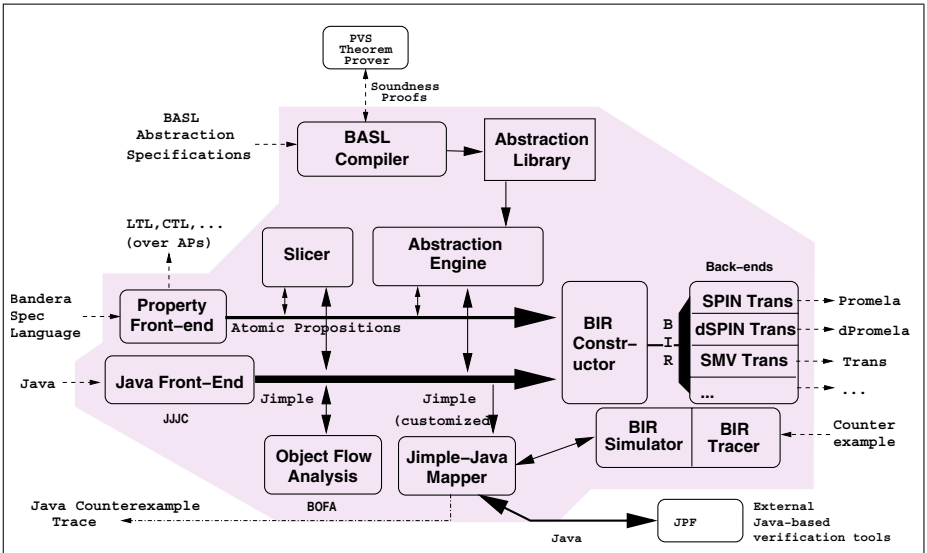


Fig. 1. Internal architecture of the Bandera Tool Set

and (b) various Java constructs such as method invocations and synchronized statements are represented in terms of their virtual machine counterparts (such as `invokevirtual`, and `monitorenter`, `monitorexit`).

Java front-end: Based on an initial prototype from the Soot group, the Bandera group has developed it's own *Java front-end* called JJJC (Java-to-Jimple-to-Java Compiler) that translates from Java to Jimple. JJJC also maintains data structures that are used by a *Jimple-Java Mapper* to move back and forth between Java and Jimple. For instance, JJJC can also decompile Jimple that has been transformed by Bandera's slicing and abstraction components back to Java. Thus, the Bandera slicer and abstraction tools can be viewed as source-to-source transformations. This is useful if one desires to use other verification tools that work at the Java source level in conjunction with Bandera. The Jimple-Java Mapper is also invoked during the process of mapping a model-checker counterexample back to the Java source level.

Property specification: Source code properties to be checked are written in the Bandera Specification Language (BSL). BSL is based on a collection of field-tested temporal specification patterns [11] that allow users to write specifications in a stylized English format. These patterns essentially are parameterized macros that can be instantiated to one or more temporal logics such as LTL or CTL. Thus, this pattern system addresses the specification problem mentioned in the previous section by providing the user with temporal structures commonly used in specifications.

BSL specifications are parameterized by *observables* (predicates on program state) that are defined in Java source code using Javadoc comment notation. From a foundational standpoint, BSL specifications are instantiated to assertions and temporal logic formulas, and user-declared observables are the primitive

propositions that can appear in those formulas. Examples include constraints on program variables and heap objects, and propositions that hold true when control is at a particular control point. The *property front-end* of Figure 1 calls the Java front-end to extract all the observables declared in the given source program, it type checks the declared observables, and it instantiates the BSL specification to a particular temporal logic, and it translates the observables used in the input specification to the lower-level model representation. This last step addresses the representation gap issue of the specification problem by automatically translating properties described in terms of source-level features to the low-level optimized model representation.

Approach to model construction: Bandera's approach to model construction is to generate one model for each property to be checked. This approach is based on the observation that, given a specific property ϕ, many parts of the software may not influence ϕ at all. This allows Bandera to employ optimizations and abstractions that remove program components irrelevant to ϕ and thus generate a highly compacted model. Note that this customization approach generally is infeasible when one is generating models from programs by hand. One might naively complain that generating one model per property incurs significant overhead. However, it is often the case that checking a particular property ϕ without customizing the model is infeasible due to the exponential cost of model-checking. In addition, Bandera's philosophy is to design the customization so that cost of customization is always dominated by model-checking (i.e., in practice, the time to customize should be shorter than the time to model-check).

Slicing: Bandera uses both *program slicing* and *data abstraction* (abstract interpretation) to customize models. The Bandera program slicer takes as input all the observables mentioned in the input property ϕ. These observables may reference particular program variables and control points. The semantics of these program features *must* be preserved for correctly checking ϕ, but all other program components that don't influence the semantics of the observable features can be eliminated in the generated model. The program slicer builds a *program dependence graph* representing several different forms of dependence, and it will generate an executable residual program (the program slice) where components that do not influence the execution of the observables in ϕ have been removed. The sliced program is created at the Jimple level, but it can also be decompiled to Java source using JJJC.

Abstract interpretation: The Bandera abstraction components provide automated support for reducing model size *via* data abstraction. This is useful when a specification ϕ to be checked does not depend on the program's concrete values but instead depends only on *properties* of those values. For example, an application might store a set of items in a vector, but if the property being verified depends only on whether a particular item is in the vector, we could abstract the large number of vector states onto a small set $\{ItemInVector, ItemNotInVector\}$. The user guides the abstraction process by binding variables to entries from an *abstraction library*. The library entries are indexed by concrete type, and each entry implements an abstract version of its corresponding concrete type. Each

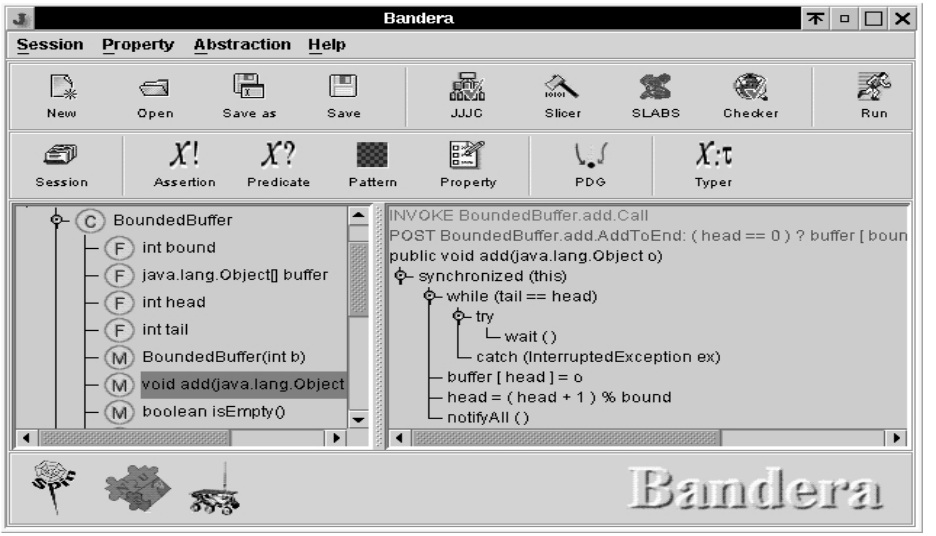


Fig. 2. Main window of the Bandera User Interface (BUI)

abstraction in the library is defined using the Bandera Abstraction Specification Language (BASL).

Back end model generation: The Bandera back end is like a code generator, taking the sliced and abstracted program and producing verifier-specific representations for targeted verifiers. The back end components communicate through BIR, the Bandera Intermediate Representation, an intermediary between compiler-based representations and verifier-based representations. As shown in Figure 1, the back end has one fixed component called BIRC (Bandera Intermediate Representation Constructor) that accepts a restricted form of Jimple and produces BIR. For each supported verifier, there is also a translator component that accepts the program represented in BIR and generates input for that verifier. Currently, translators for SPIN, dSPIN, and SMV have been incorporated. In addition to these BIR-based back ends, the JPF model-checker from NASA Ames [2] has also been incorporated. JPF works directly on Java byte code, so translation to BIR is by-passed when generating JPF input.

3 Overview of the Bandera User Interface

Figure 2 displays the main window of the BUI with some example code loaded. The main window contains two panels: the left panel is the *project panel* and the right panel is the *code panel*. The Project panel contains a tree that organizes the packages, classes, fields and methods of the Java software being analyzed. Selecting a node in the project panel brings up a detailed view of the selected object in the code panel. For instance, in the example display in Figure 2, selecting the `add` method from the `BoundedBuffer` class displays the code structure for the `add` method in the code panel.

Below we give a brief description of each tool-bar button and menu in the main Bandera window.

Sessions: Runs of Bandera are configured using *sessions*. A session is a record holding information about the file name(s) of the source code to be checked during the run, the property to be checked, the tool components that are to be enabled during the run, options and settings for the selected components, the particular back-end model-checker to be used, and other miscellaneous information such as location of working directories into which temporary output should be dumped.

Multiple session records are held in a *session file*. When performing a new run of Bandera, the session record can be saved in a session file and loaded at a later time. This allows the user to avoid restating all option information, etc. Session records in a session file can also be processed in batch mode using a command line flag. This is useful for performing regression tests on software under development. For example, you might consider creating a session file holding all the checks that you usually run on a piece of software, then using the batch mode facility to run all of the model-checks specified in the session file overnight.

Component buttons: The four buttons at the right side of the upper tool-bar (labeled JJJC, Slicer, SLABS, Checker) are used to enable/disable the four main components of Bandera. When a component is enabled, the corresponding button icon is presented in color with a check-mark. A non-checked black-and-white icon indicates that a component has been disabled. Clicking on the Run button causes the enabled components to be executed in the order in which they are presented in the tool-bar. For example, in Figure 2, JJJC is enabled, but the remaining components are disable. Therefore, pressing the run button only parses the program and loads it into the BUI.

Session manager: Starting on the lower toolbar, the first button opens the Session Manager window. The window is used for creating, modifying, and deleting of session records within the currently selected session file.

Property specification buttons: The next four buttons on the lower tool bar are associated with property specification. The Assertion button invokes the Assertion Browser which allows you to browse the assertions that you have declared in the code specified by the current session. The Predicate button invokes the Predicate Browser which allows you to browse the predicates (observables) that you have declared in the code specified by the current session. The Pattern button invokes the Pattern Manager. Recall that Bandera's temporal specification language is based on a collection of *temporal specification patterns*. The Pattern Manager allows the user to browse the patterns, and to add, modify or remove patterns. Typically, only expert users will change the collection of patterns. The Property button invokes the Property Manager which allows the user to specify properties to be checked for the code specified by the current session.

The PDG Browser: The next button in the lower toolbar invokes the PDG Browser. The PDG Browser allows the user to navigate the *program dependence*

```
/**
 * @observable
 *  EXP Full(this): (head == tail);
 *  EXP TailRange(this):
 *  (tail >= 0 && tail < bound);
 *  EXP HeadRange(this):
 *  (head >= 0 && head < bound);
 */
class BoundedBuffer {
 Object [] buffer;
 int bound, head, tail;
 /**
  * @assert
  *  PRE PositiveBound: (b > 0);
  */
 public BoundedBuffer(int b) {
  bound = b;
  buffer = new Object[bound];
  head = 0;
  tail = bound - 1;
 }
 /**
  * @observable
  *  RETURN ReturnTrue(this):
  *           ($ret == true);
  */
 public synchronized
        boolean isEmpty() {
  return
   head == (tail + 1) % bound;
 }
```

```
/**
 * @observable
 *  INVOKE Call(this);
 * @assert
 *  POST AddToEnd:
 *         (head == 0) ?
 *         buffer[bound-1] == o :
 *         buffer[head-1] == o;
 */
public synchronized
       void add(Object o) {
 while ( tail == head )
  try { wait(); }
  catch (InterruptedException ex) {}
 buffer[head] = o;
 head = (head+1) % bound;
 notifyAll();
}
/**
 * @observable
 *  RETURN Return(this);
 */
public synchronized Object take() {
 while ( isEmpty() )
  try { wait(); }
  catch (InterruptedException ex) {}
 tail = (tail+1) % bound;
 notifyAll();
 return buffer[tail];
}
}
```

Fig. 3. Bounded Buffer Implementation with Predicate Definitions

graph – the main data structure produced by the slicer. Several facilities are provided in the PDG Browser that aid in the selection of abstractions.

Abstract type inference: The next button in the lower toolbar invokes the Abstraction Manager which is used to bind abstract types to source code variables. The Abstraction Manager communicates this information about abstraction selection to the abstraction engine, and this information is used to drive the program transformation associated with abstraction.

4 Property Specification Using BSL

4.1 An Example

Figure 3 gives the implementation of a simple bounded buffer implementation in Java that is amenable to simultaneous use by multiple threads. This code illustrates several of the challenges encountered in specifying the behavior of

```
// Enable PositiveBound or AddtoEnd pre-condition assertions
BoundAssertion: enable assertions { PositiveBound };
AddToEndAssertion: enable assertions { AddToEnd };

// Indices always stay in range
IndexRangeInvariant:
  forall[b:BoundedBuffer].
    {HeadRange(b) && TailRange(b)} is universal globally;

// Full buffers eventually become non-full
FullToNonFull:
  forall[b:BoundedBuffer]. {!Full(b)} responds to {Full(b)} globally;

// Empty buffers must be added to before being taken from
NoTakeWhileEmpty:
  forall[b:BoundedBuffer].
    {BoundedBuffer.take.Return(b)} is absent
    after {BoundedBuffer.isEmpty.ReturnTrue(b)}
    until {BoundedBuffer.add.Call(b)};
```

Fig. 4. Bounded Buffer Properties rendered in BSL

Java programs. Each instance of the `BoundedBuffer` class maintains an array of objects and two indices into that array representing the `head` and `tail` of the active segment of the array. Calls to `add` objects to the buffer are guarded by a check for a full buffer using the Java condition-`wait` loop idiom. Calls to `take` objects from the buffer are guarded similarly by a check for an empty buffer.

We will use BSL to specify the following requirements of the buffer code.

1. Buffers are constructed with positive bounds.
2. Elements are always added in correct position.
3. Buffer indices always stay in range.
4. Full buffers eventually become non-full.
5. Empty buffers must be added to before being taken from.

Comments in the code of Figure 3 contain various BSL predicate and assertion declarations for these properties, and Figure 4 presents the actual assertion and temporal specifications. We will discuss these in the following sections.

4.2 Structure of BSL

The Bandera Specification Language (BSL) is a source-level, model-checker independent language for expressing temporal properties of Java program actions and data. BSL addresses the property specification problem outlined in the introduction and provides support for overcoming the hurdles one faces when specifying properties of dynamically evolving software. For example, consider the bounded buffer Requirement 4 from above stating that *no buffer stays full forever*. There are several challenges in rendering this specification in a form that can be model-checked including (a) defining the meaning of *full* in the implementation, (b)

quantifying over time to insure that full buffers eventually become non-full, and (c) quantifying over all dynamically created bounded buffers instances in the program. BSL separates these issues and treats them with special purpose sublanguages: a *predicate sublanguage* for defining basic observations (*i.e.*, propositions) about program's state (addressing item (a)), a pattern-based *temporal property sublanguage* for expressing temporal relationships between observables (addressing item (b)), and an *object-instance quantification* facility (addressing item (c)). In addition, an *assertion sublanguage* is provided.

Assertions: An *assertion sublanguage* provides a convenient way for a developer to specify a constraint on a program's data space that should hold when execution reaches a particular control location. In C and C++ programming, assertions are typically embedded directly in source code using an `assert` macro, where the location of the assertion is given by the position of the macro invocation in the source program. Due to Java's support for extracting HTML documentation from Java source code comments via Javadoc technologies, several popular Java assertion facilities, such as iContract, support definition of assertions in Java method header comments. BSL also adopts this approach.

For example, Figure 3 shows the declaration of the BSL assertion `PRE PositiveBound: (b > 0)`. In this assertion, the data constraint is `(b > 0)` and the control location is specified by the occurrence of the tag `@assert PRE` in the method header documentation for `BoundedBuffer` constructor: the constraint must hold whenever control is at the first executable statement in the constructor. Other assertion forms include *post-conditions* (see the `AddToEnd` postcondition for `add` method in Figure 3), and user-specified `LOCATION` assertions. `LOCATION[<label>] <assertion-name>: <exp>` is satisfied if `<exp>` is true when control is at the Java statement labeled by `<label>` in the corresponding method).[2]

Assertions can be selectively enabled/disabled so that one can easily identify only a subset of assertions for checking. Bandera exploits this capability by optimizing the generated models (using slicing and abstraction) specifically for the selected assertions. For example, the `BoundAssertion` property of Figure 4 enables only the `PositiveBound` assertion but not the `AddToEnd` assertion. When checking, `PositiveBound` the actual array implementing the buffer will be sliced away because it is not need for checking `PositiveBound`. Note that without selective checking, the buffer array would be included since it is required for `AddToEnd`.

Predicates: A *predicate sublanguage* provides support for specifying *observable properties* of common Java control points (*e.g.*, method invocation and return) and Java data (including dynamically created threads and objects). These predicates become the basic *propositions* in temporal specifications. For example, Figure 3 shows a declaration of a location insensitive *expression predicate* `EXP Full(this): (head == tail)` in the class `BoundedBuffer` header documentation. Expression predicates are often used to define class invariants or to

[2] Even though Java does not include `goto`'s, it includes labels to indicate the targets of `break` and `continue` statements.

indicate distinguished states (*e.g.*, a full buffer) in class or instance data. Since expression predicates do not refer to particular control points in methods, they can only be defined in class header documentation. Other predicate forms include *invocation predicates* (*e.g.*, the `Call` predicate for method `add` in Figure 3), *return predicates* (*e.g.*, the `ReturnTrue` predicate for method `isEmpty`), and *location predicates* (similar to location assertions). As an example of the semantics of these predicates, `RETURN ReturnTrue(this): ($ret == true);` of method `isEmpty` is true exactly when `isEmpty` is invoked on the instance bound to the predicate parameter `this` and control is at the `return` statement of `isEmpty` and the return value is `true`.

Temporal properties: The temporal specification language is based not on a particular temporal logic, but on a collection of field-tested temporal specification patterns developed in our earlier work [11]. This pattern language is extensible and allows for libraries of domain-specific patterns to be created. There are five basic patterns:

- *universal* properties require the argument to be true throughout the execution
- *absence* properties require that the argument is never true in the execution
- *existence* properties require that the argument is true at some point in the execution
- *response* properties require that the occurrence of a designated state/event is followed by another designated state/event in the execution
- *precedence* properties require that a designated state/event always occurs before the first occurrence of another designated state/event

In addition several *chain* patterns allow for the construction of sequences of dependent response and precedence relationships to be specified. A web-site [10] presents the current set of eight patterns and their variations as well as translations into five different common temporal specification formalisms, including LTL and CTL.

Pattern scopes define variations of the basic patterns in which checking of the pattern is disabled during specified regions of execution. There are five basic scopes; a pattern can hold *globally* throughout the system's execution, *after* the first occurrence of a state/event, *before* the first occurrence of a state/event, *between* a pair of designated states/events, during the interval, or *after* one state/event *until* the next occurrence of another state/event or throughout the rest of the execution if there is no subsequent occurrence of that state/event. For example, the `NoTakeWhileEmpty` property of Figure 4 uses the absence pattern with after/until scope.

Object Quantification: Interacting with both the predicate and pattern support is a powerful *quantification facility* that allows temporal specifications to be quantified over all objects/threads from particular classes. Quantification provides a mechanism for *naming* potentially anonymous data, and we have found this type of support to be crucial for expressive reasoning about dynamically created objects.

Bandera implements object quantification through a mechanism that avoids having to extend the functionality of any of its back-end model-checkers. This is achieved by (a) augmenting the program/model to bind quantified variables non-deterministically to allocated instances of the classes named in the quantification and by (b) augmenting the property ϕ to be checked by embedding it in another temporal formula that assures ϕ will be checked only when quantifier variables have actually been bound to allocated objects.

Although the specification file of Figure 4 can be created directly via a text editor, the BUI's Property Manager provides a nice system of pull-down menus that collect the predicates declared in the code to aid the user in constructing specifications. See [5] for a detailed presentation of the syntax and semantics of BSL, as well as several more examples.

When analyzing a unit of code U like the bounded buffer class, one begins by creating an appropriate model of U's *environment* to close the system. This model of the environment is analogous to a test harness. Typically, it will non-deterministically generate calls to the interface of U to simulate demonic behavior of the actual environment.

To model the environment for the `BoundedBuffer` class, we built a simple closure (not shown here) consisting of four threads: a main thread that instantiates two `BoundedBuffers`, loads one of the buffers until it is full, then passes the pair of buffers to threads that read from one buffer and write to the other such that a *ring* topology is established. An additional thread repeatedly polls the state of the two `BoundedBuffers` to determine if they are empty. Under this environment, each buffer will be forced through all its internal states limited by its `bound`, and Bandera determines that all of the properties in Figure 4 hold.

The current release of Bandera provides no tool support for generating environments. However, we have constructed initial prototypes based on the second author's previous work on filter-based refinement and assume-guarantee model-checking [9], and we plan to incorporate fully developed versions of these in a future Bandera release.

5 Slicing

Given a program P and some statements of interest $C = \{s_1, \ldots, s_k\}$ from P called the *slicing criterion*, a program slicer will compute a reduced version of P by removing statements of P that do not affect the computation at the criterion statements C. When checking a program P against a BSL specification ϕ, Bandera uses slicing to remove the statements of P that do not affect the satisfaction of ϕ. Thus, the specification ϕ holds for P if and only if ϕ holds for the reduced version of P (*i.e.*, the reduction of P is sound and complete with respect to ϕ) [14].

In recent work [14], we showed that the slicing transformation can driven by generating a slicing criterion C_ϕ based only on the primitive propositions (*i.e.*, the predicates) in ϕ. We noted earlier that BSL's predicates may involve two types of observations: observations about the values of values, and observations about the current control location. The criterion C_ϕ consists of those program

statements whose control locations are mentioned in ϕ predicates, as well as all assignment statements to variables mentioned in ϕ.

As an example, consider slicing with respect to the `FullToNonFull` specification of Figure 4. Since `Full` is the only predicate in this specification and it is not location sensitive, Bandera's automatically generated slicing criterion consists of all the statements that assign to the variables mentioned in `Full` (*i.e.*, `head`, and `tail`). Using this criterion, Bandera will slice away statements that are guaranteed not to influence the criterion statements. In this case, the `buffer` array and any values flowing into or out of it can sliced away — in essence, the `FullToNonFull` only concerns the control of available positions in the buffer and not its actual contents.

Building a slicer for Java requires a significant amount of effort. Fortunately, except for issues surrounding Java's concurrency primitives we were able to carry out most of the development using previously developed slicing techniques based on program dependence graphs. In recent work, we gave a formal presentation of slicing that includes additional notions of dependence that arise in Java's concurrency model [13]. These includes dependencies due to possibly infinite delays in waiting for locks or notification *via* Java's `notify/notifyall`, data dependencies due to access/definition of shared variables, and dependencies between program statements and the monitor delimiters that enclose them.

The effectiveness of slicing for reducing program models varies depending on the structure of the program. In some systems that we have considered, slicing removes entire threads and dramatically reduces the state space. In other cases, where program components are tightly coupled or where large sections of the program are relevant to the specification, the slicing reduction is moderate to negligible. However, since slicing is cheap compared to the overall cost of model-checking and since it is totally automatic, we almost always use Bandera with the slicing option enabled. We've encountered numerous examples where slicing turned an infeasible checking program into a feasible one.

For more details, we refer the reader to formalizations of the Bandera slicer's approach to property-driven slicing [14], and the notions of program dependence required for slicing the concurrent features of Java [13].

6 Abstraction

The user guides the abstraction process by binding variables to entries from an *abstraction library*. The library entries are indexed by concrete type, and each entry implements an abstract version of its corresponding concrete type. Each abstraction in the library is defined using the Bandera Abstraction Specification Language (BASL). A BASL specification for a base type abstraction consists of a declaration of a finite set of abstract tokens, an abstraction function that maps each concrete Java value to an abstract token, and an abstract operation for each operation of the concrete type. A rule-based format that incorporates pattern matching simplifies the definition of abstract operations. For base type abstractions, the BASL compiler allows the user to supply an abbreviated BASL specification containing only the abstract token set and abstraction function. It

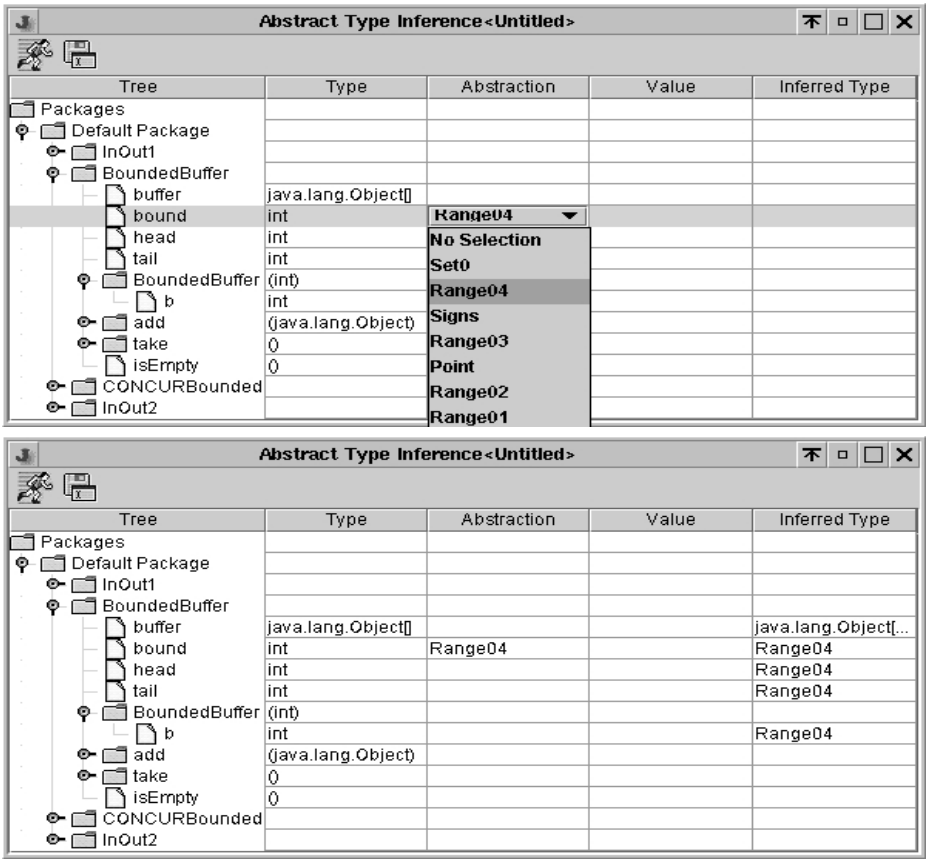


Fig. 5. Abstraction selection and abstract type inference

then uses the PVS theorem prover to automatically synthesize the abstract versions of operations such as `+`, `-`, etc. This makes it extremely easy to define new abstractions on base types. Given a (possibly synthesized) BASL specification, the BASL compiler generates a Java class containing methods that implement the defined abstraction function as well as abstract versions of the relevant concrete operators. The Java class is held in the library, and is linked with rest of source code during model-generation if the user has selected that particular abstraction.

Since abstractions are incorporated on a per variable basis, two different variables of the same concrete type can have different abstract types. For example, if I_1 and I_2 are both `int` abstractions, then variable `int x` may be bound to I_1 and variable `int y` may be bound to I_2. After the user has chosen abstractions for a few key variables that are relevant to the property being checked, a type inference phase propagates this information throughout the program and infers abstraction types for the remaining variables and for each expression in the program. Type inference also informs the user of an abstraction type conflict.

Although abstraction is not needed to check our example buffer properties, we illustrate the abstraction selection interface by abstracting the `bound` field using a *range* abstraction with the abstract token set `{LT0,0,1,2,3,4,GT4}`. This is reasonable since our environment only creates buffers of size three (although, note that the abstraction does not result in a smaller state-space in this contrived example). Intuitively, this abstraction tracks concrete values from 0 to 4, but "summarizes" negative values and values greater than 4 using the tokens `LT0` and `GT4`, respectively. The top of Figure 5 shows the state of the type inference window when the user is making an abstract type selection for the `bound` variable by selecting the `Range04` abstraction from among those integer abstractions currently held in the abstraction library. The bottom of Figure 5 shows the state of the window *after* abstract type inference has run. In this case, the typing information for `bound` has been propagated to the other integer variables in the program.

Once abstract type inference is run, the abstraction engine will transform the source code into a abstracted version where all concrete operations and tests on the relevant objects (e.g., addition and equality on integers) are replaced with abstract versions that manipulate tokens representing the abstract values. Since information is lost in this transformation, operations and tests that relied on the lost information can no longer be determined completely in the abstract program. For instance, `GT4 == GT4` cannot be determined because `GT4` represents more than one number. Any conditional with a test like this would be transformed into a non-deterministic choice between the true and false branches. As with slicing, the abstracted code is represented at the Jimple level, but JJJC can decompile it back to Java.

For more details on Bandera's abstraction facilities, see [8].

7 Back End

BIR is a guarded command language for describing state transition systems. The main purpose of BIR is to provide a simple interface for writing translators for target verifiers—to use Bandera with a new verifier, one must only write a translator from BIR to the input language of the verifier. The guarded command style of BIR meshes well with the input languages of existing model checkers.

BIR contains some higher-level constructs to facilitate modeling Java, such as threads, Java-like locks (supporting wait/notify), and a bounded form of heap allocation. Rather than choose an implementation of these constructs and remove them from BIR (e.g., model a lock as a boolean variable), we allow the translators to implement these constructs in whatever way is most efficient in the verifier input language. BIR also provides other kinds of information that can aid translators in producing more compact models. For example, a guarded command can be labeled *invisible*, indicating that it can be executed atomically with its successor. The set of local variables that are live at each control location can be specified (dead variables can be set to a fixed value for SPIN or left unconstrained for SMV).

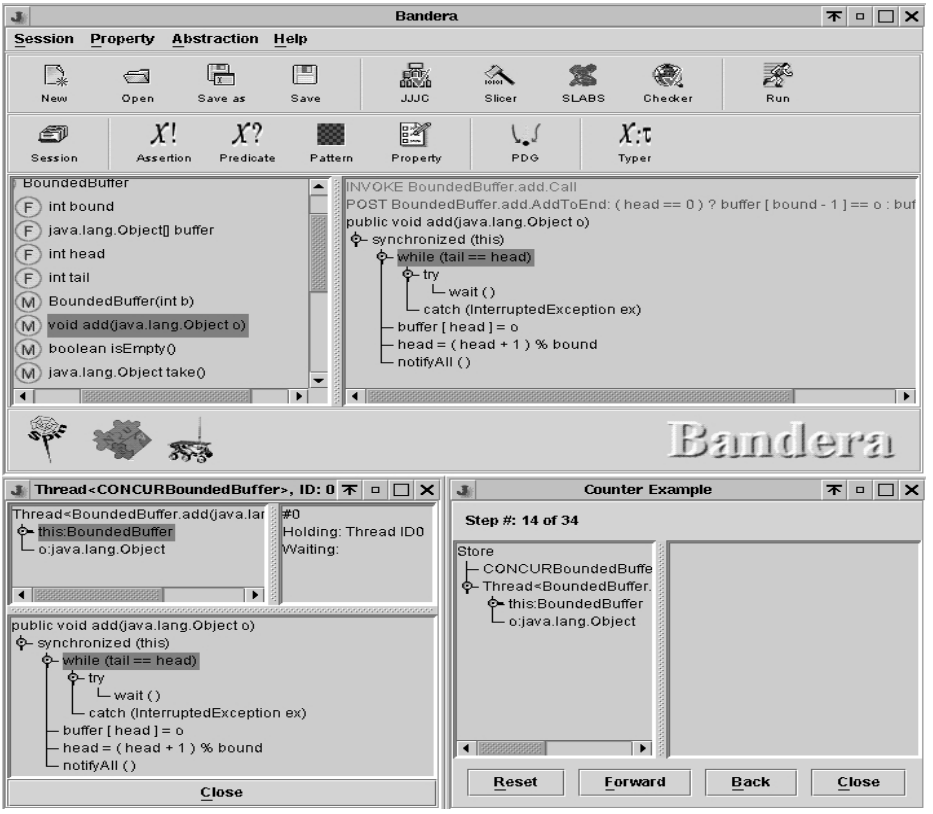


Fig. 6. BUI counterexample display

BIRC translates a subset of Jimple to BIR. Java locals and instance variables are mapped onto BIR state variables and record fields. The Jimple statement graph is traversed to construct a set of guarded commands for each thread. Each guarded command is marked as visible/invisible based on the kind of data accesses (e.g., operations on locals are invisible). BIRC also accepts a set of expressions used to define primitive propositions in the model (e.g., a thread is at a specific statement, a variable has a given value). BIRC embeds these proposition definitions into the BIR and insures that any program statement that changes the value of one of these primitive propositions will cause a visible state change in the model.

8 Counterexample Display

Since all of the specifications of Figure 4 *hold* for the example buffer code, we consider an additional specification that is *violated* by the code. Of course, when a program violates a specification, it could be that there is an error in the code, or it could be that the specification is erroneous because it does not faithfully represent the desired functionality of the system. We illustrate Bandera's counterexample display with a specification of the latter sort.

In Figure 3, we specified the notion of buffer emptiness via the predicate `isEmpty.ReturnTrue`. However, one can also describe an empty buffer state directly by adding the following predicate to the other `EXP` predicates in the `BoundedBuffer` class header.

```
EXP Empty:  head == ((tail+1) % bound);
```

Now, consider the following naive specification which requires that items cannot be taken from the buffer after it becomes empty.

```
NoTakeAfterEmpty: forall[b:BoundedBuffer].
   {take.Return(b)} is absent after {Empty(b)};
```

This specification is violated by a correct implementation because it fails to consider intervening calls to `add`, *e.g.*, the valid sequence of actions *is empty*, *add to buffer*, *take from buffer* violates the specification.

Bandera detects this violation (we use the Spin back-end in this case) and brings up the counterexample display in Figure 6. The window on the lower right is the *counterexample control* window. Buttons allow the user to step forward and backward along the trace and to reset the display to the beginning of the trace. During navigation, the current method and statement being executed by each thread is highlighted (there is a separate display window for each thread). In this case, an environment thread `CONCURBoundedBuffer` is executing the buffer `add` method (the intervening call to `add` which leads to the violation described above). This thread's display window is on the lower left, and a user can display properties of selected variables. In this case, one can see that the lock of the selected variable `this:BoundedBuffer` is held by environment thread (which has an id of 0). Finally, the top of the control panel reports that the counterexample display is positioned at step 14 out of a total of 34 steps. It is interesting to note that counterexample at the Spin level has over 600 steps (there are many steps in the model for each source code step). This clearly motivates the need for this type of tool support.

9 Related Work

There are several other significant software model-checking projects, and technical comparisons between our approach and these others can be found in our technical papers (*e.g.*, [4,5,8,14]). Below, we simply give a concise summary and literature references of notable projects.

The Automated Software Engineering group at NASA Ames has developed a flexible explicit-state model-checker Java Pathfinder (JPF) that works directly on Java byte-code [2]. They have also produced a simple predicate abstraction tool and a distributed version of the model-checking engine. In collaboration with researchers at NASA Ames, JPF has been incorporated as a back-end checker for Bandera.

The Microsoft Research SLAM Project [1] focuses on checking sequential C code using well-engineered predicate abstraction and abstraction refinement tools. Operating system device drivers are emphasized as an application domain.

Gerard Holzmann's Feaver tool extracts Promela programs from annotated C programs for checking with SPIN [15]. This tool has been used in several substantial production telecommunications applications.

Scott Stoller [20] has developed a stateless checker for multi-threaded distributed Java programs. The basic technology used is an extension of Godefroid's Verisoft approach [12].

David Dill's Hardware Verification Group at Stanford has developed a tool that translates Java into SAL – an intermediate language designed to interface with several model-checking and theorem-proving tools developed at Stanford and SRI [18].

Eran Yahav has developed a tool for checking safety properties of Java programs [22] built on top of Lev-Ami and Sagiv's three-valued logic analysis tool (TVLA) [17].

10 Conclusion

Bandera is a rather large and diverse collection of tools, and it is impossible to present anything more than a very high-level overview in a paper like this. However, we hope that the overall goals and direction of the project have been made clear and that the reader receives a reasonable impression of what is involved in using the tool set.

There are a number of limitations to the current version of the tools that we hope to overcome in future releases: each created thread must be named by a distinct reference variable, all methods are inlined, and user-defined exceptions are not treated. These restrictions will be lifted by creating an extended version of BIR. Future releases will have enhanced capabilities for BSL specifications of Java interfaces, environment generation, abstraction of heap configurations, and UML state-chart specifications.

Finally, we hope that researchers outside of our group may contribute components to Bandera, *e.g.*, back-end translators to additional model-checkers.

References

1. T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, LNCS 1885, Springer-Verlag, 2000.
2. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
3. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000. (to appear)
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000.

5. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. Submitted for publication. A shorter version of this paper appeared in the 2000 Spin Workshop.
6. C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, 1999.
7. M. B. Dwyer, J. C. Corbett, and C. S. Păsăreanu. Translating Ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.
8. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported abstraction for finite-state verification. In *Proceedings of the 23nd International Conference on Software Engineering*, pages 177–187, May 2001.
9. M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998.
10. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. A System of Specification Patterns. |http://www.cis.ksu.edu/santos/spec-patterns—, 1998.
11. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
12. P. Godefroid. Model-checking for programming languages using VeriSoft. POPL'97, pages 174–186, January 1997.
13. J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*.
14. J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4):315–254, December 2000.
15. G. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.
16. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
17. T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene-based static analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, 2000.
18. D. Y. W. Park, U. Stern, J. U. Skakkebaek, and D. L. Dill. Java model checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification*, June 2000.
19. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the DEOS scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
20. S. Stoller. Model-checking multi-threaded distributed Java programs. In K. Havelund, editor, *Proceedings of Seventh International SPIN Workshop*, LNCS 1885, pages 224–244. Springer-Verlag, 2000.
21. R .Valle-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON'99*, November 1999.
22. E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. POPL'01, pages 27–40, January 2001.

Performance Evaluation := (Process Algebra + Model Checking) × Markov Chains

Holger Hermanns and Joost-Pieter Katoen

Formal Methods and Tools Group
Faculty of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. Markov chains are widely used in practice to determine system performance and reliability characteristics. The vast majority of applications considers continuous-time Markov chains (CTMCs). This tutorial paper shows how successful model specification and analysis techniques from concurrency theory can be applied to performance evaluation. The specification of CTMCs is supported by a stochastic process algebra, while the quantitative analysis of these models is tackled by means of model checking. Process algebra provides: (i) a high-level specification formalism for describing CTMCs in a precise, modular and constraint-oriented way, and (ii) means for the automated generation and aggregation of CTMCs. Temporal logic model checking provides: (i) a formalism to specify complex measures-of-interest in a lucid, compact and flexible way, (ii) automated means to quantify these measures over CTMCs, and (iii) automated measure-driven aggregation (lumping) of CTMCs. Combining process algebra and model checking constitutes a coherent framework for performance evaluation based on CTMCs.

1 Introduction

What is performance evaluation? Performance evaluation aims at analysing quantitative system aspects that are related to its performance and dependability – what is the frequency of anomalous behaviour?, or, is correct and timely packet delivery guaranteed in at least 92% of all cases? Major performance evaluation approaches are *measurement-based* and *model-based* techniques. In measurement-based techniques, controlled experiments are performed on a concrete (prototypical) realisation of the system, and gathered timing information is analysed to evaluate the measure(s) of interest such as time-to-failure, system throughput, or number of operational components. In model-based performance evaluation, an abstract (and most often approximate) model of the system is constructed that is just detailed enough to evaluate the measure(s) of interest with the required accuracy. Depending on modelling flexibility and computational requirements, either analytical, numerical or simulative techniques are used to evaluate the required measure(s). We focus on model-based performance evaluation and their numerical analysis.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 59–81, 2001.
© Springer-Verlag Berlin Heidelberg 2001

Models and measures. Continuous-time Markov chains (CTMCs) are a widely used performance evaluation model. They can be considered as labelled transition systems, where the transition labels – rates of exponential distributions – indicate the speed of the system evolving from one state to another. Using specification techniques such as queueing networks [19], stochastic Petri nets [1] or stochastic networks [42], CTMCs can be described in a quite comfortable way. Typical performance measures of CTMCs are based on steady-state and transient-state probabilities. Steady-state probabilities refer to the system behaviour on the "long run", i.e., when the system has reached an equilibrium. Transient-state probabilities consider the system at a fixed time instant t. State-of-the-art numerical algorithms allow the computation of both kinds of probabilities with relative ease and comfortable run times, and in a quantifiable precise manner. Several software-tools are available to support the specification and analysis of CTMCs.

Performance evaluation and concurrency theory: A couple? Given the success of CTMCs and their wide industrial applications, it is stunning that these models have received scant attention in concurrency theory for a long time: where probabilistic aspects have come into play, they were mostly of a purely discrete nature. This is even more remarkable, as model specification and analysis techniques – key ingredients of performance evaluation methodology – are first class examples of the success of formal methods for concurrent or reactive systems. Moreover, in modern systems many relevant functionalities are inextricably linked to performance aspects and the difference between functional and performance properties has become blurred. While formal methods for concurrency have mainly been focused on functional features, we believe that these methods have finally reached a state in which performance aspects should play a larger rôle. As a – to our opinion – promising example of the cross-fertilisation of concurrency theory and performance evaluation, this paper surveys a formal framework for the specification and analysis of CTMCs. The proposed methodology is based on appropriate extensions of *process algebra* for the description of CTMCs and *model checking* techniques for their analysis.

Stochastic process algebra. Stochastic process algebras are extensions of process algebras such as ACP, CCS and CSP in which actions can be delayed according to some negative exponential distribution. A mapping from algebraic terms onto CTMCs is obtained by a formal semantics in traditional SOS-style. The process algebraic setting provides a specification formalism for describing CTMCs in a precise, and modular way, resembling the hierarchical nature of most modern systems. In this setting, strong bisimulation coincides with lumpability, a notion central to the aggregation of Markov chains. The computation of this bisimulation equivalence can be performed using small adaptations of existing algorithms for computing strong bisimulation without an increase of their worst-case complexity. This provides means to minimise CTMCs (w.r.t. lumpability) in an efficient and fully automated way – a result that was unknown in performance evaluation. The congruence property of bisimulation allows this minimisation to

be carried out in a compositional fashion, i.e., component-wise, thus avoiding an a priori generation of the entire (and possibly huge) state space. An appropriate choice of the basic algebraic operators supports:

- an *orthogonal* extension of traditional process algebra, yielding a *single framework* for the description of both functional and performance aspects;
- the specification of exponential and *non-exponential* distributions such as the rich class of phase-type distributions;
- the *constraint-oriented* specification of stochastic time constraints, i.e., without modifying existing untimed specifications;
- variants of *weak bisimulation* congruences (and their algorithms) that combine lumpability and abstraction of sequences of internal actions.

Model checking. The process algebraic specification of the performance model can be complemented by a specification of the performance measure(s)-of-interest in a stochastic variant of the branching-time temporal logic CTL. This logic is formally interpreted over CTMCs and allows to express quantifiable correctness criteria such as: in 99% of the cases no deadlock will be reached within t time units. The logic-based method provides ample means for the unambiguous and lucid specification of requirements on steady-state and transient-state probabilities. Besides, it allows for the specification of path-based properties, measures that in performance evaluation are described informally in an ad-hoc manner and mostly require a manual tailoring of the model. To check the validity of formulas, model checking algorithms are adapted. They are enriched with appropriate numerical means, such as simple matrix manipulations and solution techniques for linear systems of equations, to reason about probabilities. Probabilistic timing properties over paths are reduced to computing transient-state probabilities for CTMCs, for which dedicated and efficient methods such as uniformisation can be employed. The use of temporal logic and model checking supports:

- the preservation of the validity of all formulas under *lumping*;
- *automated means* to analyse state-based and path-based measures over CTMCs;
- *automated measure-driven aggregation* of CTMCs;
- *hiding specialised algorithms* from the performance engineer;
- a means to specify both functional and performance properties in a *single framework*.

Organisation of the paper. This paper gives a flavour of the approaches mentioned above. A more detailed treatment of the process algebra part can be found in [28,31,29]; the model checking part is described in full detail in [7,5]. For a broader overview and introduction into formal methods and performance evaluation we refer to [13]. The paper is organised as follows. Sec. 2 presents some introductory material on CTMCs. Sec. 3 surveys stochastic process algebras and indicates the main issues involved, such as synchronisation, abstraction and interleaving. Sec. 4 discusses the temporal logic approach and some of the

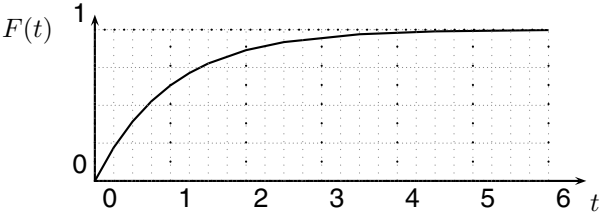
model checking algorithms. Sec. 5 concludes the paper and discusses some future research directions.

2 Continuous-Time Markov Chains

This section introduces exponential distributions, continuous-time Markov chains and their behaviour. This is done in an informal way, we refer to [29, 27] for details.

2.1 Exponential Distributions

A probability distribution (function) is a function that assigns a probability (a real value between 0 and 1) to each element of some given set. This set is usually called the sample space, and is often interpreted as *time*, of either discrete ($\mathbb{N}$) or continuous ($\mathbb{R}_{\geqslant 0}$) nature. A specific continuous probability distribution function $F : \mathbb{R}_{\geqslant 0} \mapsto [0, 1]$ defined by $F(t) = 1 - e^{-\lambda t}$ is depicted below.



Intuitively, $F(t)$ is the probability $\mathsf{Prob}(D \leqslant t)$ that a duration D has finished at time t the latest. This specific distribution is called an exponential distribution with rate $\lambda \in \mathbb{R}_{\geqslant 0}$. Evidently, a rate uniquely characterises an exponential distribution. Note that $F(0) = 0$ (the duration surely does not finish in zero time), and $\lim_{t\to\infty} F(t) = 1$ (the duration eventually finishes).

The class of exponential distributions has some important properties that explain their prominent rôle in contemporary performance evaluation. We try to summarise them here. First, we note that the mean value of an exponential distributed duration is the reciprocal $1/\lambda$ of its rate λ. Secondly, an exponential distribution of rate λ is the most appropriate approximation[1] of a random phenomenon of which only the mean value $(1/\lambda)$ is known. Furthermore, exponential distributions possess the so-called *memory-less property*: If D is exponentially distributed then $\mathsf{Prob}(D \leqslant t+t' \mid D > t) = \mathsf{Prob}(D \leqslant t')$. This means that if we observe that D is not finished at time t, and are interested in $F(t+t')$ *under this condition*, then this is just $F(t)$: The distribution is invariant under the passage of time. In this sense, it does not possess memory. In fact, the exponential distribution is the only continuous probability distribution function that possesses this property. Other relevant properties of exponential distributions for this paper are closure properties of exponentially distributions with respect to maximum

[1] In information theoretic jargon, such an approximation maximises the entropy.

and minimum. If we are waiting for several durations to finish, then we are essentially waiting for the maximum of these durations. Exponential distributions are not closed under maximum; the maximum of several (pairwise stochastic independent) exponentially distributed durations is a phase-type distribution. If, on the other hand, we are only waiting for one out of several competing durations, the situation is different. Awaiting the minimum of several (pairwise stochastic independent) exponentially distributed durations D_i (with rate λ_i, $i \in \{0..n\}$) is itself exponentially distributed. Its rate parameter is the sum of the individual rates $\sum_{i=1}^{n} \lambda_i$. In this case the probability that a specific D_j finishes first in the race is given by $\lambda_j / \sum_{i=1}^{n} \lambda_i$.

2.2 Continuous-Time Markov Chains

For the purpose of this paper, a continuous time Markov chain can be viewed as a finite state machine, where transitions are labelled with rates of exponential distributions. Intuitively, such state machines evolve as follows. Whenever a state is entered, a race starts between several exponentially distributed durations, given by the (rate labels of) transitions leaving this state. As soon as some duration D_j finishes, the state machine moves to the target state of the transition belonging to this D_j (labelled by λ_j). Clearly, a certain successor state is chosen with probability $\lambda_j / \sum_{i=1}^{n} \lambda_i$, and the time until this happens is exponentially distributed with rate $\sum_{i=1}^{n} \lambda_i$.

The memory-less property carries over from the distributions to the Markov chain[2]: If we know that the chain has been in the current state for some time already (or that it was in a specific state at a specific time in the past), then this knowledge is irrelevant for its future behaviour. The chain's behaviour is history independent, only the identity of the state currently occupied is decisive for the future behaviour.

Usually, a CTMC is characterised by its so-called generator matrix $\mathbf{Q}$ and its initial distribution. The entries of the generator matrix $\mathbf{Q}$ specify the transition rates: $\mathbf{Q}(s, s')$ denotes the rate of moving from state s to state s', where $s \neq s'$.

Definition 1. *(Generator matrix) For some finite set of states S, a square matrix $\mathbf{Q}$ of size $|S| \times |S|$ is the (infinitesimal) generator matrix of a CTMC iff, for all $s \in S$, $\mathbf{Q}(s, s') \geqslant 0$ $(s \neq s')$, and $\mathbf{Q}(s, s) = -\sum_{s' \neq s} \mathbf{Q}(s, s')$.*

While the off-diagonal entries of this matrix specify *individual* rates of *entering* a new state, the diagonal entries specify the converse, a *cumulative* rate of *leaving* the current state, so to speak. Together with an initial state (or an initial probability distribution on states) the generator matrix gives all the necessary information to determine the transient and steady-state probabilistic behaviour of the chain.

[2] The standard definition of CTMCs proceeds in the reverse way, i.e., it defines a memory-less discrete-state continuous-time stochastic process, and derives from this that exponential distributions need to govern its behaviour.

Example 1. In the leftmost column of Fig. 1 we have depicted the generator matrix of a CTMC by means of the usual state-transition representation, where s and s' are connected by a transition labelled λ iff $\mathbf{Q}(s, s') = \lambda > 0$. The initial state is coloured black. In order to illustrate how the probability mass spreads over states as time passes, we represent it as pie-charts of black colour. All black colour is initially (at time 0) in state s_0. From left to right the figure depicts snapshots at different times, where the pie-charts indicate the amount of probability $\pi_{s'}(s_0, t)$ of being in state s' at time t. The rightmost column depicts the limits of these probabilities as $t \to \infty$.

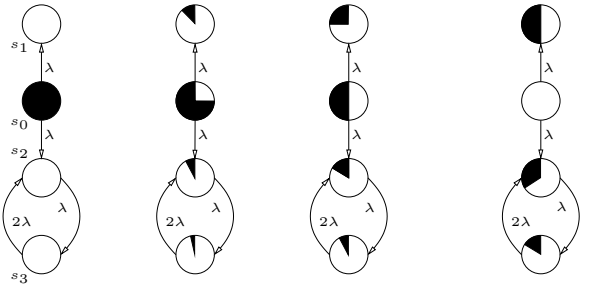


Fig. 1. Transient and steady-state behaviour of a CTMC (from left to right: transient probabilities $\underline{\pi}(s_0, 0)$, $\underline{\pi}(s_0, \ln(4/3)/(2\lambda))$, $\underline{\pi}(s_0, \ln(2)/(2\lambda))$, and steady-state probability $\underline{\pi}(s)$).

The vector $\underline{\pi}(s, t) = (\pi_{s'}(s, t))_{s' \in S}$ is the *transient probability* vector at time t if starting in state s at time 0. The vector $\underline{\pi}(s) = (\lim_{t \to \infty} \pi_{s'}(s, t))_{s' \in S}$ is called the *steady-state probability* vector. Such a limit exists for arbitrary finite CTMCs, and may depend on the starting state. Efficient numerical algorithms exist to compute steady-state as well as transient probability vectors [27].

Lumpability. We conclude this section by a remark on an important concept that allows one to aggregate states of a CTMC without affecting transient and steady-state probabilities. This concept, called *lumpability* is defined as follows [40,15].

Definition 2. *(Lumpability.) For $\mathcal{S} = \{ S_1, \ldots, S_n \}$ a partitioning of the state space S of a CTMC, the CTMC is* lumpable *with respect to $\mathcal{S}$ if and only if for any partition $S_i \subseteq S$ and states $s, s' \in S_i$:*

$$\forall 0 < k \leqslant n.\ \sum_{s'' \in S_k} \mathbf{Q}(s, s'') = \sum_{s'' \in S_k} \mathbf{Q}(s', s'').$$

That is, for any two states in a given partition the cumulative rate of moving to any other partition needs to be equal. Under this condition, the performance measures of a CTMC and its lumped quotient are strongly related. First, the

quotient stochastic process (defined on a state space $\mathcal{S}$) is a CTMC as well. In addition, the probability of the lumped CTMC being in the quotient state S_i equals the sum of the probability of being in any of the original states $s \in S_i$ in the original chain. This correspondence holds for transient and steady-state probabilities.

3 Process Algebra for CTMCs

In this section we discuss various issues one faces when designing a process algebraic formalism for CTMCs. We only summarise the crucial considerations, and refer to [29,12] for more elaborate discussions.

3.1 CTMC Algebra

To begin with we introduce a small, action-less process algebra to generate CTMCs.

Syntax. Let X be drawn from a set of process variables, and I drawn from a set of finite sets of indices. Furthermore let $\lambda_i \in \mathbb{R}_{\geqslant 0}$ for $i \in I$. The syntax of the algebra MC is

$$P ::= \sum_{i \in I} (\lambda_i) \,.\, P \;\Big|\; X \;\Big|\; recX.P$$

The term $recX.P$ defines a recursive process X by P, that possibly contains occurrences of X. If I consists of two elements we use binary choice $+$, if I is empty we write $\mathbf{0}$. The meaning of summation is as follows: For I a singleton set, the term $(\lambda) \,.\, P$ denotes a process that evolves into P within t time units $(t \geqslant 0)$ according to an exponential distribution of rate λ. That is, it behaves like P after a certain delay D that is determined by $\mathsf{Prob}(D \leqslant t) = 1 - e^{-\lambda t}$ for positive t.[3] In general, the term $\sum_{i \in I} (\lambda_i) \,.\, P_i$ offers a *timed probabilistic* choice among the processes P_i. As in a CTMC, a race is assumed among competing delays. Intuitively, a successor state P_j is entered with probability $\lambda_j / \sum_{i \in I} \lambda_i$, and the time until this happens is exponentially distributed with rate $\sum_{i \in I} \lambda_i$. We restrict MC to closed expressions given by the above grammar.

Semantics. The structured operational semantics of MC is presented below. The inference rules define a mapping of this algebra onto CTMCs (as we will see).

$$\sum_{i \in I} (\lambda_i) \,.\, P_i \overset{\lambda_j}{\longmapsto}_j P_j \quad (j \in I) \qquad\qquad \frac{P\{\, recX.P/X \,\} \overset{\lambda}{\longmapsto}_i P'}{recX.P \overset{\lambda}{\longmapsto}_i P'}$$

[3] The prefix $(\lambda) \,.\, P$ can be considered as the probabilistic version of the timed prefix $(t) \,.\, P$ that typically occurs in timed process algebras, like in TCCS [44] or in Timed CSP [50].

The rule for recursion is standard; we just recall that $P\{\, Q/X \,\}$ denotes term P in which all (free) occurrences of process variable X in P are replaced by Q. The rule for choice requires some explanation. Consider $\sum_{i\in I} (\lambda_i)\,.\,P_i$. At execution, the fastest process, that is, the process that is enabled first, is selected. This is reflecting the race condition described above. The probability of choosing a particular alternative, P_j say, equals $\lambda_j / \sum_{i\in I}\ \lambda_i$, assuming that summands with distinct indices are distinct.

The transitions are decorated with an auxiliary label indicated as subscript of the transition relation. It is used to distinguish between different deduction trees of a term. In absence of such mechanism, we would, for instance, for $(\lambda_1)\,.\,P + (\lambda_2)\,.\,P$, obtain two distinct transitions, except if $\lambda_1 = \lambda_2$. In that specific case we would obtain two different deduction trees for the same transition labelled λ_1 (or λ_2); this, however, does suggest that P can be reached with rate λ_1 (or λ_2), whereas this should be rate $\lambda_1{+}\lambda_2$. A similar mechanism is rather standard in probabilistic process calculi like PCCS [24].

The above operational semantics maps a term onto a transition system where transitions are labelled by rates. It is not difficult to check that by omitting self-loops and replacing the set of transitions from s to s' by a single transition with the sum of the rates of the transitions from s to s', a CTMC is obtained.

Example 2. The leftmost CTMC in Fig. 1 is generated from the semantics of

$$(\lambda)\,.\,\mathbf{0} + (\lambda)\,.\,recX.(\lambda)\,.\,(2\lambda)\,.\,X$$

Lumping equivalence. Lumping equivalence is defined in the same style as Larsen-Skou's probabilistic bisimulation [41] and Hillston's strong equivalence [36]. Let $\{\!|\ \ldots\ |\!\}$ denote multi-set brackets.

Definition 3. *(Lumping equivalence.) An equivalence relation $\mathcal{S}$ on* MC *is a* lumping equivalence *iff for any pair $(P,Q) \in \mathsf{MC}\times\mathsf{MC}$ we have that $(P,Q)\in\mathcal{S}$ implies for all equivalence classes $C \in \mathsf{MC}/_{\mathcal{S}}$:*

$$\gamma(P,C) = \gamma(Q,C) \ \textit{with}\ \gamma(R,C) = \sum_i \{\!|\, \lambda \mid R \stackrel{\lambda}{\longmapsto}_i R', R' \in C \,|\!\}.$$

Processes P and Q are lumping equivalent*, denoted $P\sim Q$, if $(P,Q)\in\mathcal{S}$ with $\mathcal{S}$ a lumping equivalence.*

Here, we use $\mathsf{MC}/_{\mathcal{S}}$ to denote the set of equivalence classes induced by $\mathcal{S}$ over MC. Stated in words, P and Q are lumping equivalent if the total rate of moving to equivalence class C under $\sim$ is identical for all such classes. As the name suggests, this bisimulation-style definition is in close correspondence to the concept of lumpability on CTMCs (cf. Def. 2). As first pointed out by Buchholz [16] and Hillston [36] (in settings similar to ours) $P\sim Q$ if and only if their underlying CTMCs can be partitioned into isomorphic lumpable partitionings.

Example 3. The term $(\lambda)\,.\,\mathbf{0} + (\lambda)\,.\,recX.(\lambda)\,.\,(2\lambda)\,.\,X$ is equivalent to the chain

$$(\lambda)\,.\,\mathbf{0} + (\lambda)\,.\,recX.\left(\left(\frac{1}{3}\lambda\right)\,.\,(2\lambda)\,.\,X + \left(\frac{2}{3}\lambda\right)\,.\,(2\lambda)\,.\,(\lambda)\,.\,(2\lambda)\,.\,X\right)$$

To illustrate this, both chains are depicted in Fig. 2. Let $\mathcal{S}$ be the equivalence relation containing (exactly) those pairs of states in Fig. 2 that are shaded with identical patterns. It is easy to check that $\mathcal{S}$ is a lumping equivalence, and it equates the initial states. Thus, the latter can be lumped into the former.

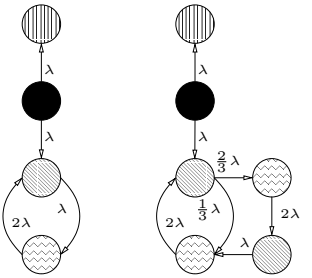


Fig. 2. Lumping equivalence classes.

The fact that lumpability is nowadays known to have a bisimulation-style (coinductive) definition is a prime example for cross-fertilisation from concurrency theory to performance evaluation. In particular, partition refinement algorithms for bisimulation can be adapted to carry out the *best possible lumping* on finite CTMCs [35]. This improves performance evaluation of large CTMCs, where the question of how to determine a lumpable partitioning of the state space (let alone the best possible one) was for a long time based on modeller's ingenuity and experience.

Equational theory. Since it can be shown that lumpability is a congruence with respect to the operators of MC, we may strive for a sound and complete axiomatisation of $\sim$ for MC. Such an axiomatisation facilitates the lumping of CTMCs at a syntactic level. The axioms for sequential finite terms are listed as (B1) through (B4) below.

$$\text{(B1) } P + Q = Q + P \qquad \text{(B2) } (P + Q) + R = P + (Q + R) \qquad \text{(B3) } P + \mathbf{0} = P$$
$$\text{(B4)} \qquad (\lambda + \mu)\,.\,P = (\lambda)\,.\,P + (\mu)\,.\,P$$

The axioms (B1) through (B3) are well known from classical process calculi. Axiom (B4) is a distinguishing law for our calculus and can be regarded as a replacement in the Markovian setting of the traditional idempotency axiom for choice ($P + P = P$). Axiom (B4) reflects that the resolution of choice is modelled by the minimum of (statistically independent) exponential distributions.

Together with standard laws for handling recursion on classical process calculi these axioms can be shown to form a sound and complete axiomatisation of MC.

Interleaving. To illustrate another feature of CTMCs from the concurrency theoretical perspective, we add a simple parallel composition operator to our calculus, denoted by $||$. Intuitively, the term $P \,||\, Q$ can evolve while either P evolves or Q evolves independently from (and concurrently to) each other. Due to the memory-less property of CTMCs, the behaviour of parallel CTMCs can be interleaved. This is different from a deterministic time setting where parallel processes typically are forced to synchronise on the advance of time, as in TCCS [44]. The operational rules are:

$$\frac{P \overset{\lambda}{\longmapsto}_i P'}{\begin{array}{l} P \,||\, Q \overset{\lambda}{\longmapsto}_{(i,*)} P' \,||\, Q \\ Q \,||\, P \overset{\lambda}{\longmapsto}_{(*,i)} Q \,||\, P'. \end{array}}$$

(Notice that we create new auxiliary labels of the form $(i, *)$ and $(*, i)$ in order to obtain a multi-transition relation.) To understand the meaning of the memory-less property in our context consider the process $(\lambda) \,.\, P \,||\, (\mu) \,.\, Q$ and suppose that the delay of the left process finishes first (with rate λ). Due to the memory-less property, the remaining delay of Q is determined by an exponential distribution with (still!) rate μ, exactly the delay prior to the enabling of these delays before the delay of the first process has been finished. Therefore, the parameters of transitions do not need any adjustment in an interleaving semantics. One of the consequences of this independent delaying is that an expansion law is obtained rather straightforwardly. For $P = \sum_i (\lambda_i) \,.\, P_i$ and $Q = \sum_j (\mu_j) \,.\, Q_j$ we have:

$$P \,||\, Q = \sum_i (\lambda_i) \,.\, (P_i \,||\, Q) + \sum_j (\mu_j) \,.\, (P \,||\, Q_j).$$

3.2 Interaction in CTMCs

The algebra MC is lacking any notion of action, and hence provides only a restricted way of specifying CTMCs in a compositional way. For instance, interaction between different parallel chains cannot be adequately described. Two different approaches can be identified when it comes to the integration of actions into CTMC algebra. One way [25,36,10] is to combine delays and actions in a single *compound* prefix $(a, \lambda) \,.\, P$. The intuitive meaning of this expression is that the process $(a, \lambda) \,.\, P$ is ready to engage in action a after a delay determined by an exponential distribution with rate λ and afterwards behaves like P. We refer to [12] for a discussion of this approach. Here we focus on a different, orthogonal approach [28,11] where the action prefix $a \,.\, P$ known from standard process algebra is added to CTMC algebra, to complement the existing delay prefix $(\lambda) \,.\, P$ with separate means to specify actions.

Syntax. We equip this new algebra, called IMC (Interactive Markov Chains) with a TCSP-style parallel composition operator parametrised with a set A of synchronising actions. Accordingly we have:

$$P ::= \sum_{i \in I} a_i \,.\, P + \sum_{i \in I'} (\lambda_i) \,.\, P \;\Big|\; X \;\Big|\; recX.P \;\Big|\; P \,||_A\, P$$

Semantics. The semantics is given by the rules listed for MC (where $||$ is now understood as $||_A$ for an arbitrary set A of actions) plus the standard rules known from process algebra: (In mixed summations like $a \,.\, P + (\lambda) \,.\, Q$ the respective summation rules are applied elementwise.)

$$\sum_{i \in I} a_i \,.\, P_i \xrightarrow{a_j} P_j \quad (j \in I) \qquad \frac{P\{\, recX.P/X \,\} \xrightarrow{a} P'}{recX.P \xrightarrow{a} P'}$$

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l} P \,||_A\, Q \xrightarrow{a} P' \,||_A\, Q \\ Q \,||_A\, P \xrightarrow{a} Q \,||_A\, P' \end{array}} \quad (a \notin A) \qquad \frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \,||_A\, Q \xrightarrow{a} P' \,||_A\, Q'} \quad (a \in A)$$

Equational theory. Since the calculus extends both standard process algebra and CTMC algebra, it is possible to integrate bisimulation and lumping equivalence. This can be done for strong, weak and other bisimulation equivalences [28]; we omit the operational definitions here for brevity. Instead we characterise the resulting equational theory for weak (lumping) bisimulation congruence by the set of axioms satisfied by finite expressions. It is given by the axioms (B1) through (B4) listed for CTMC algebra, and the following additional laws.

(B5) $a \,.\, P = a \,.\, P + a \,.\, P$ (P1) $(\lambda) \,.\, P + \tau \,.\, Q = \tau \,.\, Q$

(τ1) $\tau \,.\, P = P + \tau \,.\, P$ (τ2) $a \,.\, (P + \tau \,.\, Q) = a \,.\, (P + \tau \,.\, Q) + a \,.\, Q$
(τ3) $a \,.\, P = a \,.\, \tau \,.\, P$ (τ4) $(\lambda) \,.\, P = (\lambda) \,.\, \tau \,.\, P$

Axiom (B5) replaces the traditional idempotence axiom for choice ($P + P = P$) which is not sound for delay prefixed expressions (cf. axiom (B4)). The (P1) axiom realises *maximal progress*: A process that has something internal to do will do so, without letting time pass. No time will be spent in the presence of an internal action alternative. The axioms (τ1) through (τ3) are well known for weak bisimulation on standard process algebra [43], and (τ4) extends (τ3) to delay prefixed expressions. Together with additional laws to handle recursion (and divergence [34]), this set gives rise to a sound and complete axiomatisation for finite state IMC, illustrating that process algebra smoothly extends to this orthogonal union of CTMC algebra and process algebra. We refer to [28] for further discussion.

3.3 Time Constraints and Phase-Type Distributions

In this section, we illustrate two important features of IMC. We show how more general continuous probability distributions can be embedded into the calculus, and we illustrate how such general distributions can be used to constrain the behaviour of an IMC in a modular, constraint-oriented style. The approach presented here is detailed out in [31].

Phase-type distributions. Phase-type distributions can be considered as matrix generalisations of exponential distributions, and include frequently used distributions such as Erlang, Cox, hyper- and hypo-exponential distributions. Intuitively, a phase-type distribution can be considered as a CTMC with a single absorbing state (a state with $\mathbf{Q}(s, s') = 0$ for all s). The time until absorption determines the phase-type distribution [45]. In terms of CTMC algebra, phase-type distributions can be encoded by explicitly specifying the structure of the CTMC using summation, recursion, and termination ($\mathbf{0}$), as in the MC term $\tilde{Q}$ given by $(\lambda)\,.\,recX.(\mu)\,.\,(\mu)\,.\,X + (\lambda)\,.\,\mathbf{0}$. The possibility of specifying phase-type distributions is of significant interest, since phase-type distributions can approximate arbitrary distributions arbitrarily close [45] (i.e., it is a dense subset of the set of continuous distributions). In other words, MC and IMC can be used to express arbitrary distributions, by choosing the appropriate absorbing Markov chain, and (mechanically) encoding it in MC.

Time constraints. In IMC, phase-type distributions can govern the timing of actions. The main idea is to intersperse action sequences (such as $a\,.\,b\,.\,\mathbf{0}$) with specific phase-type distributions (such as the above $\tilde{Q}$) in order to delay the occurrences of the actions in the sequence appropriately, such as delaying the occurrence of b after a by $\tilde{Q}$. This can be achieved by explicitly changing the structure of the process into $a\,.\,(\lambda)\,.\,recX.(\mu)\,.\,(\mu)\,.\,X + (\lambda)\,.\,b\,.\,\mathbf{0}$, but this approach will be cumbersome in general.

To enhance specification convenience, we introduce the *elapse* operator that is used to impose phase-type distributed time constraints on specific occurrences of actions. The elapse operator facilitates the description of such time constraints in a *modular* way, that is, as separated processes that are constraining the behaviour by running in parallel with an untimed (or otherwise time-constrained) process. To introduce this operator, we use much of the power of process algebra, since the semantics of the operator is defined by means of a translation into the basic operators of IMC. Due to the compositional properties of IMC, important properties (congruence results, for instance) carry over to this operator in a straightforward manner.

We shall refer to a time constraint as a delay that necessarily has to elapse between two kinds of actions, unless some action of a third kind occurs in the meanwhile. In order to facilitate the definition of such time constraints, the elapse operator is an operator with four parameters, syntactically denoted by [**on** S **delay** D **by** Q **unless** B]:

- a phase-type distribution Q (represented as an MC term) that determines the duration of the time constraint,
- a set of actions S (start) that determines when the delay (governed by Q) starts,
- a set of actions D (delay) which have to be delayed, and
- a set of actions B (break) which may interrupt the delay.

Thus, for instance, [**on** $\{a\}$ **delay** $\{b\}$ **by** $\tilde{Q}$ **unless** $\varnothing$] imposes the delay of $\tilde{Q}$ between a and b. We claim that a wide range of practical timing scenarios can be covered by this operator (in particular if non-empty intersections between the action sets are allowed). This is illustrated in [31] where this operator is used to impose various time constraints on an existing, untimed process algebraic specification (of more than 1500 lines of LOTOS code) of the plain ordinary telephone system.

Semantically, the intuition behind this operator is that it enriches the chain Q with some synchronisation potential, that is used to initialise and reset the time constraint in an appropriate way. The time constraint is imposed on a process P by means of parallel composition, such as in

$$P \,||_{S \cup D \cup B}\, [\textbf{on } S \textbf{ delay } D \textbf{ by } Q \textbf{ unless } B].$$

The elapse operator is an auxiliary operator that can be defined using sequential composition and disrupt, LOTOS-operators that can be easily added to IMC [31]. For instance, the semantics of $a \,.\, b \,.\, \mathbf{0} \,||_{\{a,b\}}\, [\textbf{on } \{a\} \textbf{ delay } \{b\} \textbf{ by } \tilde{Q} \textbf{ unless } \varnothing]$ agrees with $a \,.\, (\lambda) \,.\, recX.(\mu) \,.\, (\mu) \,.\, X + (\lambda) \,.\, b \,.\, \mathbf{0}$ up to weak bisimulation.

3.4 Compositional Aggregation

Interactive Markov chains can be used to specify CTMCs, but due to the presence of nondeterminism (inherited from standard process algebra), the model underlying IMC is richer, it is the class of continuous time Markov decision chains [49], a strict superset of CTMCs. Nondeterminism is one of the vital ingredients of process algebra and hence of IMC, though it appears as an additional hurdle when it comes to performance evaluation, because the stochastic behaviour may be underspecified. In order to eliminate nondeterminism – and to aggregate the state space – we have developed a general recipe leading from an IMC specification to a CTMC:

1. Develop a specification of the system under investigation using the operators provided by IMC. A possible approach is to start from an existing process algebraic specification and to enrich the specification by incorporating time constraints. The elapse operator is convenient for this purpose.
2. Close the specification by abstracting from all actions using the standard abstraction (encapsulation) operator of process algebra.
3. Apply weak bisimulation congruence to aggregate (lump) the state space, to eliminate action transitions, and to remove nondeterminism. Due to the

congruence property, this aggregation step is preferably done compositionally, by applying it to components of the specification prior to composition. In this way, the state space explosion problem can be diminished [31].

If the aggregated, minimal transition system does not contain action transitions, it trivially corresponds to a lumped CTMC. If, on the other hand, the resulting transition system still contains action transitions the stochastic process is under-specified, it is a continuous time Markov decision chain, because nondeterminism is present. The above recipe has been exercised in practice successfully [31]. The necessary algorithms for state space generation, and efficient aggregation are implemented [30,17], and compositional aggregation is supported.

4 Model Checking CTMCs

Once a CTMC has been generated, the next step is to evaluate the measure(s) of interest such as time to failure, system throughput or utilisation, with the required accuracy. In this section, we use temporal logic to express constraints (i.e., bounds) on such measures and show how model checking techniques can be employed for the automated analysis of these constraints. We only summarise the crucial considerations, and refer to [7,5] for more elaborate discussions.

4.1 CTMC Temporal Logic

To specify performance and dependability measures as logical formulas over CTMCs, we assume the existence of a set AP of atomic propositions with $a \in AP$ and extend CTMCs with a labelling function $L : S \to 2^{AP}$ which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in s. These labelled CTMCs can be viewed as Kripke structures with transitions labelled by rates.

Syntax. CSL (Continuous Stochastic Logic) is a branching-time temporal logic à la CTL [23] based on [4] that is interpreted on CTMCs. Let p be a probability ($p \in [0,1]$) and $\trianglelefteq$ a comparison operator, i.e., $\trianglelefteq \in \{\leqslant, \geqslant\}$. CSL state-formulas are constructed according to the following syntax:

$$\Phi ::= a \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathcal{S}_{\trianglelefteq p}(\Phi) \mid \mathcal{P}_{\trianglelefteq p}(\varphi)$$

The two probabilistic operators $\mathcal{S}$ and $\mathcal{P}$ refer to the steady-state and transient behaviour, respectively, of the CTMC being studied. Whereas the steady-state operator $\mathcal{S}$ refers to the probability of residing in a particular set of *states* (specified by a state-formula) on the long run, the transient operator $\mathcal{P}$ allows us to refer to the probability of the occurrence of particular *paths* in the CTMC, similar to [26]. The operator $\mathcal{P}_{\trianglelefteq p}(.)$ replaces the usual CTL path quantifiers $\exists$ and $\forall$. In fact, for most cases (up to fairness) $\exists\varphi$ can be written as $\mathcal{P}_{>0}(\varphi)$ and $\forall\varphi$ as $\mathcal{P}_{\geqslant 1}(\varphi)$. For instance, $\mathcal{P}_{>0}(\Diamond a)$ is equivalent to $\exists\Diamond a$ and $\mathcal{P}_{\geqslant 1}(\Diamond a)$ stands for $\forall\Diamond a$ given a fair interpretation of the CTL-formula $\forall\Diamond a$.

For I an interval on the real line ($I \subseteq \mathbb{R}_{\geqslant 0}$), the syntax of CSL path-formulas is

$$\varphi ::= \mathcal{X}^I\ \Phi \ \big|\ \Phi\, \mathcal{U}^I\ \Phi.$$

The operators $\mathcal{X}^I$ and $\mathcal{U}^I$ are the timed variants of the usual next-operator and until-operator, respectively. Similar timed variants of these operators appear in timed CTL [2].

Semantics. State-formulas are interpreted over the states of a CTMC; for s a state of the CTMC under consideration and Φ a state-formula, $s \models \Phi$, if and only if Φ is valid in s. The semantics of the Boolean operators is standard (i.e., $s \models a$ iff $s \in L(s)$, $s \models \neg\Phi$ iff $s \not\models \Phi$, and $s \models \Phi_1 \vee \Phi_2$ iff $s \models \Phi_1 \vee s \models \Phi_2$.) The state-formula $\mathcal{S}_{\trianglelefteq p}(\Phi)$ asserts that the steady-state probability for the set of Φ-states meets the bound $\trianglelefteq p$:

$$s \models \mathcal{S}_{\trianglelefteq p}(\Phi) \quad \text{if and only if} \quad \sum_{s' \models \Phi} \pi_{s'}(s) \trianglelefteq p$$

where we recall that $\pi_{s'}(s)$ equals $\lim_{t\to\infty} \pi_{s'}(s,t)$, where $\pi_{s'}(s,t)$ denotes the probability to be in state s' at time t when starting in state s. Finally, $\mathcal{P}_{\trianglelefteq p}(\varphi)$ asserts that the probability measure of the paths satisfying φ meets the bound $\trianglelefteq p$. Let $Prob(s,\varphi)$ denote the probability of all paths satisfying φ when the system starts in state s. (The probability measure $Prob$ is formally defined in [7].) Then:

$$s \models \mathcal{P}_{\trianglelefteq p}(\varphi) \quad \text{if and only if} \quad Prob(s,\varphi) \trianglelefteq p$$

A path σ in a CTMC is an alternating sequence of the form $s_0\, t_0\, s_1\, t_1\, \ldots$ where $t_i \in \mathbb{R}_{\geqslant 0}$ indicates the amount of time stayed in state s_i.[4] Let $\sigma[i]$ denote the $(i{+}1)$-state in σ and let $\sigma@t$ denote the state occupied by σ at time t. The satisfaction relation for the path-formulas is defined as follows. The path-formula $\mathcal{X}^I\,\Phi$ asserts that a transition is made to a Φ-state at some time point $t \in I$:

$$\sigma \models \mathcal{X}^I\ \Phi \quad \text{if and only if} \quad \sigma[1] \models \Phi\ \wedge\ \delta(\sigma,0) \in I$$

where $\delta(\sigma,0) = t_0$, the duration of staying in the initial state s_0 of σ. The path-formula $\Phi\,\mathcal{U}^I\,\Psi$ asserts that Ψ is satisfied at some time instant in the interval I and that at all preceding time instants Φ holds:

$$\sigma \models \Phi_1\,\mathcal{U}^I\,\Phi_2 \quad \text{if and only if} \quad \exists t \in I.\ (\sigma@t \models \Phi_2\ \wedge\ \forall u \in [0,t).\, \sigma@u \models \Phi_1)\,.$$

The usual (untimed) next- and until-operator are obtained as $\mathcal{X}\,\Phi = \mathcal{X}^{[0,\infty)}\,\Phi$, and $\Phi\,\mathcal{U}\,\Psi = \Phi\,\mathcal{U}^{[0,\infty)}\,\Psi$. Other Boolean connectives are derived in the usual way (e.g. $\Phi \vee \Psi = \neg(\neg\Phi \wedge \neg\Psi)$). Temporal operators like $\Diamond^I$ ("eventually in I") and $\Box^I$ ("always in I") are derived by, for example: $\mathcal{P}_{\trianglelefteq p}(\Diamond^I\,\Phi) = \mathcal{P}_{\trianglelefteq p}(\mathrm{tt}\,\mathcal{U}^I\,\Phi)$ and $\mathcal{P}_{\trianglerighteq p}(\Box^I\,\Phi) = \mathcal{P}_{\leqslant 1-p}(\Diamond^I\,\neg\Phi)$.

[4] For simplicity, we assume all paths to be infinite.

Expressiveness. Besides standard state-based performance measures such as steady-state and transient-state probabilities, the logic-based approach allows one to specify bounds on the occurrence probability of certain (sets of) paths. We exemplify the type of properties that one can express using CSL by considering a simple re-configurable fault tolerant system. The system can be either *Up* or *Down*, and it may (or may not) be in a phase of (initial or re-)configuration (*Config*). Thus we consider $AP = \{ Up, Down, Config \}$.

Example 4. Assume that the states of the CTMC in Fig. 1 are labelled – from top to bottom – by $\{ Down \}$, $\{ Up, Config \}$, $\{ Up \}$, and $\{ Down, Config \}$. For instance, $L(s_0) = \{ Up, Config \}$.

As an overview of some well-known performance and dependability measures [51] and their formulation in terms of CSL we list the following CSL formulas:

(a)	steady-state availability	$\mathcal{S}_{\trianglelefteq p}(Up)$
(b)	transient configuration probability at time t	$\mathcal{P}_{\trianglelefteq p}(\Diamond^{[t,t]} Config)$
(c)	instantaneous availability at time t	$\mathcal{P}_{\trianglelefteq p}(\Diamond^{[t,t]} Up)$
(d)	distribution of time to failure	$\mathcal{P}_{\trianglelefteq p}(Up\, \mathcal{U}^{[0,t]}\, Down)$

Measure (a) expresses a bound on the steady-state availability of the system and (b) expresses a bound on the transient-state probability of (re-)configuring the system at time t. Measure (c) states (a bound on) the probability to be in a non-failed state at time t, i.e., the instantaneous availability at time t and (d) expresses, indirectly, the time until a failure, starting from a non-failed state. That is, evaluating this measure for varying t, gives the distribution of the time to failure.

The above standard transient measures are expressed using only simple instances of the $\mathcal{P}$-operator. However, since this operator allows an arbitrary path-formula as argument, much more general measures can be described and nested.

Example 5. An example of an interesting non-standard measure is the probability of reaching a certain set of states provided that all paths to these states obey certain properties. For instance,

$$\neg Config \Rightarrow \mathcal{P}_{\geqslant 0.99}(Up\ \mathcal{U}^{[0,20]}\ Config)$$

states that the probability to turn from a non-configuring state into a reconfiguration in no more than 20 time units without any system down time on the way is more than 99%. As another example, we may require that in the steady-state, there is a chance of at most 10% that a down time is likely (that is, has more than half of the probability) to occur within 5 and 10 time units from now.

$$\mathcal{S}_{\leqslant 0.1}(\mathcal{P}_{>0.5}(\Diamond^{[5,10]} Down))$$

Lumpability revisited. In the same spirit as the relations between bisimulation and CTL (and CTL*) equivalence [14] and between Larsen-Skou's probabilistic bisimulation and PCTL-equivalence [3], there exists a relation between lumping equivalence and CSL-equivalence. This is illustrated by means of a slight variant of the earlier treated notion of lumping equivalence, cf. Def. 2.

Definition 4. *(F-Lumpability.) For* $\mathcal{S} = \{ S_1, \ldots, S_n \}$ *a partitioning of the state space* S *of a CTMC and* F *a set of CSL state-formulas, the CTMC is* F-lumpable *with respect to* $\mathcal{S}$ *if and only if for any partition* $S_i \subseteq S$ *and states* $s, s' \in S_i$*:*

$$\forall 0 < k \leqslant n.\ \sum_{s'' \in S_k} \mathbf{Q}(s, s'') = \sum_{s'' \in S_k} \mathbf{Q}(s', s'')$$

and

$$\{ \Phi \mid s \models \Phi \} \cap F = \{ \Phi \mid s' \models \Phi \} \cap F.$$

That is, for any two states in a given partition the cumulative rate of evolving to another partition (like before) must be equal, and the set of formulas in F that are fulfilled must coincide. Clearly, if a CTMC is F-lumpable with respect to some partitioning of its state space, then it is also lumpable. A CTMC and its lumped quotient are strongly related with respect to the validity of CSL formulae. In particular, a (state in a) CTMC and its (quotient state in the) AP-lumped quotient – obtained by the above notion where the set F equals the set of atomic propositions – satisfy the same CSL-formulas [5]. This result can be exploited by aggregating the CTMC as far as possible during checking the validity of CSL-formulas, or prior to this process by considering its quotient with respect to the coarsest AP-lumping.

4.2 CTMC Model Checking

There are two distinguishing benefits when using CSL for specifying constraints on measures-of-interest over CTMCs: (i) the specification is entirely formal such that the interpretation is unambiguous, and (ii) it allows the possibility to state performance and dependability requirements over a selective set of paths (similar to [47]) through a model. These features are paired with the (practically most relevant) possibility to check CSL-formulas in a completely automated manner. This can be done by combining model checking techniques with numerical solution techniques. The basic procedure is as for model checking CTL: in order to check whether state s satisfies the formula Φ, we recursively compute the sets $Sat(\Psi) = \{s' \in S \mid s' \models \Psi\}$ of all states that satisfy Ψ, for the subformulas Ψ of Φ, and eventually check whether $s \in Sat(\Phi)$. For atomic propositions and Boolean connectives this procedure is exactly the same as for CTL. Next and (unbounded) until-formulas can be treated in a similar way as in the discrete-time probabilistic setting [26]. Checking steady-state properties reduces to solving a system of linear equations combined with standard graph analysis methods [7].

Fixed-point characterisation. Most interesting (and complicated) though is the handling of time-bounded until-formulas, as their treatment require to deal with the interplay of timing and probabilities. For the sake of simplicity, we treat the case $I = [0, t]$; the general case is a bit more involved, but can be treated in a similar way [5]. Let $\varphi_t = \Phi\, \mathcal{U}^{[0,t]}\, \Psi$. We have from the semantics that

$$s \in Sat(\mathcal{P}_{\trianglelefteq p}(\varphi_t)) \ \text{ if and only if } \ Prob(s, \varphi_t) \trianglelefteq p$$

The probability $Prob(s, \varphi_t)$ is the least solution of the following set of equations:

$$Prob(s,\varphi_t) = \begin{cases} 1 & \text{if } s \in Sat(\Psi) \\ 0 & \text{if } s \notin Sat(\Phi) \cup Sat(\Psi) \\ \displaystyle\int_0^t \sum_{s' \in S} \mathbf{T}(s,s',x) \cdot Prob(s', \varphi_{t-x})\, dx & \text{otherwise} \end{cases}$$

where $\mathbf{T}(s, s', x)$ denotes the density of moving from state s to state s' in x time-units and can be derived from the matrix $\mathbf{Q}$. The first two cases are self-explanatory; the last equation is explained as follows. If s satisfies Φ but not Ψ, the probability of reaching a Ψ-state from s within t time-units equals the probability of reaching some direct successor state s' of s within x time-units ($x \leqslant t$), multiplied by the probability to reach a Ψ-state from s' in the remaining time-span $t-x$.

This recursive integral characterisation provides the theoretical basis for model checking time-bounded until-formulas over CTMCs in the same way as the fixed-point characterisations for CTL provide the basis for the model checking algorithms for usual until-formulas [18].

Algorithmic procedure. To illustrate how performance evaluation recipes can be exploited for model checking purposes, we now sketch an efficient and numerically stable strategy for model checking the time-bounded until-formulas [5]. As lumping preserves the validity of all CSL-formulas, a first step is to switch from the original state space to the (possibly much smaller) quotient space under lumping. Next, prior to computing the exact set of states that satisfy φ_t, the states fulfilling the (fair) CTL-formula $\exists(\Phi\,\mathcal{U}\,\Psi)$ is determined. For states not in this set, the respective probabilistic until-formula will have probability 0. In a similar way, the set of states satisfying $\forall(\Phi\,\mathcal{U}\,\Psi)$ (up to fairness, cf. [8]) is computed; these states satisfy φ_t with probability 1. As a result, the actual computation of the system of Volterra integral equations needs to be done only for the remaining states. How to do this? The basic idea is to employ a transformation of the CTMC and the formula at hand, such that a transient analysis problem is obtained for which well-known and efficient computation techniques do exist. This idea is based on the observation that formulas of the form $\mathcal{P}_{\trianglelefteq p}(\Diamond^{[t,t]}\Phi)$ characterise transient probability measures, and their validity (in some state s) can be decided on the basis of the transient probability vector $\underline{\pi}(s,t)$. This vector can be calculated by transient analysis techniques. For $\mathcal{P}_{\trianglelefteq p}(\Phi\ \mathcal{U}^{[0,t]}\,\Psi)$ the CTMC $\mathcal{M}$ under consideration is transformed into another CTMC $\mathcal{M}'$ such that checking $\varphi_t = \Phi\,\mathcal{U}^{[0,t]}\,\Psi$ on $\mathcal{M}$ amounts to checking $\varphi'_t = \Diamond^{[t,t]}\Psi$ on $\mathcal{M}'$; a transient analysis of $\mathcal{M}'$ (for time t) then suffices. The question then is, how do we transform $\mathcal{M}$ in $\mathcal{M}'$? Concerning a $(\Phi \wedge \neg\Psi)$-state, two simple observations form the basis for this transformation:

- once a Ψ-state in $\mathcal{M}$ has been reached (along a Φ-path) before time t, we may conclude that φ holds, regardless of which states will be visited afterwards. This justifies making all Ψ-states absorbing.

– once a state has been reached that neither satisfies Φ nor Ψ, φ is violated regardless of which states will be visited afterwards. This justifies making all $\neg(\Phi \wedge \Psi)$-states absorbing.

It then suffices to carry out a transient analysis on the resulting CTMC $\mathcal{M}'$ for time t and collect the probability mass to be in a Ψ-state (note that $\mathcal{M}'$ typically is *smaller* than $\mathcal{M}$):

$$Prob^{\mathcal{M}}(s, \Phi\, \mathcal{U}^{[0,t]}\, \Psi) = Prob^{\mathcal{M}'}(s, \Diamond^{[t,t]}\Psi) = \sum_{s' \models \Psi} \pi_{s'}(s,t).$$

In fact, by similar observations it turns out that also verifying the general $\mathcal{U}^I$-operator can be reduced to instances of (a two-phase) transient analysis [5].

Example 6. In order to check one of the above mentioned requirements on the CTMC of Fig. 1, one needs to check $s_2 \models \mathcal{P}_{\geqslant 0.99}(\mathit{Up}\ \mathcal{U}^{[0,20]}\ \mathit{Config})$. To decide this, it is sufficient to compute $\underline{\pi}(s_2, 20)$ on a CTMC where state s_0 and s_3 (as well as s_1) are made absorbing, and to check $\pi_{s_0}(s_2, 20) + \pi_{s_3}(s_2, 20) \geqslant 0.99$.

The transformation of the model checking problem for the time-bounded until-operator into the transient analysis of a CTMC has several advantages: (i) it avoids awkward numerical integration, (ii) it allows us to use efficient and numerically stable transient analysis techniques, such as uniformisation [38], and (iii) it employs a measure-driven transformation (aggregation) of the CTMC. The fact that a dedicated and well-studied technique in performance evaluation such as uniformisation can be employed for model checking is a prime example for the cross-fertilisation from performance evaluation to concurrency theory.

Efficiency. The worst-case time complexity of model checking CSL is

$$\mathcal{O}(|\Phi| \cdot (M \cdot q \cdot t_{max} + N^{2.81}))$$

where M is the number of non-zero entries in $\mathbf{Q}$, q is the maximal diagonal entry of $\mathbf{Q}$, t_{max} is the maximum time bound of the time-bounded until sub-formulas occurring in Φ, and N is the number of states. If we make the practically often justified assumption that $M < kN$ for a constant k then the space complexity is linear in N using a sparse matrix data structure. The space complexity is polynomial in the size of the CTMC. The model checking algorithms have been implemented both using sparse matrix data structures [32] and using BDD-based data structures [39].

5 Research Perspectives

This paper has presented how two important branches of formal methods for reactive systems – process algebra and model checking – can be exploited for performance and dependability modelling and analysis. The stochastic process

algebra approach is a prominent example of cross-fertilisation of formal specification techniques and performance modelling techniques, whereas the quantitative model checking approach is a promising combination of computer-aided verification technology and performance analysis techniques. We believe that the developments in these areas mark the beginning of a new paradigm for the modelling and analysis of systems in which qualitative and quantitative aspects are studied from an integrated perspective. We hope that the further work towards the realisation of this goal will be a growing source of inspiration and progress for both communities. Examples of issues for future work in this direction are:

- *Specification:* in order to bridge the gap towards (performance) engineers, and to obtain a better integration into the design cycle, efforts should be made to the usage of (appropriate extensions of) specification languages such as UML and SDL for the description of performance models. Similarly, the usage of temporal logic by performance engineers needs to be simplified, for instance, using dedicated specification patterns [22].
- *Verification:* similar to the development of model checking techniques, smart improvements of both algorithms and data structures are needed to make the verification approach more successful. Initial investigations show that symbolic data structures (such as multi-terminal BDDs) and tailored variants of existing techniques ("backwards" uniformisation) yield a substantial efficiency improvement [39]. Promising alternative techniques, such as Kronecker representations [21], and/or refinement techniques for probabilistic systems as recently proposed in [20] could be beneficial in this context as well.
- *Extensions:* several extensions of the process algebra and model checking techniques are worthwhile to investigate. For instance, Markov reward models – an important extension of CTMCs with costs – are not yet satisfactorily treated in a process algebraic or logic-based setting, although some initial attempts have been made [6,9]. In addition, the application of model checking to non-memoryless models, such as (generalised) semi-Markov processes, remains an interesting topic for further research. Initial work in this direction is reported in [37].

Finally, we highlight two gaps between the process algebraic and model checking approach we discussed: (i) whereas the formal model specifications are behaviour-oriented (i.e., action based), the temporal logic approach is state-based, and (ii) the semantic model of the process algebra may contain nondeterminism, whereas the verification is based on a fully probabilistic model. The first problem can be handled by considering an action-based variant of CSL. Although it turns out that a transformation of this logic into CSL (à la the relationship between CTL and its action-based variant [46]) is possible, it is more beneficial to use direct model checking techniques – basically a tailored version of the CSL model checking algorithms. Details are in [33]. This yields a combination with stochastic process algebras that treat actions and stochastic delays as a single compound entity [10,25,36]. In order to close the gap w.r.t. our process algebra IMC that strictly distinguishes between action occurrences

and time delays, we are currently investigating model checking procedures for continuous-time Markov decision chains.

Acknowledgements. Ed Brinksma (Univ. Twente) and Ulrich Herzog (Univ. Erlangen) have contributed to the work on stochastic process algebra reported here. The model checking research reported in this paper has been developed in collaboration with Christel Baier (Univ. Bonn) and Boudewijn Haverkort (RWTH Aachen). The first author is supported by the Netherlands Organisation of Scientific Research (NWO).

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
3. A. Aziz, V. Singhal, F. Balarin, R. Brayton and A. Sangiovanni-Vincentelli. It usually works: the temporal logic of stochastic systems. In *CAV'95*, LNCS 939:155–165. Springer, 1995.
4. A. Aziz, K. Sanwal, V. Singhal and R. Brayton. Model checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1): 162–170, 2000.
5. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous time Markov chains by transient analysis. In *CAV 2000*, LNCS 1855:358–372. Springer, 2000.
6. C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *ICALP 2000*, LNCS 1853:780–792. Springer, 2000.
7. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *CONCUR'99*, LNCS: 1664:146–162. Springer, 1999.
8. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11:125–155, 1998.
9. M. Bernardo. An algebra-based method to associate rewards with EMPA terms. In *ICALP'97*, LNCS 1256:358–368. Springer, 1997.
10. M. Bernardo and R. Gorrieri. A tutorial on EMPA: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.
11. H.C. Bohnenkamp and B.R. Haverkort. Semi-numerical solution of stochastic process algebra models. In *ARTS'99*, LNCS 1601:228–243. Springer, 1999.
12. E. Brinksma and H. Hermanns. Process algebra and Markov chains. In [13].
13. E. Brinksma, H. Hermanns, and J.-P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis*, LNCS 2090. Springer, 2001.
14. M. Brown, E. Clarke, O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59: 115–131, 1988.
15. P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *Journal of Applied Probability*, 31–75:59–75, 1994.
16. P. Buchholz. Markovian Process Algebra: composition and equivalence. In U. Herzog and M. Rettelbach, editors, *Proc. of PAPM'94*, Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg, 1994.

17. M. Cherif, H. Garavel, and H. Hermanns. bcg_min – Minimization of normal, probabilistic, or stochastic labeled transitions systems encoded in the BCG format. `http://www.inrialpes.fr/vasy/cadp/man/bcg_min.html`.
18. E. Clarke, E. Emerson and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8: 244–263, 1986.
19. A.E. Conway and N.D. Georganas. *Queueing Networks: Exact Computational Algorithms*. MIT Press, 1989.
20. P.R. D'Argenio, B. Jeannet, H.E. Jensen, and K.G. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *PAPM/PROBMIV'01*, LNCS. Springer, 2001. To appear.
21. M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.
22. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In *Formal Methods in Software Practice*. ACM Press, 1998.
23. E.A Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2: 241–266, 1982.
24. R.J. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121:59–80, 1995.
25. N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *Tutorial Proc. of PERFORMANCE '93*, LNCS 729:121-146. Springer, 1993.
26. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing* 6: 512–535, 1994.
27. B. Haverkort. Markovian models for performance and dependability evaluation. In [13].
28. H. Hermanns. *Interactive Markov Chains*. PhD thesis, Universität Erlangen-Nürnberg, September 1998. Arbeitsberichte des IMMD 32/7.
29. H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 2001. To appear.
30. H. Hermanns, U. Herzog, U. Klehmet, M.Siegle, and V. Mertsiotakis. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39(1-4):5–35, 2000.
31. H. Hermanns and J.-P. Katoen. Automated compositional Markov chain generation for a plain-old telephony system. *Science of Computer Programming*, 36(1):97–127, 2000.
32. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In *TACAS 2000*, LNCS 1785:347–362, 2000.
33. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. Towards model checking stochastic process algebra. In *IFM 2000*, LNCS 1945:420–439. Springer, 2000.
34. H. Hermanns and M. Lohrey. Priority and maximal progress are completely axiomatisable. In *CONCUR'98*, LNCS 1466:237–252. Springer, 1998.
35. H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In *ARTS'99*, LNCS 1601:244–264. Springer, 1999.
36. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
37. G.G. Infante-Lopez, H. Hermanns, and J.-P. Katoen. Beyond memoryless distributions: model checking semi-Markov chains. In *PAPM/PROBMIV'01*, LNCS. Springer, 2001. To appear.

38. A. Jensen. Markov chains as an aid in the study of Markov processes. *Skand. Aktuarietidskrift*, 3: 87–91, 1953.
39. J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *PAPM/PROBMIV'01*, LNCS. Springer, 2001. To appear.
40. J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer, 1976.
41. K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.
42. J.F. Meyer, A. Movaghar and W.H. Sanders. Stochastic activity networks: structure, behavior and application. In *Proc. Int. Workshop on Timed Petri Nets*, pp. 106–115, IEEE CS Press, 1985.
43. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
44. F. Moller and C. Tofts. A temporal calculus for communicating systems. In *CONCUR'90*, LNCS 458:401–415. Springer, 1990.
45. M.F. Neuts. *Matrix-geometric Solutions in Stochastic Models–An Algorithmic Approach*. The Johns Hopkins University Press, 1981.
46. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Concurrency*, LNCS 469: 407–419, 1990.
47. W.D. Obal and W.H. Sanders. State-space support for path-based reward variables. *Performance Evaluation*, 35: 233–251, 1999.
48. B. Plateau and K. Atif, Stochastic automata networks for modeling parallel systems. *IEEE Transactions on Software Engineering*, 17(10): 1093–1108, 1991.
49. M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.
50. S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116:193–213, 1995.
51. R.M. Smith, K.S. Trivedi and A.V. Ramesh. Performability analysis: measures, an algorithm and a case study. *IEEE Trans. on Comp.*, 37(4): 406–417, 1988.

Typing Mobility in the Seal Calculus

Giuseppe Castagna[1], Giorgio Ghelli[2], and Francesco Zappa Nardelli[1,2]

[1] C.N.R.S., Département d'Informatique
École Normale Supérieure, Paris, France
[2] Dipartimento di Informatica
Università di Pisa, Pisa, Italy

Abstract. The issue of this work is how to type mobility, in the sense that we tackle the problem of typing not only mobile agents but also their movement. This yields higher-order types for agents. To that end we first provide a new definition of the Seal Calculus that gets rid of existing inessential features while preserving the distinctive characteristics of the Seal model. Then we discuss the use of interfaces to type agents and define the type system. This type system induces a new interpretation of the types: interfaces describe interaction *effects* rather than, as it is customary, provided *services*. We discuss at length the difference of the two interpretations and justify our choice of the former.

1 Introduction

In concurrent languages based on communication over channels it is customary to type both channels and messages in order to assure that only appropriate messages will transit over channels of a given type. When these languages are endowed with *agents* and *locations*, we also need typing information about the agents that are moved around the locations. Hence, we have to decide what is described by the type of an agent, and when this type is checked.

Our proposal is expressed in terms of a simplified version of the *Seal Calculus*. The Seal Calculus was defined in [19] as a set of primitives for a secure language of mobile agents to be used to develop commercial distributed applications at the University of Geneva; these primitives constitute the core of the JavaSeal language [18,2]. It can be considered a safety-oriented calculus. From the Ambient Calculus [6], it borrows the idea that locations are places with a "boundary", which can only be crossed with some effort. In the Seal Calculus, boundaries can only be crossed when two parties, one inside the boundary and the other one outside, agree. Moreover, this movement operation takes place over a support layer, constituted by a set of channels. Communication takes place over channels too, as in the π-calculus [11], and the dangerous *open* operation of the Ambient Calculus is not present.

In our proposal the type of an agent is a description of the requests that it may accept from its enclosing environment. This is similar to an object type, in an object-oriented language; however, we will show that the subtype relation goes "the other way round". We will discuss this fact, which means that, while an object type describes a subset of the *services* that an object offers, our interface

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 82–101, 2001.
© Springer-Verlag Berlin Heidelberg 2001

types describe a superset of the *effects* of an agent on its environment. We shall be more precise about it later on, but for the time being the reader can think of *services* as interactions that *must eventually* occur and of *effects* as the interactions that *may possibly* occur.

Our main results are the following ones. First, we define a variant of the Seal Calculus, which retains its essential security-oriented features but is simple enough to be suited to foundational studies. Then, in this context, we define a type system where we are able both to type mobile agents and to "type their mobility", i.e., to allow one to declare, for each location, the types of the agents that can enter or exit it. This yields to the first, as far as we know, higher order type system for agent mobility.

The article is structured as follows. In Section 2 we define our variant of the Seal Calculus. In Section 3 we introduce the typed calculus and justify our choices. In Section 4 we define the type system, a sound and complete type-checking algorithm, and we state some relevant properties they satisfy. In Section 5 we analyze our system and discuss the duality of *effects* vs. *services*. Section 6 describes an example that uses the key features of our calculus, while in Section 7 we hint at how our work can be used to provide the an Java agent kernel with a minimal type system. A summary and directions for future work conclude the article.

Related Work

Many works extend the basic type systems for π-calculus as described in [12,15] giving more informative types to processes. We comment those closest to our work.

Yoshida and Hennessy propose in [20] a type system for a higher-order π-calculus that can be used to control the effects of migrating code on local environments. The type of a process takes the form of an interface limiting the resources to which it has access, and the type at which they may be used. In their type system both input and output channels can appear in the interface, appearing strictly more expressive than the system we propose here, where input channels are only considered. However, they do not allow active agents, but only pieces of code, to be sent over a channel. When code is received it can be activated, possibly after parameter instantiation. Besides, their type system limits the application of dependent types to the instantiation of parameters, resulting in the impossibility of giving an informative type to processes in which an output operation depends on an input one.

In [8] Hennessy and Riely define Dπ, a distributed variant of the π-calculus where agents are "located" (i.e., "named") threads. The main difference with respect to our work is that locations cannot be nested (that is, locations are not threads), and therefore mobility in [8] consist in spawning passive code rather than migrating active agents. In [8] locations types have the form $\mathsf{loc}\{x_1 : T_1, \ldots x_n : T_n\}$ where x_i's are channels belonging to the location (they are located resources and as such Dπ is much closer to the Seal Calculus as defined in [19], than to the version we introduce here: see Footnote 2). These types are syntactically very close to those we introduce here for Seal but they

have opposed interpretations. Location types in [8] are intended both to describe the *services* provided by a location and to regulate access to them (they work as *views* in databases). Thus they embrace an object-oriented perspective (location types are subtyped as record types) without fully assuming it (services are declared but they are not ensured to be provided[1]). As we broadly discuss in the following—in particular in Section 5— our location types take the opposite perspective and rather describe the *effects* of a possible interaction with the location at issue.

Types for locations have been extensively studied in a series of papers [7,5,4] on the Ambient Calculus. Seal Calculus differs from Ambient Calculus in many aspects: seals cannot be opened (i.e. they boundaries cannot be dissolved), many channels exist, and moves are performed *objectively* (i.e., agents are passively moved by their environment) rather than *subjectively* (i.e., agents autonomously decide to move to a different location) —according to the terminology of [6]. From this work's viewpoint the prominent difference between Ambient and Seal Calculus is that the former does not provide an explicit "physical" support for mobility, while in the latter this support is provided by channels. In other words while in Ambients mobility take place on some unmaterialized *ætheral* transport medium, in Seal the medium is materialized by channels. Therefore the main novelty of this work is that not only we type locations (agents or seals), but we also type mobility (more precisely, its support). In some sense we introduce higher-order typing: while in Ambient Calculus an agent can not discriminate which agents can traverse its boundaries, this is possible in our type system. For the same reason we can make a mobile location become immobile, while this is not possible in the cited works on Ambient Calculus. Moreover, the mobility model of Ambient Calculus had to be extended with objective moves in [4], since the interaction of subjective moves with the *open* operation tends to produce typings where every ambient is typed as mobile. We show here that the mobility model of Seal Calculus is free from this problem.

2 Revising Untyped Seal Calculus

Seal Calculus is basically a π-calculus extended with *nested named locations* (dubbed *seals*) and mobility primitives. In Seal, interaction consists of synchronous communication of a value or of a whole seal. Both forms of interaction take place over named channels. Thus, mobility is obtained by communicating a seal on a channel. The existence of separate locations constraints the possible interactions: a channel can only allow interactions either among processes in the same seal, or among processes in two seals that are in parent-child relationship.

Two basic security principles underlay the design of the Seal Calculus: first, each seal must be able to control all interactions of its children, both with the outside world and one with the other; second, each seal must have total control over its name-space and therefore must determine the names of its children.

Besides these two basic features the Seal Calculus defined in [19] included some other features dictated by implementation issues. More precisely the cal-

[1] This is to solve the type dependency problem we describe in Section 4.1.

culus in [19] allowed seal duplication and destruction, and a strictly regulated access to remote channels[2].

In what follows we define a lighter version of Seal where seal creation and destruction is not possible and the access to remote channels is replaced by the introduction of shared channels[3].

The syntax of the language (parametric on an infinite set of *names*, ranged over by u, v, x, y, and z) is defined as follows:

Processes			**Actions**			**Locations**		
P	::=	$\mathbf{0}$ inactivity	α	::=	$x^{\eta}(y)$ input	η	::=	$*$ local
	□	$P \mid P$ composition		□	$\overline{x}^{\eta}(y)$ output		□	$\uparrow$ up
	□	$!P$ replication		□	$\overline{x}^{\eta}\langle y\rangle$ send		□	z down
	□	$(\boldsymbol{\nu}\, x)P$ restriction		□	$x^{\eta}\langle y\rangle$ receive			
	□	$\alpha.P$ action						
	□	$x[P]$ seal						

The first five process constructs have the same meaning as in the π-calculus, namely: the $\mathbf{0}$ process does nothing, the composition $P \mid Q$ denotes two processes P and Q running in parallel, the replication $!P$ unleashes an unbounded number of copies of P, the restriction $(\boldsymbol{\nu}\, x)P$ introduces a new name x and limits its scope to P (the scoping is lexical), and the prefix allows one to construct complex processes using the base actions α. A seal $x[P]$ is the representation in the syntax of a place named x that is delimited by boundaries and where the computation P takes place. The bare syntax of processes is the same as Ambient Calculus.

The basic computational steps in Seal are *communication* and *movement*. Communications (inputs/outputs on channels) are as in π-calculus with the only difference that channel names are super-scripted by location denotations. These are either $*$, or $\uparrow$, or z, and denote respectively the current seal (i.e. the seal where the action occurs), the parent seal, and a child-seal named z. Thus an action on x^* synchronizes only with local processes, $x^{\uparrow}$ means that x is a channel shared between the current seal and the parent seal and that actions on it will synchronize with processes in the parent, and finally the shared channel x^z admits interactions between the current seal and a child-seal named z. These interactions are expressed by the first three rules in Figure 1.

Mobility is achieved in a similar way: seal bodies, rather than names, are moved over channels. It should be remarked that, contrary to input, receive is not a binding action: y is free in $x^{\eta}\langle y\rangle$. A seal identified by its name is sent over a localized named channel: the seal together with its contents will disappear from the location of the sending processes and will reappear in the location of the receiving process. The receiving process can give a new name to the received seal: seal names are seen as local pointers to the location, and the actual name of a seal makes no sense outside the current location. Thus the action $\overline{x}^{\eta}\langle y\rangle$

[2] In [19] channels are considered as resources. Each channel belongs to one and only one seal. Some syntactic constructs allow the owner of a channel to regulate remote accesses to it and, thus, to control both remote communication and mobility.

[3] A similar solution was independently proposed for a calculus without agent mobility in [17].

$$
\begin{array}{rll}
x^*(u).P \mid \overline{x}^*(v).Q \rightarrow P\{^v/_u\} \mid Q & & \text{(write local)} \\
x^y(u).P \mid y[\overline{x}^{\uparrow}(v).Q \mid R] \rightarrow P\{^v/_u\} \mid y[Q \mid R] & & \text{(write out)} \\
\overline{x}^y(v).P \mid y[x^{\uparrow}(u).Q \mid R] \rightarrow P \mid y[Q\{^v/_u\} \mid R] & & \text{(write in)} \\
 & & \\
x^*\Box u\Box.P \mid \overline{x}^*\Box v\Box.Q \mid v[R] \rightarrow P \mid u[R] \mid Q & & \text{(move local)} \\
x^y\Box u\Box.P \mid y[\overline{x}^{\uparrow}\Box v\Box.Q \mid v[R] \mid S] \rightarrow P \mid u[R] \mid y[Q \mid S] & & \text{(move out)} \\
\overline{x}^y\Box v\Box.P \mid v[R] \mid y[x^{\uparrow}\Box u\Box.Q \mid S] \rightarrow P \mid y[Q \mid S \mid u[R]] & & \text{(move in)}
\end{array}
$$

Fig. 1. Reduction rules.

sends the body of the seal named y over the channel x^η, while $x^\eta\Box y\Box$ waits for a body on x^η and reactivates it as a seal named y. The precise semantics is given by the last three rules in Figure 1.

As customary, reduction uses structural congruence $\equiv$ that is the smallest congruence that is a commutative monoid with operation $\mid$ and unit $\mathbf{0}$, and is closed for the following rules:

$$
\begin{array}{lll}
!P \equiv !P \mid P & (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & (\boldsymbol{\nu}\, x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}\, x)Q \quad \text{for } x \notin fn(P) \\
(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & & (\boldsymbol{\nu}\, x)y[P] \equiv y[(\boldsymbol{\nu}\, x)P] \quad \text{for } x \neq y
\end{array}
$$

The reduction semantics is completed by standard rules for context and congruence:

$$
\begin{array}{lll}
P \rightarrow Q \;\Rightarrow\; (P \mid R) \rightarrow (Q \mid R) & & P \rightarrow Q \;\Rightarrow\; (\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q \\
P \rightarrow Q \;\Rightarrow\; u[P] \rightarrow u[Q] & & P \equiv P' \;\wedge\; P' \rightarrow Q' \;\wedge\; Q' \equiv Q \;\Rightarrow\; P \rightarrow Q
\end{array}
$$

3 Typing Mobility

In the introduction we anticipated that our solution for typing mobility was to type the transport media of mobility, that is, channels. We follow the standard π-calculus solution to type channels: a channel named x has type `Ch` V if V is the type of the *values* allowed to transit over x. We saw that in Seal channels are used both for communication (in which case they transport *messages*, i.e., base values, or names) and mobility (in which case they transport *agents*, i.e., seal bodies).

It is easy to type base values (in this work the only base values we consider are synchronization messages typed by `Shh`) and we just saw how to type channel names. So to define V we still have to define the type A of agents and, consequently, the type `Id` A of names denoting agents of type (more precisely, of interface) A.

3.1 Intuition about Interfaces

Seals are named agents. The idea is to type them by describing all interactions a seal may have with the surrounding environment. We know that such interactions have to take place over the channels that cross the seal boundary. Thus these channels partially specify the interaction protocol of an agent. Keeping track of the set of upward communications (that is, communications with the parent) that a seal may establish can be statically achieved by keeping track of the channels that would be employed: this gives rise to a notion of interface of an

agent as a set of *upward channels* (i.e., characterized by ↑ locations). Actually not all the upward channels are interesting for describing the interaction protocol of a seal. Those the seal is listening on suffice:

> *The* interface *of a seal is the set of upward channels that the process local to the seal may be listening on, with the type expected from interactions on them.*

We will discuss this choice later on (see Sections 4.1 and 5) but, for the moment, to see why such a definition is sensible we can consider as an example a networked machine. For the outer world the interface of such a machine —the description of how it is possible to interact with it— is given by the set of ports on which a dæmon is listening, together with the type of messages that will be accepted on them. So the interface of a machine can be described as a set of pairs (*port*:*type*). For example in our system a typical ftp and mail server would be characterized by a type of the following form [21:*ftp*; 23:*telnet*; 79:*finger*; 110:*pop3*; 143:*imap*; ...]. Similarly, if you consider a seal as an object, then a process contained in it that listens on a upward channel $x^{\uparrow}$ can be assimilated to a method associated with a message x. In other words the sending of a message m with argument v to an object x (that is, `x.m(v)` in Java syntax) can be modeled in Seal by the action $\overline{m}^{x}(v)$, which would be allowed, in our type system, by a pair $m{:}M$ in the type of the seal x.

Hence, we consider interfaces such as $[x_1{:}\texttt{Shh}; x_2{:}\texttt{Ch}\ V; x_3{:}A; x_4{:}\texttt{Id}\ A]$ that characterizes agents that may: 1) synchronize with an input operation on the upward channel x_1; 2) read over x_2 a channel name of type $\texttt{Ch}\ V$ (the name of a channel that transports messages of type V); 3) receive over x_3 a seal whose interface is A; 4) read over x_4 a seal name of type $\texttt{Id}\ A$. It is important to stress the difference between what can be transmitted over x_3 and x_4, respectively seals and seal names: the former requires mobility primitives, the latter communication primitives.

3.2 Syntax

The syntax of the types is reported in the following table.

Types

V	::=	M	messages
	▯	A	agents

Annotations

Z	::=	$\curvearrowright$	mobile
	▯	$\underline{\vee}$	immobile

Message Types

M	::=	$\texttt{Shh}$	silent
	▯	$\texttt{Ch}\ V$	channel names
	▯	$\texttt{Id}^{Z}A$	agent names

Interfaces

A	::=	$[x_1{:}V_1; \cdots ; x_n{:}V_n]$	agents

There are four syntactic categories in the type syntax, V, M, Z, and A respectively denoting types, message types, mobility annotations and agents. In the previous section we informally described three of them, omitting annotations. Let us see them all in more detail:

V: Types V classify *values*, that is computational entities that can be sent over a channel. While in π-calculus values are just channel names, in Seal we have both messages (classified by message types M) —which includes base values, channel names and agent names— and seals (more precisely seal's bodies, classified by interfaces A).

M: Message types M classify *messages*, that is entities that can be *communicated* (sent by an i/o operation) over channels. A message can be either a synchronization message (without any content) of type $\mathtt{Shh}$, or a name. In the syntax there is no distinction between channel names and seal names. This distinction is done at the type level: if $x : \mathtt{Ch}\ V$, then x is the name of a channel that transports values of type V; if $x : \mathtt{Id}^Z A$, then x is the name of a seal with interface A and with mobility attribute Z.

Z: On the lines of [4] we use mobility attributes to specify elemental mobility properties of seals: a $\curvearrowright$ attribute characterizes a mobile seal, while a $\veebar$ attribute characterizes an immobile one. Being able to discriminate between mobile and immobile agents is one of the simplest properties related to mobility. Contrary to what happens in [4], adding this feature does not require any syntax modification for Seal.

A: Interfaces A classify the *agents* of the calculus, keeping track of their interface. The notation $[x_1{:}V_1; \cdots ; x_n{:}V_n]$ is used to record the information about the interface of an agent. It is syntactic sugar for a set of pairs *channel_name* : *type*, that represent a functional relation. If a process has this interface, then it can be enclosed in an agent with the same interface, that is whose name has type $\mathtt{Id}^Z[x_1{:}V_1; \cdots ; x_n{:}V_n]$. This agent may listen from the upward level only on channels $x_1, \ldots, x_n$.

The introduction of types requires a minimal modification to the syntax of the untyped calculus of Section 2. We have to add (message) type annotations to the two binders of the language: $(\boldsymbol{\nu}\, x{:}M)$ and $x^\eta(y{:}M)$, and to redefine free names *fn* as follows.

$$\begin{array}{lll} fn(x) = \{x\} & fn((\boldsymbol{\nu}\, x{:}M)P) = (fn(P) \setminus \{x\}) \cup fn(M) & fn(\uparrow) = fn(*) = fn(\mathtt{Shh}) = \emptyset \\ fn(x^\eta(y{:}M).P) = (fn(P) \setminus \{y\}) \cup fn(M) \cup fn(\eta) \cup \{x\} & & fn(\mathtt{Id}\ A) = fn(A) \\ fn([x_1{:}V_1; \ldots ; x_n{:}V_n]) = \{x_1, \ldots, x_n\} \cup fn(V_1) \cup \cdots \cup fn(V_n) & & fn(\mathtt{Ch}\ V) = fn(V) \end{array}$$

The rule of structural congruence that swaps binders has to be changed too

$$(\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P \equiv (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P \qquad \text{for } x \notin fn(M') \wedge y \notin fn(M) \wedge x \neq y$$

The reduction rules as well as the other rules and definitions are unchanged.

4 The Type System

In this section we define the type system we informally described in the previous section. However, before that, we need to add a last ingredient in order to deal with the technical problem of type dependencies.

4.1 Type Dependencies

The notion of interface introduces names of the calculus at the type level. Since names are first order terms, type dependencies may arise. Consider for example the following terms.

$$P' = x^*(y{:}\mathtt{Ch}\ M).y^{\uparrow}(z{:}M) \qquad P = \overline{x}^*(w) \mid P'$$

P' offers upwards input on channel y. Hence, a naive syntax based analysis would associate P' and P with the interface $[y{:}M]$, producing the following typing judgment:

$$x{:}\mathtt{Ch}(\mathtt{Ch}\ M)\ ,\ y{:}\mathtt{Ch}\ M\ ,\ w{:}\mathtt{Ch}\ M \vdash P : [y{:}M].$$

However, the process P may perform an internal reduction on the channel x, and then it would offer upwards input on channel w, hence changing its interface type:

$$\underbrace{\overline{x}^*(w) \mid x^*(y{:}\mathtt{Ch}\ M).y^{\uparrow}(z{:}M)}_{[y:M]} \quad \rightarrow \quad \underbrace{w^{\uparrow}(z{:}M)}_{[w:M]}$$

This is the recurrent problem when trying to define non-trivial channel-based types for processes: to solve it one may consider using dependent types and deal explicitly with types that change during computation. Dependent types work fine for calculi where the notion of interaction is syntactically well-determined, as in λ-calculus. Unfortunately in process calculi, where interaction is a consequence of parallel composition (which admits arbitrary rearrangements of sub-terms), all the tries are somewhat unsatisfactory: they are usually restricted to a subset of the calculus, allowing dependent types only in particular, well-determined constructions [20].

Following a suggestion of Davide Sangiorgi, we decide to disallow input action on names bound by an input action. In this way interfaces cannot change during reduction: for example the process P above is not well-typed, since y is first bound by an input on x and then used to perform an input from $\uparrow$.

To make the type system uniform, we impose this condition on all the input operation, not only on the input operations from $\uparrow$, which are the only ones determining the interface.

There is no harm in doing that since this restriction does not limit the expressive power of the calculus: besides being theoretically well studied (see for example [10]), nearly all programming languages based on π-calculus impose this constraint, while programs written in concurrent languages that do not, mostly seem to obey to the same condition.

4.2 Typing Rules

Judgments have the form $\Gamma \vdash_{\Xi} \Im$, where $\Im$ is either $\diamond$, or V, or $x\colon M$, or $P\colon A$. The pair Γ, Ξ will be referred to as *typing environment* and the judgments have the following standard meaning:

$\Gamma \vdash_{\Xi} \diamond$	well-formed environment	$\Gamma \vdash_{\Xi} V$	well-formed type
$\Gamma \vdash_{\Xi} x{:}M$	x has message type M	$\Gamma \vdash_{\Xi} P : A$	P has interface A

Γ is a function (actually, an ordered list) that assigns types to names. At the same time, we need some machinery to enforce the restriction on input channels we described above, that is, that only names *not* bound by an input action (i.e., names introduced by $\boldsymbol{\nu}$) are used to perform input operations. Thus we use the set of names Ξ to record the $\boldsymbol{\nu}$-introduced names:

$$\Gamma ::= \varnothing \;\square\; \Gamma, x{:}M \qquad\qquad \Xi ::= \varnothing \;\square\; \Xi, x$$

The typing and subtyping rules are:

(Env Empty)
$$\frac{}{\varnothing \vdash_{\varnothing} \diamond}$$

(Env Add)
$$\frac{\Gamma \vdash_{\Xi} M}{\Gamma, x{:}M \vdash_{\Xi} \diamond} \quad x \notin \mathrm{dom}(\Gamma, \Xi)$$

(Env Add Xi)
$$\frac{\Gamma \vdash_{\Xi} M}{\Gamma, x{:}M \vdash_{\Xi, x} \diamond} \quad x \notin \mathrm{dom}(\Gamma)$$

(Type Shh)
$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} \mathtt{Shh}}$$

(Type Id)
$$\frac{\Gamma \vdash_{\Xi} A}{\Gamma \vdash_{\Xi} \mathtt{Id}^{Z} A}$$

(Type Ch)
$$\frac{\Gamma \vdash_{\Xi} V}{\Gamma \vdash_{\Xi} \mathtt{Ch}\ V}$$

(Type Interface)
$$\frac{\Gamma \vdash_{\Xi} \diamond \quad \forall i \in 1..n \quad \Gamma \vdash_{\Xi} x_i{:}\mathtt{Ch}\ V_i \quad x_i \in \mathrm{dom}(\Xi)}{\Gamma \vdash_{\Xi} [x_1{:}V_1, \ldots, x_n{:}V_n]}$$

(Var)
$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} x{:}\,\Gamma(x)}$$

(Dead)
$$\frac{\Gamma \vdash_{\Xi} \diamond}{\Gamma \vdash_{\Xi} \mathbf{0} : []}$$

(Par)
$$\frac{\Gamma \vdash_{\Xi} P_1 : A \qquad \Gamma \vdash_{\Xi} P_2 : A}{\Gamma \vdash_{\Xi} P_1 \mid P_2 : A}$$

(Bang)
$$\frac{\Gamma \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} !P : A}$$

(Res)
$$\frac{\Gamma, x{:}M \vdash_{\Xi, x} P : A}{\Gamma \vdash_{\Xi} (\boldsymbol{\nu}\, x{:}M)P : A} \quad x \notin fn(A)$$

(Seal)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Id}^{Z} A \qquad \Gamma \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} x[P] : []}$$

(Output Local)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma \vdash_{\Xi} y : M \qquad \Gamma \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} \overline{x}^{*}(y).P : A}$$

(Input Local)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma, y{:}M \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} x^{*}(y{:}M).P : A} \quad x \in \mathrm{dom}(\Xi)$$

(Output Up)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma \vdash_{\Xi} y : M \qquad \Gamma \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} \overline{x}^{\uparrow}(y).P : A}$$

(Input Up)
$$\frac{\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma, y{:}M \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} x^{\uparrow}(y{:}M).P : (A \oplus [x{:}M])} \quad x \in \mathrm{dom}(\Xi)$$

(Output Down)
$$\frac{\Gamma \vdash_{\Xi} z : \mathtt{Id}^{Z} A' \qquad \Gamma \vdash y : M \qquad \Gamma \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} \overline{x}^{z}(y).P : A} \quad (x{:}M) \in A'$$

(Input Down)
$$\frac{\Gamma \vdash_{\Xi} z : \mathtt{Id}^{Z} A' \qquad \Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M \qquad \Gamma, y{:}M \vdash_{\Xi} P : A}{\Gamma \vdash_{\Xi} x^{z}(y{:}M).P : A} \quad x \in \mathrm{dom}(\Xi)$$

(Rcv Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi x^*\Box y\Box.P : A'} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^\frown A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi \overline{x}^*\Box y\Box.P : A'}$$

(Rcv Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi x^\uparrow\Box y\Box.P : (A' \oplus [x{:}A])} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\ A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^\frown A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi \overline{x}^\uparrow\Box y\Box.P : A'}$$

(Rcv Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z'} A' \qquad \Gamma \vdash_\Xi x : \mathtt{Ch}\,A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^Z A \qquad \Gamma \vdash_\Xi P : A''}{\Gamma \vdash_\Xi x^z\Box y\Box.P : A''} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^Z A_1 \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^\frown A \qquad \Gamma \vdash_\Xi P : A_2}{\Gamma \vdash_\Xi \overline{x}^z\Box y\Box.P : A_1} \quad (x{:}A) \in A_1$$

(Subsumption)
$$\frac{\Gamma \vdash_\Xi P : A \qquad \Gamma \vdash_\Xi A' \qquad A \leq A'}{\Gamma \vdash_\Xi P : A'}$$

(Sub Interface)
$$\frac{A \subseteq A'}{A \leq A'}$$

We discuss the most important rules:

(Env Add _): It is possible to add names with their types to Γ, and also to Ξ (rule (Res)), provided that they are not already in Γ. Notice that $\mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Gamma)$ holds for well-formed environments.

(Type Interface): An interface type is well-formed if every name in it has been previously declared with the correct type and appears in Ξ (i.e., it is not bound by an input action). The premise $\Gamma \vdash_\Xi \diamond$ ensures the well formation of the type environment for the case of the empty interface.

(Res): Particular attention must be paid to restrictions of channel names, as channels may occur in the interface. A first idea could be to erase the newly restricted name from the interface as in the rule aside, but this rule is not sound with respect to the structural congruence relation: if you consider the processes $(\boldsymbol{\nu}\, y{:}\mathtt{Id}^Z[\,])y[(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})x^\uparrow()]$ and $(\boldsymbol{\nu}\, y{:}\mathtt{Id}^Z[\,])(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})y[x^\uparrow()]$ they are structurally equivalent, but while the former would be well-typed, the latter would not.

(*Wrong* Res Ch)
$$\frac{\Gamma, x{:}\mathtt{Ch}\ V \vdash_{\Xi,x} P : A}{\Gamma \vdash_\Xi (\boldsymbol{\nu}\, x{:}\mathtt{Ch}\ V)P : (A - [x{:}V])}$$

Therefore we rather use the *(Res)* rule that imposes that a restricted name can not appear in the interface of the process (see also comments right below

Property 1 Section 4.4). As all the names must be declared in Γ, it may seem that this condition forces all the interfaces to be empty. But note that this restriction applies only to process interfaces not to seal identifiers. The reader must avoid confusion between the name a which has type $\mathtt{Id}^Z A$ (where A may be a very complex interface) and the process $a[P]$ which, as stated by the rule (*Seal*), has type $[\,]$. What is necessary is that the type of P (rather than the one of $a[P]$) has interface A. That is, that the process P inside a seal $a[P]$ respects the interface declared for its name a. Therefore the side condition of (*Res*) simply demands that the upward channels of a are not restricted inside $a[P]$. In other words, a channel appearing in an interface must be already known to the enclosing environment. This is a very desirable feature of the type system: the interface's names must be somewhat public.

A brief example can clarify what "somewhat" means. Consider the following two terms in the light of the *(Res)* rule, and notice that they are structurally equivalent:

$$1)\quad y[(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})\ x^{\uparrow}()] \qquad\qquad 2)\quad (\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})\, y[x^{\uparrow}()]$$

Clearly, the first is not well-typed, since the process inside the seal should offer a restricted channel in the interface, and this is forbidden by the *(Res)* rule. Interestingly, the latter is not well-typed either: the type of the name y should include the channel x in the interface, but y is defined out of the scope of x; therefore process in the scope of the restriction could be typed only under a context in which x is declared twice, which is impossible (see Property 1(c) in Section 4.4). The correct term is $(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})(\boldsymbol{\nu}\, y{:}\mathtt{Id}\ [x{:}\mathtt{Shh}])\ y[x^{\uparrow}()]$ in which the channel x is declared *before* the seal y. Briefly, a name that is used by a seal to read from its environment must already exist in the environment where the seal is declared.

In terms of the examples in Section 3.1, this means that we can declare that a machine x has interface $[23 : \mathit{telnet}]$ only if the channel named 23 and the type *telnet* are both already known (that is, declared) in the environment.

(Input _): All the rules for typing a process of the form $\alpha.P$ follow a common pattern: this is especially true if we consider input and output rules separately.

The action $x^{\eta}(y{:}M).P$ binds y in P. Thus y must be added to the environment of P, provided that its type matches the type of x; y is not added to Ξ since it is bound by an input operation. Because we are doing an input, we also have to check that x is a $\boldsymbol{\nu}$-introduced name, that is $x \in \mathrm{dom}(\Xi)$. In *(Input Local)*, the input operation is local and nothing more has to be done. In *(Input Down)* we also check that the name of the seal from which the process wants to read is declared in Γ. In *(Input Up)* the input is from $\uparrow$, therefore the channel the process wants to read from must be added to the interface already deduced for P. This is done by the $\oplus$ operator, which computes the union of two interfaces, but is not defined when the result would contain two different pairs $y{:}M$ and $y{:}M'$ with the same name y but different M, M'.

(Output _): In the case of local and upward output actions the rules *(Output Local)* and *(Output Up)* check that the types of the channel and of the argument

match. The rule *(Output Down)* furthermore checks that the channel appears in the interface of the target seal with the right type. This enforces the interpretation of the interfaces: a process can write inside a seal only if the processes local to the seal are possibly going to read it.

(Rcv _): The typing rules for mobility actions do not differ from the respective communication actions. The main point is that in a receive operation the object name is not bound, so it is not added to the names in the scope of the continuation[4]. Remark that in order to send a seal on a channel, it must be declared to be mobile (attribute $\curvearrowright$). In the Seal's model of mobility, when a seal is received it gets a name chosen by the receiver process. We use this feature, together with the fact that the mobility attribute is tied to seals names, to turn a mobile seal into an immobile one. For instance, $(\boldsymbol{\nu}\, x{:}\mathtt{Ch}\ A)(\boldsymbol{\nu}\, a{:}\mathtt{Id}^{\curvearrowright}A)(\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\not\curvearrowright}A)\ \overline{x}^*\Box a\Box \mid x^*\Box b\Box \mid a[P] \ \rightarrow\ (\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\not\curvearrowright}A)\ b[P]$ turns the mobile seal named a into an immobile seal named b (the opposite is also possible). This is achieved by imposing no constraints on the mobility attribute of the receiving name in the receive typing rule. Neither this nor the opposite is possible in [4].

(Subtyping) During reductions, actions can be consumed. Consider for example the process $P = x^{\uparrow}(y{:}M).\overline{z}^*(y)$. It is ready to input a name of type M on channel x and its type is $[x{:}M]$. Now place it in the context $\mathcal{C}[-] = \overline{x}^a(w) \mid a[-]$ and consider the type of P and of its reductum:

$$\overline{x}^a(w).Q \mid a[\underbrace{x^{\uparrow}(y{:}M).\overline{z}^*(y)}_{[x:M]}] \ \rightarrow\ Q \mid a[\underbrace{\overline{z}^*(w)}_{[\,]}]$$

To satisfy the subject reduction property we introduce a subtyping relation. We already discussed that the interface of a process should be regarded as the set of channels on which the process *may* perform input operations from $\uparrow$. This suggests that the addition of new channels in the interface of a process should not be considered as an error, since they are channels on which interaction will never take place. This is formalized by the subtyping notion defined in the *(Sub Interface)* rule, that allows channels to be added to the interface of a process.

This possibility of extending the interface is limited to the process types, and is not extended to seal interfaces. The interface of a seal is associated with its name and is immutable, hence it characterizes forever the range of interactions admitted by that seal. At the same time, subsumption allows a process with a smaller interface to be placed inside the seal. This is essential, since the more limited interface may be a consequence, as in the previous example, of the "consumption" of some actions. In this way, actions can get consumed inside a seal, while the seal preserves its crystallized interface.

4.3 Typing Algorithm

The type rules in the previous section just need some slight modification to be converted into a type algorithm. As usual in type systems with subtyping, we

[4] This is due to the specificity of the receive action: when a seal is received it is activated at the same level as the process that received it. The movement actions look like interactions in the Fusion Calculus [14].

must eliminate the subsumption rule by embedding subtyping in the other rules. Actually there are only two rules that need modifications. The first is the *(Par)* rule: in order to type-check $P_1 \mid P_2$ in the environment Γ, Ξ both P_1 and P_2 are checked resulting respectively in the two types A_1 and A_2. If the process $P_1 \mid P_2$ can perform an input at $\uparrow$ then either P_1 or P_2 must be able to perform it, and so it has been registered in one of A_1 and A_2. Thus we have to merge the type informations kept in A_1 and A_2, and this is achieved by means of the $\oplus$ operator.

The second rule we need to modify is the *(Seal)* rule, to take into account that the interface of the process inside a seal may be a subtype of the interface associated with the seal name.

(Par Algo)
$$\frac{\Gamma \rhd_\Xi P_1 : A_1 \qquad \Gamma \rhd_\Xi P_2 : A_2}{\Gamma \rhd_\Xi P_1 \mid P_2 : A_1 \oplus A_2}$$

(Seal Algo)
$$\frac{\Gamma \rhd_\Xi x : \mathtt{Id}\ A \qquad \Gamma \rhd_\Xi P : A'}{\Gamma \rhd_\Xi x[P] : [\,]} \quad A' \leq A$$

4.4 Properties

The typing algorithm defined above is sound and complete with respect to the type system.

Theorem 1 (Soundness and completeness).
1. *If* $\Gamma \rhd_\Xi P : A$ *then* $\Gamma \vdash_\Xi P : A$.
2. *If* $\Gamma \vdash_\Xi P : A$ *then* $\exists A'$ *such that* $A' \leq A$ *and* $\Gamma \rhd_\Xi P : A'$.

A corollary of this theorem is the minimality of the algorithmic type:

Corollary 1. $\Gamma \rhd_\Xi P : \min\{A \mid \Gamma \vdash_\Xi P : A\}$, *if the set is not empty.*

In order to prove the subject reduction property we need a substitution lemma that states that substituting names for names of the same type in well-typed terms yields well-typed terms. This would fail if we allowed names that appear in interfaces to be substituted, hence we have to add a condition $x \notin \mathrm{dom}(\Xi)$ in the theorem hypothesis. This restriction is not a problem, since, as formalized by the management of Ξ in the *(Input _)* rules, interactions can only substitute names that do not appear in $\mathrm{dom}(\Xi)$.

Thanks to Theorem 1, the substitution lemma can be stated directly on the type algorithm rather than on the type system.

Lemma 1. *If* $\Gamma, x{:}M \rhd_\Xi P : A$, $x \notin dom(\Xi)$, *and* $\Gamma \rhd_\Xi y : M$, *then* $\Gamma \rhd_\Xi P\{^y/_x\} : A$.

This lemma is used to prove the subject reduction property for the algorithmic system whence subject reduction for the type system can be straightforwardly derived:

Theorem 2 (Subject Reduction). *If* $\Gamma \rhd_\Xi P : A$ *and* $P \rightarrow Q$ *then* $\Gamma \rhd_\Xi Q : A$.

Besides the characteristics discussed in Section 4.2 there several subtleties hidden in the type system that make subject reduction hold while keeping the rules

relatively simple. Among these it is noteworthy to remark that the provability of a judgment $\Gamma \vdash_\Xi \Im$ implies the following properties[5]:

Property 1. If $\Gamma \vdash_\Xi \Im$ si provable then:
a. Γ, Ξ are well formed (i.e., $\Gamma \vdash_\Xi \diamond$ is provable);
b. $\mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Gamma)$;
c. each variable has at most one type assignment in Γ.

These three properties allowed us to omit several sensible conditions from the typing rules since they are implicitly satisfied. So for example in the (Res) rule it is useless to require that $x \notin \mathrm{dom}(\Xi)$ since this already holds by the well-formation of the environment. Indeed $\Gamma, x{:}M \vdash_{\Xi,x} \diamond$ implies that $x \notin \mathrm{dom}(\Gamma, \Xi)$. Even more, $\Gamma, x{:}M \vdash_{\Xi,x} \diamond$ implies that x does not occur in Γ, since by construction Γ is an ordered list; this rules out envirements such as $y{:}\mathtt{Id}[x{:}M'], x{:}M$. Similarly, in all the rules (Input $\_$) it always holds that y does not occur in Γ. This implies that $y \notin \mathrm{dom}(\Xi)$ since otherwise $\Gamma \not\vdash_\Xi \diamond$ which contradicts that $\Gamma \vdash_\Xi x : \mathtt{Ch}\ M$ is provable.

5 Services vs. Effects

In the introduction we hinted at two possible interpretations of agent interfaces. Interfaces may describe either *services*, that is the interactions that *must eventually* occur, or *effects*, that is the interactions that *may possibly* occur.

The former interpretation is the one that characterizes the type systems for object-oriented languages, while the latter is the one of our system. Indeed, superficially our interfaces look like the types of the objects in the "objects as records" analogy: just an array of methods one can invoke (in fact, the analogy between agents and objects is not a piece of news). However, there is an important difference. In the object framework, sending a message should result in a method being activated: the type of an object reports the set of messages the object will answer to. We can say that the interface of an object characterizes the *services* that the object offers to its environment.

According to our definition, a channel that appears in the interface of an agent (a seal) does not guarantee that interaction on this channel is always going to happen (indeed the channel may be guarded or already be consumed by a previous action). A more precise intuition of our system is that an interface limits the *effects* that the agent can have on the environment: if an interaction occurs, it occurs on a channel defined in the interface and not on other channels.

There is a clear tension between the two interpretations and in this paper we opted for the second one. The reason for such a choice resides in the fact that π-calculus channels are essentially consumable resources. One of the clearest lessons we draw from this work is that there is an inherent difference between requiring a service (such as sending a message) and writing on a channel: the former does not affect the set of admissible interactions, while the latter does (by consuming a channel).

[5] The first property follows by straightforward induction whose base are the rules (Type `Shh`), (Var), and (Dead). The other two are equally straightforward.

This tension is manifest at the level of subtyping: in case of effects the "may-provide" interpretation is embodied by a subtyping relation typical of *variant types* while in the case of services, we recover the classical record types relation that characterizes objects and their "must-provide" interpretation, as expressed by the rules on the side.

$$\text{(effects)}\quad \frac{A \subseteq A'}{A \leq A'} \qquad \text{(services)}\quad \frac{A' \subseteq A}{A \leq A'}$$

Our analysis clearly shows that the two approaches are mutually exclusive, and that either one or the other has to be adopted according to the "consumability" of the communication layer.

In our system it is possible to recover the object/services characteristics by imposing restrictions to ensure *receptiveness* [16] of channels in the interface[6], which roughly corresponds to make all the external interactions of an agent unconsumable. The intuition is that in this way we transform interface channels into (object) methods. Receptiveness can be ensured by imposing restrictions such as those presented in [1] or, in a coarser approach, by requiring that all receive and input actions on upper channels are guarded by replications, that is they must all be of the form $!x^{\uparrow}(y).P$ and $!x^{\uparrow}\square y\square.P$. In the latter case some simple modifications to our type system allow us to recover the service interpretation together with its (services) subtyping rule. It just suffices to straightforwardly modify the typing rules (Input Up) and (Rcv Up) to account for the new syntax, and the results of the previous section bring forth. However we decided to follow the other approach since the presence of concurrency does not ensure that services will be eventually satisfied. Indeed, even if the remote interactions are replicated they may still be guarded. Therefore a complete treatment would require further restrictions on these interactions bringing us too far from our original subject. Nevertheless we believe that such a direction is worth exploring.

6 Example: A Web Crawler

In this section we give a simple example that uses mobility, higher-order types, and parametric write channels. Chapter 5 of [21] contains a much more complex example we did not include here for lack of space: in that example the toy distributed language introduced in [4] to show the expressivity of typed ambients is encoded in the Seal calculus version presented here.

In order to show a possible application of higher order typing and mobility attributes, we suggest the specification of a possible *web crawling* protocol. Currently, most commercial web search engines periodically access all the web pages that are reachable by some starting pages and index them in a database. Web searches access the database and retrieve relevant entries.

This technique is a greed bandwidth consumer. It may be interesting to define an alternative protocol where mobile agents are spawned over the web sites, where they collect and pre-elaborate the relevant information, so that the

[6] An alternative solution is to use the object framework but to give up with the "must provide" interpretation, as it is done in [8].

computational effort is distributed, and bandwidth consumption is dramatically reduced.

The Seal specification of this protocol is depicted in Figure 2, where top level represents the network and hosts are immobile seals that lie inside it; crawlers are modeled by mobile seals, being able to navigate among hosts.

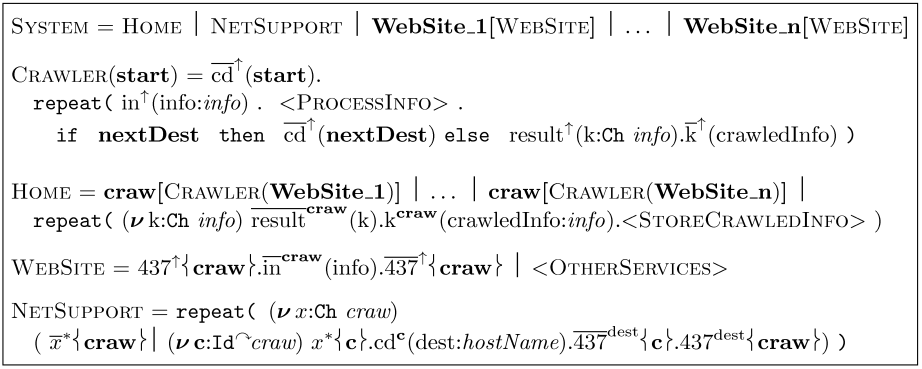


Fig. 2. A web crawler protocol

Home, which is a process that lives at the network level, spawns a crawler for each root web site. The crawler will go away and come back, to tell Home about its findings, as we will see later.

Crawler communicates on channel "cd" ("crawler destination") the name of the first site it wants to visit[7]. This information is received by NetSupport which first renames the crawler with a fresh name c; this renaming is performed by sending the crawler along the local channel x. Then, NetSupport sends the crawler to the requested destination, via the port 437. Once the crawler is in the site, it reads the information via the port "in", and is sent out of WebSite along channel 437. The crawler processes the information (which generates a list of other possible destinations), then checks whether it has to visit more sites; if it does not, it uses the channel "result" to ask Home for a secure channel k and sends the result on it.

Home sends the secure channel name k along the result$^{\mathbf{craw}}$ channel, reads the collected information from k, and stores it.

A generic WebSite must have a dæmon that is ready to receive crawlers on port 437 and, after having provided them with information on channel "in", sends them out via the port 473 again.

The interface that characterizes a crawler is *craw* = [in: *info* ; result: `Ch` (*info*)]. All the crawlers in the toplevel have the name **craw**, of type `Id`⌢[in: *info* ; result: `Ch` (*info*)]. The other relevant interface in the example is the one of the hosts: it is a higher-order type, since it contains the interface *craw*, i.e. it specifies the protocol of the agents it is willing to accept. This interface

[7] `repeat` is syntactic sugar for ! and `if_then_else` can be easily encoded. We use italics for types, roman font for channels, small capitals for metavariables, and boldface font for seal names.

has the following form: $[437 : \mathit{craw} ; < \textsc{OtherPorts} >]$. Since hosts are immobile, they are denoted by names whose type is $\mathit{hostName} = \mathtt{Id}^{\vee}[437 : \mathit{craw} ; < \textsc{OtherPorts} >]$.

7 Practical Applications

In order to show the potential of our type system we hint at how it can be used to provide the JavaSeal Agent Kernel [2,18] with a minimal type system. JavaSeal is an experimental platform that provides several abstractions for constructing mobile agent systems in Java. JavaSeal uses relatively coarse grained types; in particular, objects exchanged during interaction are encapsulated in `Capsules`. Capsules are the only entities liable to be sent over channels. The contents of a capsule are widening to the generic type `Object` thus loosing all static information. Furthermore, the system does not distinguish between channel and seal identifiers as both are denoted by objects of the class `Name`. In other words, JavaSeal does little type checking and what it does is mostly performed at run time through dynamic type casts. This means that JavaSeal agents are prone to errors.

In particular, each object exchanged during interaction is encapsulated with type `Object` into a `Capsule`, being capsules the only entities liable to be sent over channels. Also there is not a clear distinction between channels and seal identifiers since they are generically classified by the class `Name`. In other words JavaSeal type checking is rather weak since it heavily relies on the use of dynamic type casts, and as such it is quite prone to errors.

JavaSeal is based on the primitives of the original Seal calculus. Therefore it does not provide shared channels: channels are localized and access to them is granted via portals opening operations. More precisely this signifies that for example a downward output operation on channel x^y synchronizes only with local input operation on x in the seal y, and that the interaction needs presence in y of an explicit permission $\mathsf{open}_{\uparrow} x$ that authorizes the parent to use the local channel x. That is the (write in) becomes the following three parties reduction rule:

$$\overline{x}^y(v).P \mid y[x^*(u).Q \mid \mathsf{open}_{\uparrow} x \mid R] \quad \rightarrow \quad P \mid y[Q\{^v/_u\} \mid R] \qquad \text{(write in)}$$

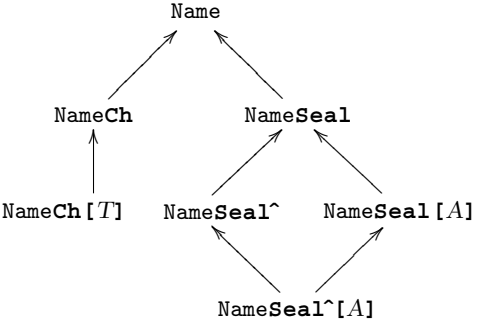
It is quite straightforward to adapt our interfaces types to located channels and portals: recall that interfaces trace all the channels on which there is an information flow from the parent to the child. Therefore the interface of a (Java)Seal agent must contain all channels the agent may perform input on and that (a) either are located in the parent (b) or are local and have a matching upward portal open operations.

Our proposal is then to endow the actual JavaSeal syntax with some type informations that will be processed by a preprocess to type-check the source, and then will be erased yielding a standard JavaSeal program. In order to enhance readability we write the information that are to be erased by the preprocessor in boldface. More particularly we propose to add the following (preprocessor) types:

`NameCh[`T`]` it is used to classify channel *names* (it corresponds to `Ch` T). The type part `[`T`]` is optional (its absence means that the content of the channel does not need to be checked)

`NameSeal^[`A`]` it is used to classify seal names (it corresponds to $\mathtt{Id}^{\vee}A$). Both the immobility attribute `^` and the interface part `[`A`]` are optional (the absence of `^` corresponding to $\curvearrowright$, and the one of the interface meaning that outputs towards the seal do not need to be checked).

In order to have backward compatibility and let the programmer decide how fine-grained the preprocessor analysis should be, we order the newly introduced types according to the subtyping relation described by the diagram below.



Thus the `Name` type that in the current JavaSeal implementation identifies all names, will be refined by separating channel names from agent names. Agent names will allow a second refinement by specifying their interfaces or its mobility attribute. A similar specialization is possible by specifying or not the content of the channel.

The idea is that the programmer is free to decide whether the preprocessor has just to check that, say, the name used to denote a channel is indeed a channel name, or also match the type of its content. Similarly the programmer may force the check of downward write operations, or just require that they are directed to some named seal. The more the leaves of the hierarchy are used the more the check will be complete.

Thus the `Name` type that in the current JavaSeal implementation identifies all names, will be refined by separating channel names from agent names. Agent names will allow a second refinement by specifying their interfaces or its mobility attribute. A similar specialization is possible by specifying or not the content of the channel.

The idea is that the programmer is free to decide whether the preprocessor has just to check that, say, the name used to denote a channel is indeed a channel name, or also match the type of its content. Similarly the programmer may force the check of downward write operations, or just require that they are directed to some named seal. The more the leaves of the hierarchy are used the more the check will be complete.

This system is particularly interesting when it is used in conjunction with parametric classes such as they are defined for example in Pizza [13]. So the `Capsule` and `Channel` classes of JavaSeal could be rewritten as follows

```
final class Capsule<X> implements Serializable {
  Capsule(X obj);
  final X open();
}
```

```
final class Channel<X> {
  static void send(NameCh[X] chan, NameSeal seal, Capsule<X> caps);
  static Capsule<X> receive(NameCh[X] chan, NameSeal seal);
}
```

It is interesting to notice that after preprocessing, by applying the Pizza homogeneous translation of [13] to all non-erased occurrences of the type variable, one recovers the original interface of JavaSeal:

```
final class Capsule implements Serializable {
  Capsule(Object obj);
  final Object open();
}

final class Channel {
  static void send(Name chan, Name seal, Capsule caps);
  static Capsule receive(Name chan, Name seal);
}
```

8 Conclusion

In this work we presented a new definition of the Seal Calculus that gets rid of existing inessential aspects while preserving the distinctive features of the Seal model. We used it to tackle the problem of typing not only mobile agents but also their movement, so that the latter can be controlled and regulated. The solution we used, typed channels, is an old one —it is standard in π-calculus— but its use for typing mobility is new, and results into a higher order type systems for agents (as [20] is a higher order type system for processes). At the same time, we designed our type system so that it induces an interpretation of interfaces as effects that differs from the customary interpretation as services and we discussed its distinctive features.

This work is just a starting point and for some aspects it is still unsatisfactory. For example more work is needed to define a type system that captures one of the peculiar security characteristics of the Seal Calculus, that is the name-spaces separation: in the actual version if two agents residing in different locations have the same name, then the type system forces them to have the same type too.

At the same time this work already constitutes an exciting platform whence further investigation can be started. In particular we are planning to use some form of grouping similar to those in [5,3] for a different solution to the problem of type dependencies, as well as to investigate a distributed version of the type system, on the lines of [3]. It would also be interesting to verify what the *single threaded types*, introduced in [9] for the Ambient Calculus with co-actions, would bring forth in the Seal Calculus, where co-actions are inherently present.

Acknowledgments. The authors want to thank Mariangiola Dezani, Jan Vitek, and the anonymous referees for useful comments on this paper. This work was partially supported by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet".

References

1. R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed π-calculus. In *FST&TCS*, number 1738 in Lecture Notes in Computer Science, pages 304–315, 1999.
2. C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 2002. To appear.
3. M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, pages 222–235, London, 2001. ACM Press.
4. L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.
5. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science, pages 333–347. Springer, August 2000.
6. L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98*. ACM Press, 1998.
7. L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999.
8. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 2000. To appear.
9. F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.
10. M. Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, École de Mines de Paris, October 2000.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.
12. Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.
13. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *24th Ann. ACM Symp. on Principles of Programming Languages*, 1997.
14. J. Parrow and B. Victor. The Fusion Calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*. IEEE Computer Society Press, 1998.
15. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
16. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999.
17. P. Sewell and J. Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop*, 1999.
18. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal: or how to make Java safe for agents. In Dennis Tsichritzis, editor, *Electronic Commerce Objects*. University of Geneva, 1998.
19. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science. Springer, 1999.
20. N. Yoshida and M. Hennessy. Assigning types to processes. In *Proceedings, Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–348, 2000.
21. F. Zappa Nardelli. Types for Seal Calculus. Master's thesis, Università degli Studi di Pisa, October 2000. Available at `ftp://ftp.di.ens.fr/pub/users/zappa/readings/mt.ps.gz`.

Reasoning about Security in Mobile Ambients*

Michele Bugliesi[1], Giuseppe Castagna[2], and Silvia Crafa[1,2]

[1] Dipartimento di Informatica
Univ. "Ca' Foscari", Venezia, Italy
michele@dsi.unive.it
[2] Département d'Informatique
École Normale Supérieure, Paris, France

Abstract. The paper gives an assessment of security for *Mobile Ambients*, with specific focus on *mandatory access control* (MAC) policies in multilevel security systems. The first part of the paper reports on different formalization attempts for MAC policies in the Ambient Calculus, and provides an in-depth analysis of the problems one encounters. As it turns out, MAC security does not appear to have fully convincing interpretations in the calculus. The second part proposes a solution to this *impasse*, based on a variant of Mobile Ambients. A type system for resource access control is defined, and the new calculus is discussed and illustrated with several examples of resource management policies.

1 Introduction

Distributed computation based on mobile code is already ubiquitous and represents an essential aspect of our computing environments. Mobile computing relies on sharing of data and software resources among computing sites distributed across wide-area open networks. This sharing is successful inasmuch as it satisfies several criteria, including safety, e.g. execution of mobile code without failure, and security, e.g. protection of sites against malicious intruders and misuse of their computing resources.

A substantial body of the research on programming languages has recently been directed towards the study of formal calculi providing high-level support for mobile agents. A non exhaustive list of examples includes the Ambient Calculus [CG98], the Seal Calculus [VC99,CGZ01], the Dπ-calculus [HR00b], and the Join Calculus [FGL$^+$96].

The initial motivation for this paper was an assessment of security in calculi for mobility. As a preliminary step, we thought it instructive to study what (if any) new insight and challenges mobile code languages provide for well-established security models. For some of the calculi we just mentioned, notably for the Dπ-calculus, an in-depth study of these aspects has already been conducted in [HR00b]. Here we present a corresponding analysis for Mobile Ambients, for which, to our knowledge, no previous attempt in this direction has been made.

* Work partially supported by MURST Project 9901403824_003, by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet", and by Galileo Action n. 02841UD

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 102–120, 2001.
© Springer-Verlag Berlin Heidelberg 2001

The focus of our analysis is on *mandatory access control* policies (MAC) in multilevel security systems. In particular, the emphasis is on the specific aspects of MAC policies related to confidentiality and integrity, and their different implementations as *military* security (no read-up, no write-down) and *commercial* security (no read-up, no write-up).

The first part of the paper (§ 2) is a survey of our formalization attempts. As it turns out, the main problem comes far ahead the point where one starts the formalization, because the security concepts assumed as references do not appear to have any fully convincing interpretation in the calculus. In fact, the very meaning of basis notions such as "read access" and "write access" by subjects on objects, or even "ownership", is somehow difficult to grasp and characterize when looked at from within the Ambient Calculus. As a consequence of these difficulties, one is led to the conclusion that Ambients lack adequate primitives to capture and characterize those security concepts. While our arguments are only informal, the analysis we detail in the first part of the paper does provide convincing evidence in favor our conclusion.

The second part of the paper proposes a solution to this *impasse*, based on a variant of Mobile Ambients we dub *Boxed Ambients*. The calculus of Boxed Ambients is introduced, formally defined and studied in a companion paper [BCC01]. Here, instead, we keep the presentation largely informal, and put the emphasis on the role of the new calculus for describing and expressing resource access control policies. After a brief description of the calculus, we introduce a type system for resource access control (§ 3). Then (§ 4) we propose several examples that illustrate what we believe to be the merits and strengths of the calculus. A final section (§ 5) is dedicated to related work and conclusions.

While the second part of the paper represents the main contribution of the paper, the preliminary analysis was extremely useful to us to understand the problems, and we hope will be equally valuable to the reader.

2 Mobile Ambients and Multilevel Security

Standard models of security for resource access control are built around *subjects* performing access requests on *objects* by *write* (in some models, also *append*, *execute*, and others) and *read* operations.

Multilevel security presupposes a lattice of security levels, and every subject and object is assigned a level in this lattice. Based on these levels, access to objects by subjects are classified as *read-up* (resp. *read-down*) when a subject access by a read an object of higher (resp. lower) level, and similarly for write accesses. Relying on this classification, one may distinguish two security policies: *military* security, which forbids (both direct and indirect) read-up's and write-down's, and *commercial* security that forbids read-up's and write-up's. These notions cover also *indirect* accesses resulting from the composition of atomic operations: thus also the fact of writing into an object of any level a piece of information read (or just coming) from another object whose level is higher than

the level of the first object is considered as a write-down (classic security handles these cases by the so-called $\star$-property [BP76,Gol99][1]).

2.1 Mobile Ambients

Ambients are processes of the form $a[\,P\,]$ where a is a name and P a process. Processes can be composed in parallel, as in $P \mid Q$, exercise a capability, as in $M.P$, declare local names as in $(\nu x)P$, they can be replicated, as in $!P$, or simply do nothing as in $\mathbf{0}$.

Mobility. Ambients may be nested to form a tree structure that can be dynamically reconfigured as a result of mobility and ambient dissolution determined by the capabilities $\mathsf{in}, \mathsf{out}$ and open. To exemplify, consider the ambients a and b in the configuration $a[\,\mathsf{open}\ b.\mathsf{in}\ c\,] \mid b[\,\mathsf{in}\ a.\mathsf{in}\ d\,]$. The ambient b may enter a, by exercising the *capability* $\mathsf{in}\ a$, and reduce to $a[\,\mathsf{open}\ b.\mathsf{in}\ c \mid b[\,\mathsf{in}\ d\,]\,]$. Then a may dissolve b by exercising $\mathsf{open}\ b$, and reduce to $a[\,\mathsf{in}\ c \mid \mathsf{in}\ d\,]$.

Security. The ability or inability to cross boundaries, which is conferred by the capabilities in and out, is also at the core of the security model underlying Mobile Ambients. Permission to cross ambient boundaries is given by making the name available to the clients willing to enter or exit. Names are thus viewed as passwords, or alternatively as cryptokeys: when embedded in a capability, an ambient name provides the pass that enables access to, or else the cryptokey that discloses the contents of that ambient.

While this model of security is suggestive, and powerful for its simplicity, it appears to not be fully adequate for modeling realistic policies for resource access control. The problem is that it entirely depends on the ability by the authorization mechanism to filter out undesired clients: an authorization breach could grant malicious agents full access to all the resources located inside the ambient boundary. Clearly, one first has to identify what "resource access" is in the Ambient Calculus. Entering an ambient, or opening it are all good notions of access: in addition, there is of course communication.

Communication. In the Ambient Calculus, communication is anonymous, and happens inside ambients. The configuration $(x)P \mid \langle M \rangle$ represents the parallel composition of two processes, the output process $\langle M \rangle$ "dropping" the message M, and the input process $(x)P$ reading the message M and continuing as $P\{x := M\}$. The open capability has a fundamental interplay with this form of communication: opening an ambient enables synchronization between the processes located in the opening and the opened ambients. To exemplify, synchronization between the input process $(x)P$ and the output $\langle M \rangle$ in the system $(x)P \mid \mathsf{open}\ b \mid b[\,\langle M \rangle \mid Q\,]$ is enabled by exercising the capability $\mathsf{open}\ b$ to unleash the message $\langle M \rangle$.

It is the interplay between communication and the primitives for ambient mobility which makes it difficult to reason about resource access in terms of classical security models. To make our point, we use a simple concrete example.

[1] As a matter of fact, these references do not define precisely what a *write-down access* is; instead, they give a definition of *no-write down policy*.

2.2 A Simple Resource Access Problem

Suppose we have a system consisting of a set of resources $\{r_1, \ldots, r_n\}$ and an agent named a that runs program P and is willing to access any of the r_i's. To control the access requests by the agent, one would typically refer to [DoD85] and set up a resource manager. In the Ambient Calculus the system under consideration can be represented as follows:

$$a[\,P\,] \mid m[\,r_1[\,\cdots\,] \mid \cdots \mid r_n[\,\cdots\,] \mid R\,]$$

Here, m is the resource manager running process R. To access, say r_i, the agent needs to know the name m, to be able to move inside the resource manager. Assuming the agent knows that name, the result of the move is the new system:

$$m[\,a[\,P\,] \mid r_1[\,\cdots\,] \mid \cdots \mid r_n[\,\cdots\,] \mid R\,]$$

Looking at this configuration, it is clear that the process R does not have an active role in the system: given the primitive constructs of the Ambient Calculus, there is indeed nothing R can do to enable or control the access, as the interaction between $a[\,P\,]$ and each of the r_i's may only result from autonomous actions by either the agent or the resource[2]. The role of the ambient m is therefore reduced to the role of its name: it is simply the first password required for the access. Rather, it is each of the r_i's that needs to include its own manager.

We can thus formulate the problem in simpler terms, and look directly at the case of the agent a and the resource r shown below:

$$\textit{Initial configuration:}\quad a[\,P\,] \mid r[\,R \mid \langle M \rangle\,]$$

R is the manager for r, and M is the contents: for the purpose of the example, we assume that the content is a value the agent is willing to read.

2.3 Overview of Possible Solutions

Having defined the problem, we now look at different ways to attack it in the Ambient Calculus, and discuss their implications for MAC security.

2.3.1. Agent Dissolution. A first solution is based on the following protocol proposed by [CG98]. In order for a to access r, a first enters r:

$$\textit{Enter:}\quad r[\,R \mid \langle M \rangle \mid a[\,P\,]\,]$$

Now, the idea of the protocol is that the manager R should be the process $!\mathsf{open}\ p$, which unleashes authorized clients that entered the resource within a transport ambient named p. In other words, the protocol requires the client to know the name of the resource, as well the name of the "port" p used for access. The agent would first rename itself to p to comply with the rules of the protocol, and then enter: if the access to r is in read mode, the agent will contain a reading process. After renaming, the new configuration would then be:

$$\textit{Renaming:}\quad r[\,!\,\mathsf{open}\ p \mid \langle M \rangle \mid p[\,(x)P\,]\,]$$

Finally, the resource manager enables the read access, by opening p:

$$\textit{Read Access:}\quad r[\,!\,\mathsf{open}\ p \mid \langle M \rangle \mid p[\,(x)P\,]\,] \rightarrow r[\,!\,\mathsf{open}\ p \mid \langle M \rangle \mid (x)P\,]$$

[2] *Safe Ambients* [LS00] would not help here, as R would still be unable to mediate the access to r_i.

The protocol is elegant and robust, as the agent needs to know two passwords (r and p). There are, however, a number of unsatisfactory aspects to it.

A first reason for being unsatisfied with the protocol is that it is hardly realistic to assume that agents willing to read a value should be prepared to be dissolved. A second problem is that opening $p[\,P\,]$ may be upsetting to the resource manager, or else to the resource itself, because there is no telling what P might do once unleashed. For what we know, the contents of p could very well be the process $N.P$, with N a path of in or out capabilities. Unleashing this process inside r could result into r being carried away to possibly hostile locations, or otherwise being made unavailable to other clients requesting access to it.

Further problems arise when we try to classify the protocol according to the principles of MAC security. As we noted, the action in the protocol that eventually enables the access to the resource is taken by the resource manager, which opens the incoming agent. In other words, it is the last step of the protocol that effectively determines the access, and since the process enclosed in p is an input process, it is classified as a read access (had p contained an output, it would have been a write access). In multilevel security, it would then be possible to further classify the access according to the security levels associated with r and p, and use that definition to enforce either the military or the commercial security policies.

However, while this form of classification is sensible for the protocol, it becomes rather artificial when applied to the primitives of the calculus. Indeed, saying that $\mathsf{open}\ p \mid p[\,P\,]$ is a read (or write) access from P is rather counter-intuitive, as $p[\,P\,]$ undergoes the action rather than actively participating into it. The problem is that the protocol is tightly dependent on the effects of open, but when exercised to enable a read/write request, open exchanges the roles of the two participants in the request, as it is the subject, rather than the object, that is accessed, in fact, opened.

2.3.2. Resource Dissolution. The problem could be circumvented by a change of perspective. One could devise a different protocol where the active role of the subject is rendered by a combination of open and input/output. Thus, for instance, the process $\mathsf{open}\ r.(x)P$ could be interpreted, in the protocol, as a read request on r. This might work reasonably for read requests, even though the interpretation is not too convincing given that the access has also the side-effect of dissolving the resource. Even less convincing would be the interpretation of $\mathsf{open}\ r.\langle M\rangle$ as a write access: after dissolving r the output $\langle M\rangle$ really has nothing to do with a write on r.

2.3.3. Agents and Messengers. To avoid indiscriminate dissolution upon access, [CG98] suggests a different approach, based on a protocol similar to the first one we discussed, but in which agents rely on "special" ambients acting as messengers. The idea is to envisage two classes of messengers:

output messenger: $o[\,M.\langle N\rangle\,]$. M is a path to the location where deliver message N;

input messengers: $i[\, M.(x)o[\, M^{-1}.\langle x\rangle \,]\,]$. M is the path to the location where a value can be read. Once read, the messenger goes back to its original location where it delivers the value.

Thus, a read access would be encoded by a protocol based on the following initial configuration:

$$a[\,\mathsf{open}\ o.(x)P \mid i[\,\mathsf{out}\ a.\mathsf{in}\ r.(x)o[\,\mathsf{out}\ r.\mathsf{in}\ a.\langle x\rangle\,]\,]\,] \mid r[\,!\mathsf{open}\ i \mid \langle N\rangle\,]$$

The protocol still requires cooperation from the resource manager, which is expected to open the input messenger. Also, looking at the primitive reductions, it would still be counter-intuitive to say that $\mathsf{open}\ i \mid i[\,P\,]$ is a read access. However, if i could be identified as an input-messenger within r, then the access classification would be more realistic.

The problem is that there is no way to syntactically tell messengers from ambients playing the role of "pure" agents, nor is there any way to syntactically detect "illegal" attempts to dissolve "pure" agents. Defining a notion of access, and attempting a syntactic classification would therefore still be problematic, if at all possible.

Types could be appealed to for more satisfactory solution. One could devise a type system to complement the syntax by enforcing a typed partition of ambients into agents (i.e. ambients that cannot be dissolved) and messengers (as above). Based on the typed ambient classification and on an assignment of security levels, it would then be possible to classify access requests according to MAC policies. There would be only one remaining problem. Consider the protocol structure and evolution. From the initial configuration: $a[\,P' \mid i[\,M.(x)o[\,M^{-1}.\langle x\rangle\,]\,]\,] \mid r[\,!\mathsf{open}\ i \mid \langle N\rangle\,]$ a sequence of reductions routes the input messenger to its, where it it is opened and consumes N. At this stage, the structure of the system is: $a[\,P'\,] \mid r[\,!\mathsf{open}\ i \mid o[\,M^{-1}.\langle N\rangle\,]\,]$. This is the encoding of a write access by r to a. In other words, a read access by a includes a write access by r: if the former is, say, a read-up, then the latter is a write-down. In other words, the protocol has somehow the effect of merging read-up's and write-down's, and dually, write-up's and read-down's. Therefore, military security could still be accounted for with this approach, while commercial security could not.

2.4 Summary and Assessment

The survey of solutions we have given may still be incomplete, but we do not see any significantly different approach to attack the problem. As to the approaches we have presented, none of them is fully adequate to reason about security. Some of them appear artificial, since essential intuition is lost in the encoding of the protocol (§ 2.3.1,§ 2.3.2), while in others, intuition is partially recovered but only at the expenses of failing to provide full account for both military and commercial security (§ 2.3.3).

Consequently, while possibly incomplete, the analysis does provide a basis for drawing a conclusion. Certainly, the Ambient Calculus *enables* resource access control, in that it provides constructs for encoding access protocols. On the other

hand, the calculus *does not, by itself, support* these mechanisms and policies, as it does not provide built-in facilities to make it convenient or natural to reason about them. As we showed, the reasoning is possible at the level of access *protocols*, while when we look at the access *primitives*, there appears to be no general principle to which one can steadily appeal.

The conclusion we may draw, then, is that *support* for resource access control with Mobile Ambients requires different, finer-grained, constructs for ambient interaction and communication. The new constructs should be designed carefully, so as to complement the existing restrictions on ambient mobility based on authorization, without breaking them. In other words, access to remote resources should still require mobility, hence authorization: local access, instead, could be made primitive.

To see how that can be accomplished, consider once more the protocol of § 2.3.3, based on messengers. We can re-state it equivalently as follows:

$$a[\,\mathsf{in}\ r.\mathsf{open}\ o.(x)\mathsf{out}\ r.P \mid i[\,\mathsf{out}\ a.(x)o[\,\mathsf{in}\ a.\langle x\rangle\,]\,]\,] \mid r[\,!\,\mathsf{open}\ i \mid \langle M\rangle\,]$$

In other words, it is now the agent that is responsible for the moves needed to reach the resource, while the messenger just makes the in and out moves needed for the, now local, access. After the move of a into r, and of i out of a, the structure of the system (disregarding a) is the following: $r[\,\mathsf{open}\ i \mid \langle M\rangle \mid i[\,(x)P\,]\,]$. This is where the read access takes place. Now, instead of coding it, via open, we can make it primitive and do without open. If we denote with $(x)^{\uparrow}$ input from the enclosing ambient, the read access is simply: $r[\,\langle M\rangle \mid i[\,(x)^{\uparrow}P\,]\,]$. But then, the whole protocol can be simplified: $a[\,\mathsf{in}\ r.(x)^{\uparrow}.P\,] \mid r[\,\langle M\rangle\,]$.

A choice of communication primitives based on this observation led us to the design of *Boxed Ambients*, a calculus we formally define in [BCC01] and outline in the next section. The new primitives provide the calculus with what we believe to be more effective constructs for resource protection and access control, while at the same time retaining the expressive power and most of the computational flavor of Mobile Ambients, as well as the elegance of their formal presentation.

3 Boxed Ambients

Boxed Ambients are a variant of Cardelli and Gordon's Mobile Ambients. From the latter, they inherit the primitives in and out for mobility, with the exact same semantics. Instead, Boxed Ambients rely on a completely different model of communication, which results from dropping the open capability.

As in the Ambient Calculus, processes in the new calculus communicate via anonymous channels, inside ambients. In addition, to compensate for the absence of open, Boxed Ambients are equipped with primitives for communication across ambient boundaries, between parent and children. Syntactically, this is obtained by means of tags specifying the *location* where the communication has to take place. So for example, in $(x)^n P$ the input prefix $(x)^n$ is an input from child ambient n, while $\langle M\rangle^{\uparrow}$ is an output to the parent ambient.

The choice of these primitives is inspired to Castagna and Vitek's *Seal Calculus* [VC99], from which Boxed Ambients also inherit the two principles of

mediation and *locality*. Mediation implies that remote communication, e.g. between sibling ambients, is not directly possible: it either requires mobility, or intervention by the ambients' parent. Locality means that communication resources are *local* to ambients, and message exchanges result from explicit read and write requests on those resources.

As it turns out, the resulting communication model has rather interesting payoffs when it comes to resource protection policies and security. Before entering further details, we briefly review the syntax and the semantics of the calculus.

Syntax and Semantics. The untyped syntax of the polyadic synchronous calculus is as follows.

$$\begin{array}{lrl}
\textit{Expressions} & M & ::= a, b, \ldots \mid x, y, \ldots \mid \mathsf{in}\ M \mid \mathsf{out}\ M \mid M.M \mid (M_1, \ldots, M_k) \\
\textit{Patterns} & \boldsymbol{x} & ::= x \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_k \\
\textit{Locations} & \eta & ::= M \mid \uparrow \mid \star \\
\textit{Processes} & P & ::= \mathbf{0} \mid M.P \mid (\boldsymbol{\nu} x)P \mid \ P \mid P \ \mid M[\,P\,] \mid !P \mid (\boldsymbol{x})^{\eta}P \mid \langle M \rangle^{\eta} P
\end{array}$$

We use a number of notation conventions. We reserve $a - q$ for ambient names, and x, y, z for variables. As usual we omit trailing dead processes, writing M for $M.\mathbf{0}$. The superscript $\star$ denoting local communication, is also omitted.

The operational semantics is defined by reduction, with the help of an auxiliary relation of structural congruence. All these are very standard (in fact, exactly as in Ambient Calculus). We only give the top-level reduction rules, and refer the reader to [BCC01] for details.

Mobility. Reduction for the in and out capabilities is exactly as for Mobile Ambients:

$$\begin{array}{ll}
(\textit{enter}) & a[\,\mathsf{in}\ b.P \mid Q\,] \mid b[\,R\,] \rightarrow b[\,a[\,P \mid Q\,] \mid R\,] \\
(\textit{exit}) & a[\,b[\,\mathsf{out}\ a.P \mid Q\,] \mid R\,] \rightarrow b[\,P \mid Q\,] \mid a[\,R\,]
\end{array}$$

Communication. The primitives for local and parent-child communication are governed by the following rules. Note that in all cases input-output is synchronous (see § 4 for a brief digression on asynchronous communication).

$$\begin{array}{ll}
(\textit{local}) & (x)P \mid \langle M \rangle Q \rightarrow P\{x := M\} \mid Q \\
(\textit{input}\ n) & (x)^n P \mid n[\,\langle M \rangle Q \mid R\,] \rightarrow P\{x := M\} \mid n[\,Q \mid R\,] \\
(\textit{input}\uparrow) & \langle M \rangle P \mid n[\,(x)^{\uparrow} Q \mid R\,] \rightarrow P \mid n[\,Q\{x := M\} \mid R\,] \\
(\textit{output}\ n) & \langle M \rangle^n P \mid n[\,(x)Q \mid R\,] \rightarrow P \mid n[\,Q\{x := M\} \mid R\,] \\
(\textit{output}\uparrow) & (x)P \mid n[\,\langle M \rangle^{\uparrow} Q \mid R\,] \rightarrow P\{x := M\} \mid n[\,Q \mid R\,]
\end{array}$$

3.1 Resources and Access Control

Four different reductions for non-local exchange may be thought of as redundant, especially because there are only two reducts. Instead, different directions for input/output is a key design choice that has a number of interesting consequences.

– First, the primitives for communication have immediate and very natural interpretations as access requests. To exemplify, the input prefix $(x)^n$ can

be seen as a request to read from the channel located into child ambient n. In fact, given the anonymous nature of channels, $(x)^n$ can equivalently be seen as an access to the ambient n. Dually, $\langle M\rangle^{\uparrow}$ can be interpreted as write request to the parent ambient (equivalently, its local channel)[3].

- Secondly, full and flexible support is now available for resource protection. An agent entering a resource needs not be opened there to enable the access: the resource manager can mediate and keep full control over the read and write requests made by the agent. If we take the resource access problem of § 2.2 we now have a fairly natural and elegant solution, and we also find back a role for the resource manager m. Consider again the configuration $m[\, a[\, P\,] \mid r_1[\cdots] \mid \cdots \mid r_n[\cdots] \mid R\,]$ where now all ambients are boxed, and a has entered the resource manager. We need not to include a manager in each resource, as R may act as a mediator. For instance, R could be defined as the parallel composition $R_1 \mid \cdots \mid R_n$ where each R_i is the process $!\,(x)\langle x\rangle^{r_i}$ waiting for upward output from a and forwarding it to the ith resource. Some of the R_i's could be less generous with the agent, and ignore upward input from a to request read access on a instead: $!\,(x)^a\langle x\rangle^{r_i}$. Should any of the r_i's be made non-accessible, one would simply define $R_i = \mathbf{0}$.
- The communication model fits nicely the security model of Mobile Ambients which is based on authorization and predicates in/out access to ambients on possession of appropriate passwords or cryptokeys.
- Finally, multilevel security for boxed ambients may be modeled by embedding security levels in types, and using typing rules to enforce and verify *Mandatory* (system-wide) *Access Control* (MAC) policies. We give a detailed account of how this can be done in § 3.2 below.

The calculus has other interesting aspects to it. For a thorough discussion on these aspects, and a detailed comparison between the communication primitives of Boxed and Mobile Ambients the reader is referred to [BCC01]. Here, instead, we focus our attention to security issues, and move on to multilevel security.

MAC Security. In MAC security, the behavior of system is described by a two-dimensional *Access Control Matrix* M indexed over a set S of *subjects* and a set O of *objects*, and whose values are *access modes* $\mathcal{A}, \mathcal{B} \in \{\mathsf{w}, \mathsf{r}, \mathsf{rw}, \mathsf{shh}\}$. $\mathsf{M[s,o]}{=}\mathsf{w}$ (respectively $\mathsf{r}, \mathsf{rw}, \mathsf{shh}$) indicates that subject s has write (respectively, read, read&write, no) access to object o.

For multilevel security, one presupposes a lattice $(\Sigma, \preceq)$ of *security levels* (ranged over by ρ, σ, τ), and a function $\mathit{level} : \mathsf{S} \cup \mathsf{O} \to \Sigma$. A *security policy* is a ternary boolean predicate $\mathcal{P}$ on subject levels, object levels, and access modes. An access control matrix M satisfies a security policy $\mathcal{P}$ if for every $\mathsf{s}{\in}\mathsf{S}$, $\mathsf{o}{\in}\mathsf{O}$, $\mathcal{P}(\mathit{level}(\mathsf{s}),\mathit{level}(\mathsf{o}),\mathsf{M[s,o]})$ holds true. Military (no read-up, no write-down) and commercial (no read-up, no write-up) security can then be formally defined as

[3] The possibility to associate owners to channels is the reason why we do not consider *shared channels* in the style of [CGZ01], that is, we do not have reductions such as, say,

$$(x)^n P \mid n[\, \langle M\rangle^{\uparrow} Q \mid R\,] \rightarrow P\{x := M\} \mid n[\, Q \mid R\,].$$

follows:

$$\mathcal{P}_{Mil}(\rho,\sigma,\mathsf{r}) \triangleq \sigma \preceq \rho \qquad \mathcal{P}_{Com}(\rho,\sigma,\mathsf{r}) \triangleq \sigma \preceq \rho$$
$$\mathcal{P}_{Mil}(\rho,\sigma,\mathsf{w}) \triangleq \rho \preceq \sigma \qquad \mathcal{P}_{Com}(\rho,\sigma,\mathsf{w}) \triangleq \sigma \preceq \rho$$
$$\mathcal{P}_{Mil}(\rho,\sigma,\mathsf{rw}) \triangleq \sigma = \rho \qquad \mathcal{P}_{Com}(\rho,\sigma,\mathsf{rw}) \triangleq \sigma \preceq \rho$$
$$\mathcal{P}_{Mil}(\rho,\sigma,\mathsf{shh}) \triangleq \mathit{true} \qquad \mathcal{P}_{Com}(\rho,\sigma,\mathsf{shh}) \triangleq \mathit{true}$$

EXCURSUS. *In process algebras, it is interesting to take a powerset of security labels* $(2^L, \subseteq)$ *as lattice of security levels. Based on that, it is possible to use standard* π*-calculus restrictions to dynamically define security levels:* $(\boldsymbol{\nu}\ell\colon L)(\boldsymbol{\nu}x\colon \{\ell\})P$*. New formalizations of* Discretionary Access Control *policies (DAC) are then possible if, in addition, one also allows security labels to be communicated over channels. We discuss this possibility with a brief* aperçu *in Section 4.*

3.2 A Type System for MAC Multilevel Security

The type system results from a rather simple refinement of the type system for Boxed Ambients defined in [BCC01]. As in that case, ambient and process types are defined as two-place constructors describing the types of exchanges that may occur locally and with the enclosing context. Interestingly, this simple type structure is all that is needed to give a full account of ambient interaction. This is a consequence of (*i*) there being no way for ambients to communicate directly across more than one boundary, and (*ii*) communication being the only means for ambient to interact.

Multilevel security is accounted for in the type system by instrumenting the structure of types to include additional information about the security level associated with each ambient (viewed as subject or object) and the access mode of the ambient's exchange types. The resulting syntax of types, as well as its intended meaning, are defined as follows, where the metavariable $\mathcal{A}$ ranges over access modes:

Expression Types	W	::=	$\sigma\mathsf{Amb}[E, F^{\mathcal{A}}] \mid \sigma\mathsf{Cap}[E^{\mathcal{A}}] \mid W_1 \times \cdots \times W_n$
Exchange Types	E, F	::=	$\mathsf{shh} \mid W$
Process Types	T	::=	$\sigma\mathsf{Pro}[E, F^{\mathcal{A}}]$

Ambient types $\sigma\mathsf{Amb}[E, F^{\mathcal{A}}]$: the type of ambients with clearance σ, enclosing processes whose local and upward exchanges are of type E and F; the upward exchanges have mode $\mathcal{A}$.

Capability types $\sigma\mathsf{Cap}[F^{\mathcal{A}}]$: the type of capabilities exercised within an ambient of clearance σ, whose upward exchanges have type F and mode $\mathcal{A}$.

Process types $\sigma\mathsf{Pro}[E, F^{\mathcal{A}}]$: the type of processes running at clearance σ, whose local and upward exchanges are, respectively, of type E and F. The tag $\mathcal{A}$ defines the mode in which the process accesses the channel located in its parent ambient.

In all cases, the type shh indicates no exchange, that is, absence of input *and* output. The syntax allows the formation of the types $\mathsf{Amb}[E, \mathsf{shh}^{\mathcal{A}}]$, $\mathsf{Cap}[\mathsf{shh}^{\mathcal{A}}]$,

and $\mathsf{Pro}[E, \mathsf{shh}^{\mathcal{A}}]$. These types are convenient in stating definitions and typing rules: to make sense of them, we stipulate that $\mathsf{shh}^{\mathcal{A}} = \mathsf{shh}$ for any access $\mathcal{A}$.

To enhance the flexibility of the type system, we introduce the following subtype relation over exchange types.

Definition 1 (*Exchange Subtyping). Let $\leqslant$ be the smallest reflexive and transitive relation over exchange types satisfying the following axioms for every exchange type E and access mode $\mathcal{A}$:*
$\mathsf{shh} \leqslant E, \quad \mathsf{shh} \leqslant E^{\mathcal{A}}, \quad E^{\mathsf{r}} \leqslant E^{\mathsf{rw}}, \quad E^{\mathsf{w}} \leqslant E^{\mathsf{rw}}$ □

Exchange subtyping is not used in conjunction with subsumption. Subtyping may be lifted to capability and process types to allow sound uses of subsumption. This enhanced form of subtyping is studied in [BCC01], but is essentially orthogonal to the subject of our present discussion. We therefore disregard it, and move on to illustrate the typing rules.

The typing rules are presented in Figure 1, and discussed next. The rules (In) and (Out) define the constraints for ambient mobility. They explain why capability types are built around a single component, and motivate the subtyping relation over exchange types. The intuition of the (In) is as follows: if $\mathsf{Cap}[F]$ is the type of the capability, say $\mathsf{in}\ n$, then $\mathsf{in}\ n$ is exercised within an ambient, say m, with upward exchanges F. Now, for the move of m into n to be safe, one must ensure that the local exchanges of n also have type F. In fact, one may be more liberal, and only require type compatibility between the upward exchanges of m and the local exchanges of n: this explains the premise $E \leqslant G$. The predicate $\mathcal{P}$ provides a guarantee that after moving, the ambient will still satisfy the security policy. Dual reasoning applies to the (Out) rule: upward exchanges by the exiting ambient must have the same (in fact, $\leqslant$-compatible) type as the upward exchanges of the ambient being exited. The security policy is enforced, in this case, directly by the subtyping relation over exchange types. It is worth noting that upward silent ambients (that is, ambients whose upward exchanges have type shh) can freely move across ambient boundaries. This is a consequence of our interpretation of capabilities, and of how $\leqslant$ is defined: capabilities exercised within upward silent ambients have type $\sigma\mathsf{Cap}[\mathsf{shh}]$ and $\mathsf{shh} \leqslant E$ for every E.

The rule (Amb), for typing ambients, defines the constraints that must be satisfied by P to legally be enclosed in a: specifically, the type of the upward exchanges performed by P, must comply with the security policy defined by the predicate $\mathcal{P}$ and must be a subtype of the local exchanges of the current ambient (that is either shh or G). As an example, if P tries to read from the channel located in the ambient that encloses a, then, to avoid a *read up* operation, the clearance of P (i.e. that of the ambient a) must be higher than that of the accessed channel (i.e. that of the ambient enclosing a).

The other interesting rules are those for communication. Local communication, i.e. local access within an ambient, needs no security constraint. The rules (Input M) and (Output M) govern input/output to subambients. Besides connecting the types of the input-output processes and their continuations, the rules also enforce the constraints that processes at clearance σ read only from

Typing of Expressions

$$\text{(Project)}\ \frac{\Gamma(n) = W}{\Gamma \vdash n : W} \qquad \text{(Tuple)}\ \frac{\Gamma \vdash M_i : W_i \quad \forall i \in 1..k}{\Gamma \vdash (M_1, \ldots, M_k) : W_1 \times \cdots \times W_k} \qquad \text{(Path)}\ \frac{\Gamma \vdash M_i : \sigma\mathsf{Cap}[E^{\mathcal{A}}] \quad i = 1,2}{\Gamma \vdash M_1.M_2 : \sigma\mathsf{Cap}[E^{\mathcal{A}}]}$$

$$\text{(In)}\ \frac{\Gamma \vdash M : \rho\mathsf{Amb}[G, H^{\mathcal{B}}] \quad \mathcal{P}(\sigma, \rho, \mathcal{A}) \quad E \leqslant G}{\Gamma \vdash \mathsf{in}\ M : \sigma\mathsf{Cap}[E^{\mathcal{A}}]} \qquad \text{(Out)}\ \frac{\Gamma \vdash M : \rho\mathsf{Amb}[G, H^{\mathcal{B}}] \quad E^{\mathcal{A}} \leqslant H^{\mathcal{B}}}{\Gamma \vdash \mathsf{out}\ M : \sigma\mathsf{Cap}[E^{\mathcal{A}}]}$$

Typing of Processes

$$\text{(Prefix)}\ \frac{\Gamma \vdash M : \sigma\mathsf{Cap}[F^{\mathcal{A}}] \quad \Gamma \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}{\Gamma \vdash M.P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]} \qquad \text{(Parallel)}\ \frac{\Gamma \vdash P_i : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}] \quad i = 1,2}{\Gamma \vdash P_1 \mid P_2 : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}$$

$$\text{(Amb)}\ \frac{\Gamma \vdash a : \sigma\mathsf{Amb}[E, F^{\mathcal{A}}] \quad \Gamma \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}] \quad \mathcal{P}(\sigma, \rho, \mathcal{A}) \quad F \leqslant G}{\Gamma \vdash a[\,P\,] : \rho\mathsf{Pro}[G, H^{\mathcal{B}}]}$$

$$\text{(Input } \star\text{)}\ \frac{\Gamma, x : W \vdash P : \sigma\mathsf{Pro}[W, F^{\mathcal{A}}]}{\Gamma \vdash (x : W)P : \sigma\mathsf{Pro}[W, F^{\mathcal{A}}]} \qquad \text{(Output } \star\text{)}\ \frac{\Gamma \vdash M : W \quad \Gamma \vdash P : \sigma\mathsf{Pro}[W, F^{\mathcal{A}}]}{\Gamma \vdash \langle M \rangle P : \sigma\mathsf{Pro}[W, F^{\mathcal{A}}]}$$

$$\text{(Input } \uparrow\text{)}\quad \mathcal{A} \in \{\mathsf{r}, \mathsf{rw}\}\ \frac{\Gamma, x : W \vdash P : \sigma\mathsf{Pro}[E, W^{\mathcal{A}}]}{\Gamma \vdash (x : W)^{\uparrow} P : \sigma\mathsf{Pro}[E, W^{\mathcal{A}}]} \qquad \text{(Output } \uparrow\text{)}\quad \mathcal{A} \in \{\mathsf{w}, \mathsf{rw}\}\ \frac{\Gamma \vdash M : W \quad \Gamma \vdash P : \sigma\mathsf{Pro}[E, W^{\mathcal{A}}]}{\Gamma \vdash \langle M \rangle^{\uparrow} P : \sigma\mathsf{Pro}[E, W^{\mathcal{A}}]}$$

$$\text{(Input } M\text{)}\ \frac{\Gamma, x : W \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}] \quad \Gamma \vdash M : \rho\mathsf{Amb}[W, U^{\mathcal{B}}] \quad \mathcal{P}(\sigma, \rho, \mathsf{r})}{\Gamma \vdash (x : W)^{M} P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}$$

$$\text{(Output } M\text{)}\ \frac{\Gamma \vdash N : W \quad \Gamma \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}] \quad \Gamma \vdash M : \rho\mathsf{Amb}[W, U^{\mathcal{B}}] \quad \mathcal{P}(\sigma, \rho, \mathsf{w})}{\Gamma \vdash \langle N \rangle^{M} P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}$$

$$\text{(dead)}\ \frac{}{\Gamma \vdash \mathbf{0} : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]} \qquad \text{(Replication)}\ \frac{\Gamma \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}{\Gamma \vdash\ !P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]} \qquad \text{(New)}\ \frac{\Gamma, x : W \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}{\Gamma \vdash (\boldsymbol{\nu} x : W)P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]}$$

Fig. 1. Typing Rules

(resp. write only to) ambients of clearance ρ compatible with σ according to the given security policy.

We conclude with the (INPUT $\uparrow$) and (OUTPUT $\uparrow$) for upward input/output which, perhaps surprisingly, do not impose any security constraint: that is because security on upward communication is already regulated by the ambient rule, and by the rules governing mobility.

The type system satisfies standard properties, notably, Subject Reduction:

Theorem 1. *If* $\Gamma \vdash P : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]$ *and* $P \rightarrow Q$, *then* $\Gamma \vdash Q : \sigma\mathsf{Pro}[E, F^{\mathcal{A}}]$.

However, the main purpose of types is to statically detect access violations. It is a simple technical matter to show the soundness of our type system. Let *level* be the function that associates the types $\sigma\mathsf{Amb}[E, F^{\mathcal{A}}]$ and $\sigma\mathsf{Cap}[E^{\mathcal{A}}]$ to σ. We decorate reduction with a function ℓ that associate names to security levels. The definition is straightforward in all cases, except for the case of restrictions:

$$\frac{P \quad \rightarrow_{\ell,(x:\, level(W))} \quad Q}{(\boldsymbol{\nu} x{:}W)P \quad \rightarrow_{\ell} \quad (\boldsymbol{\nu} x{:}W)Q}$$

Also, we instrument this form of labeled reduction with *error rules*.

$$\begin{array}{lll}
(e\text{-}input\ n) & m[\![\, (x{:}W)^n P \mid n[\![\, \langle M\rangle Q \mid R \,]\!] \mid S \,]\!] \rightarrow_{\ell} \mathtt{err} & \text{if } \neg\mathcal{P}(\ell(m), \ell(n), \mathsf{r}) \\
(e\text{-}input \uparrow) & m[\![\, \langle M\rangle P \mid n[\![\, (x)^{\uparrow} Q \mid R \,]\!] \mid S \,]\!] \rightarrow_{\ell} \mathtt{err} & \text{if } \neg\mathcal{P}(\ell(n), \ell(m), \mathsf{r}) \\
(e\text{-}output\ n) & m[\![\, \langle M\rangle^n P \mid n[\![\, (x) Q \mid R \,]\!] \mid S \,]\!] \rightarrow_{\ell} \mathtt{err} & \text{if } \neg\mathcal{P}(\ell(m), \ell(n), \mathsf{w}) \\
(e\text{-}output \uparrow) & m[\![\, (x) P \mid n[\![\, \langle M\rangle^{\uparrow} Q \mid R \,]\!] \mid S \,]\!] \rightarrow_{\ell} \mathtt{err} & \text{if } \neg\mathcal{P}(\ell(n), \ell(m), \mathsf{w})
\end{array}$$

In addition, we have structural rules that propagate errors from a process to its enclosing terms. Finally, given a type environment Γ, we say that ℓ is Γ-compatible if for all $x \in dom(\Gamma)$, one has $\ell(x) = level(\Gamma(x))$. If we assume that `err` is a distinguished process, with no type, it is very easy to verify that no system containing an occurrence of `err` can be typed in our type system. Absence of run time errors may now be stated as follows:

Theorem 2 (Soundness). *For every* Γ, P *and* Γ*-compatible* ℓ, *if* $\Gamma \vdash P : T$, *then* $P \not\rightarrow_{\ell} \mathtt{err}$.

4 Examples

In this section we consider several examples from the literature on security and related issues, and show how to handle them with Boxed Ambients.

Wrappers. As a solution for resource protection and access control in wide-area networks, Sewell and Vitek [SV00] propose to use *wrappers* to isolate potentially malicious programs. Their framework is based on an extension of the π-calculus, known as the *boxed* π-calculus: *wrappers* enable a programming style in which incoming code can be secured into a *box*, and its interactions with the enclosing environment filtered by the *wrapper* that only forwards legitimate messages between the boxed program and its enclosing environment via secured channels.

The paradigmatic example of that work can be rephrased in our syntax as follows:

$$(\nu a, b)\left(a[P] \mid !(x)^a\langle x\rangle^b \mid b[Q]\right)$$

P and Q are arbitrary processes encapsulated in ambients ("named boxes" in [SV00] terminology) with private names a and b, placed in parallel with a forwarder process from ambient a to ambient b. The configuration above is interesting when P and Q are distrusted processes since ambient boundaries forbid them to interact directly, while the restrictions ensure that the only possible interaction with the environment is with the forwarder process $!(x)^a\langle x\rangle^b$. This is the way for Boxed-π to enforce a security policy that prevents (i) Q from leaking secrets to P and (ii) P and Q from corrupting the environment. This holds true also in Boxed Ambients. Besides that, in Boxed Ambients we have the choice of other alternatives. For example, to enforce (i) we can use military security and ensure a more general property: if we assign to a a security level strictly greater than the level of b, then our type system statically ensures that there cannot be any unwanted access from Q to P. To enforce also (or only) the property (ii) we can once more rely on military (but also commercial) security, and assign to the environment a security level incomparable with the levels of a and b. Then the two processes cannot access and corrupt the resources of the environment.

Asynchronous Communication. In wide-area networks it is hardly reasonable to rely only on synchronous communication (see [Car00] for discussion, and [BV02] for experience with implementations). In [BCC01], we show how to account for asynchronous output in Boxed Ambients and discuss the consequences of this choice. Besides other changes, asynchronous communication results from introducing the following new reduction rules

$$(\textit{asynch output } n) \qquad \langle M\rangle^n P \mid n[\,Q\,] \rightarrow P \mid n[\,\langle M\rangle \mid Q\,]$$
$$(\textit{asynch output } \uparrow) \qquad n[\,\langle M\rangle^{\uparrow} P \mid Q\,] \rightarrow \langle M\rangle \mid n[\,P \mid Q\,]$$

to direct an output in the appropriate ambient. The enhanced flexibility obtained using asynchronous communications is paid by lesser security since now we loose the total mediation principle. Consider the following two examples:

$$a[\,(x:W)^b P \mid b[\,c[\,\langle M\rangle^{\uparrow} \mid Q\,]\,]\,] \qquad b[\,a[\,(x:W)^{\uparrow} P\,] \mid c[\,\langle M\rangle^{\uparrow} Q\,]\,]$$

they both implement a *covert channel* between ambients a and c, since with asynchronous reductions, they evolve into $a[\,(x{:}W)^n P \mid b[\,\langle M\rangle \mid c[\,Q\,]\,]\,]$ and $b[\,a[\,(x{:}W)^{\uparrow} P\,] \mid \langle M\rangle \mid c[\,Q\,]\,]$ respectively. In both cases by a further reduction step the ambient a gets hold of the message $\langle M\rangle$ without any mediation of b.

These kind of covert channels are two examples of security breaches that cannot be prevented by the primitives of the calculus and it is where the use of security policies comes to rescue. In both cases it just suffices to assign to ambient a a clearance strictly lower than that of b to make the read operation performed by a illegal in both commercial and military security (since it would be a read-up) and, as such, statically detected.

Firewalls. We now look at the protocol for firewall crossing defined in [CG99] and refined in [LS00], and show how it can be defined with Boxed Ambients.

The idea of the protocol is to let an *Agent* cross a *Firewall* by means of a shared key k.

$$\mathit{Firewall} = (\boldsymbol{\nu} f) f[\, k[\, \mathsf{out}\ f.\langle \mathsf{in}\ f\rangle^a \,] \mid \ldots \,] \qquad \mathit{Agent} = a[\, \mathsf{in}\ k.(x)\mathsf{out}\ k.x.Q \,]$$

$$\begin{array}{lll} (\boldsymbol{\nu} k)(\mathit{Firewall} \mid \mathit{Agent}) & \rightarrow^* & (\boldsymbol{\nu} k, f) f[\ldots] \mid k[\, \langle \mathsf{in}\ f\rangle^a \mid a[\, (x)\mathsf{out}\ k.x.Q \,] \,] \\ & \rightarrow^* & (\boldsymbol{\nu} k, f) f[\ldots] \mid k[\, a[\, \mathsf{out}\ k.\mathsf{in}\ f.Q \,] \,] \\ & \rightarrow^* & (\boldsymbol{\nu} f) f[\ldots \mid a[\, Q \,] \,] \end{array}$$

The *Firewall*, with secret name f, sends out a pilot ambient k to guide the agent inside. The name k is a password that the agent a must know in order to enter (to acquire the path to) the firewall.

Besides authenticating entering agents, the firewall must in general provide other security guarantees. For example, the firewall administrator may want to ensure that processes inside the firewall can access the resources of an entered agent, but not the converse. This can be enforced with commercial security, by the following type assignments: $f : \phi\mathsf{Amb}[E, F^{\mathcal{A}}]$ and $k : \kappa\mathsf{Amb}[\mathsf{shh}, \mathsf{shh}]$, where E and $F^{\mathcal{A}}$ are appropriate types, and ϕ and κ are security levels such that $\kappa \prec \phi$. To illustrate the effects of this type assignments, consider a generic agent a (whose definition may differ from that of *Agent*) that wants to enter the firewall, and assume that $a : \alpha\mathsf{Amb}[G, H^{\mathcal{B}}]$. To cross the firewall, a must comply with the protocol and therefore accept write requests from k. With commercial security, this is possible only if $\alpha \preceq \kappa$ and this, by transitivity, implies that $\alpha \prec \phi$. Moreover, commercial security forbids low-level ambients (such as a) contained in high-level ambients (such as f) to perform upward communications. But then, $\alpha \prec \phi$ implies $\mathcal{B} = \mathsf{shh}$. In summary, the type assignment enforces for a a security level strictly smaller than the level of f, and the policy we choose ensures that the agents that enter the firewall f cannot directly access to local resources of f, as expected.

The protocol we just discussed depends on the assumption that the firewall knows the name of the entering agents. This is clearly unrealistic but, fortunately, easily remedied, as we show next. In order to provide guarantees of commercial security, the new protocol assumes that the agent know two passwords, k_1 and k_2, to cross the firewall.

$$\begin{array}{ll} \mathit{Firewall}_2 = (\boldsymbol{\nu} f) f[\, k_1[\, \mathsf{out}\ f.(y{:}A)^{k_2}\mathsf{in}\ f.\langle N\rangle^y\langle y\rangle \,] \mid (y{:}A)^{k_1} P\{y\} \,] \\ \mathit{Agent}_2 \quad = a[\, \mathsf{in}\ k_1.\,(k_2[\, \mathsf{out}\ a.\langle a\rangle \,] \mid (x)\mathsf{out}\ k_1.Q) \,] \end{array}$$

$$\begin{array}{ll} (\boldsymbol{\nu} k, k')(\mathit{Firewall}_2 \mid \mathit{Agent}_2) & \\ \quad \rightarrow^* & (\boldsymbol{\nu} k_1, k_2, f) f[\ldots] \mid k_1[\, (y{:}A)^{k_2}\mathsf{in}\ f.\langle N\rangle^y\langle y\rangle \mid k_2[\, \langle a\rangle \,] \mid a[\, (x)\mathsf{out}\ k_1.Q \,] \,] \\ \quad \rightarrow^* & (\boldsymbol{\nu} f) f[\, a[\, Q \,] \mid P\{a\} \,] \end{array}$$

Again, the protocol starts with the agent a entering the pilot ambient k_1. The ambient k_1, in turn, reads from k_2 the name of the agent, carries the agent inside the firewall, and communicate the name a in order to let the firewall interact with the incoming agent. The message $\langle N\rangle^y$ is just used for synchronization, to ensure that the agent a exits the pilot ambient k_2 only after k_2 is back into the firewall. Note that the firewall may interact only with agents whose type

is the one used for the variable y. It would be nice to add to subtyping (and tuple?) polymorphism other forms of polymorphism (for example on the lines of the excellent [AKPG01]) so that to extend the possible interactions with the incoming agents.

Trojan Horses. In [BC01] a type system that can statically detect Trojan horses is defined. The motivating example is the system $a[\,\mathsf{in}\ c.P\,] \mid b[\,\mathsf{in}\ a.\mathsf{out}\ a.\mathsf{in}\ d.Q\,] \mid c[\,R \mid d[\,S\,]\,]$, where the ambient d contains confidential data that should be made available to ambients running within c but not to ambients that will enter c. The question is whether c should let a enter or not. Apparently a does not attempt to access d; nevertheless the move must be forbidden since b can use it as a Trojan Horse to enter c ad then access d.

$$a[\,\mathsf{in}\ c.P\,] \mid b[\,\mathsf{in}\ a.\mathsf{out}\ a.\mathsf{in}\ d.Q\,] \mid c[\,R \mid d[\,S\,]\,] \quad \rightarrow^* \quad c[\,R \mid a[\,P\,] \mid d[\,S \mid b[\,Q\,]\,]\,]$$

The attack is detected in [BC01] by means of a type system that traces the behavior of a, revealing the move of b into a and hence a chance for the attack. In [BC01] it is also shown how to perform this verification even when c runs in a possibly untyped context. Here we can obtain the same effect by setting the clearance of d to a level that is incomparable to any security level that is defined outside c. As we hinted in the excursus in Section 3.1 this can be obtained by using security labels with limited scope: $(\boldsymbol{\nu}\ell : L)(\boldsymbol{\nu}d : \{\ell\}\mathsf{Amb}[E, \mathsf{shh}])c[\,R \mid d[\,S\,]\,]$. No matter how and where the name d is communicated and whether c is in a well typed-context or not, if we impose commercial security only the processes that are already inside the ambient c can access information contained in d. Indeed to reproduce the initial configuration, c must communicate to b the name d. But, unlike what happens in Mobile Ambients, revealing the name of an ambient does not imply granting access to its resources.

5 Related Work and Conclusions

We have studied the problem of MAC security for Mobile Ambients (MA), and argued that the calculus is not fully adequate to express security concepts. As a solution, we have presented Boxed Ambients (BA), whose primitives provide elegant and natural mechanisms of resource access control. We conclude with discussion on related work on security for calculi of mobility.

The Dπ Calculus. In [HR00b] Hennessy and Riley discuss a type system for resource protection in the $D\pi$-calculus, a distributed variant of π-calculus where processes are located, and may migrate across locations. In Dπ, communication occurs via named channels that are associated with read/write capabilities: the type system controls that processes accessing a resource possess the appropriate capability.

In our approach, instead, in order to classify an access as legal or illegal, the type system checks that the security levels of subject and object satisfy the constraints imposed by the security policy for that access. A further difference is that in $D\pi$ the topology of locations is completely flat, while in BA ambients

may be nested at will: the interplay between the dynamic nesting structure determined by moves, and the dynamic binding of the parent location $\uparrow$ for upward communication makes access control for BA more complex. In [HR99], the type system for Dπ is extended to cope with *partially typed networks*, in which some of the agents (and/or locations) are untyped, hence untrusted: type safety for such networks requires a form of dynamic type checking. Plans for future work on BA include extensions along similar lines.

The Security Pi Calculus. In [HR00a], Hennessy and Riely discuss the *security π-calculus*, a variant of the π-calculus in which processes are syntactically defined as running at a given security level, and whose type system ensures that low-level processes never gain access to high-level resources. In BA, instead, we assume that clearances are specified by types, and the security level associated to an ambient type represents the clearance of resources contained in that ambient, as well as the clearance of the agent it implements. Besides resource protection, in [HR00a], the authors also investigate non-interference, trying to provide guarantees against implicit information flow from high levels to lower levels. To that end, they check that the clearance of values are compatible with clearance of channels along which they (the values) are exchanged. Furthermore, they show that a notion of non-interference based on *may testing* equivalence is soundly captured by the type system.

We did not study these issues in detail (but see § 4 for examples of information flow) in this paper. In fact, in its current version, the type system only checks the clearance of subjects against the clearance of objects, disregarding the clearance of the values. We believe that it is possible to study BA in a similar way taking these issues into account. We leave this and the study of information flow and non-interference as subject of future work.

Typing of Mobility and Security for Mobile Ambients. Our type system is clearly related to other typing systems developed for Mobile Ambients. In [CG99] types guarantees absence of type confusion for communications. The type systems of [CGG99] and [Zim00] provide control over ambients moves and opening. Furthermore, the introduction of *group* names [CGG00] and the possibility of creating fresh group names, give flexible ways to statically prevent unwanted propagation of names. The powerful type discipline for Safe Ambients, presented in [LS00], adds finer control over ambient interactions and prevents all *grave interferences*.

All these approaches are orthogonal to the resource access control mechanisms we studied. We believe that similar typing disciplines as well as the use of group names, can be adapted to Boxed Ambients to obtain similar strong results. A paper more directly related to ours is [DCS00], where ambient types are associated with security levels in ways similar to ours. The difference is that in [DCS00], security checks are over opening and moves, while in our work we focus on read and write operations.

A final mention goes to [BC01], [NNHJ99] and [NN00], where control and data flow analysis, rather than typing disciplines, are used to check security properties of Ambients.

Other Languages and Calculi. In the literature on language-based security, several papers deal with access control techniques and information flow. Two examples are [DNFP99] and [LR99]. In [DNFP99], authors take a similar approach to that of Hennessy and Riley, based on a variant of Linda with multiple "tuple spaces". In [LR99], Leroy and Rouaix focus on integrity of typed applets via access control.

Acknowledgments. Thanks to Luca Cardelli for insightful comments and discussion, and to the anonymous referees. The second author would like to thank Ilaria Castagna for several corrections she made on an early draft of the paper.

References

[AKPG01] T. Amtoft, A.J. Kfoury, and S.M. Pericas-Geertsen. What are polymorphically-typed ambients? In *ESOP 2001*, volume 2028 of *LNCS*, pages 206–220. Springer, 2001.

[BC01] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, pages 222–235, London, 2001. ACM Press.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. Technical report, L.I.E.N.S., 2001. Available at ftp.ens.fr/pub/dmi/users/castagna.

[BP76] D.E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation,. Technical Report MTR-2997, MITRE Corporation, Bedford, MA. March 1976.

[BV02] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 2002. To appear.

[Car00] L. Cardelli. Global computing. In *IST FET Global Computing Consultation Workshop*. 2000. Slides.

[CG98] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98*. ACM Press, 1998.

[CG99] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999.

[CGG99] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, number 1644 in LNCS, pages 230–239. Springer, 1999.

[CGG00] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science, pages 333–347. Springer, August 2000.

[CGZ01] G. Castagna, G. Ghelli, and F. Zappa. Typing mobility in the seal calculus. In *CONCUR 2001 (12th. International Conference on Concurrency Theory)*, Lecture Notes in Computer Science, Aahrus, Danemark, 2001. Springer. This same volume.

[DCS00] M. Dezani-Ciancaglini and I. Salvo. Security types for safe mobile ambients. In *Proceedings of ASIAN'00*, pages 215–236. Springer, 2000.

[DNFP99] R. De Nicola, G. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer, 1999.

[DoD85] US Department of Defense. Dod trusted computer system evaluation criteria, (the orange book). DOD 5200.28-STD, 1985.

[FGL+96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1996.

[Gol99] D. Gollmann. *Computer Security.* John Wiley & Sons Ltd., 1999.

[HR00a] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous π-calculus (extended abstract). In *Automata, Languages and Programming, 27th International Colloquium*, volume 1853 of *LNCS*, pages 415–427. Springer, 2000.

[HR00b] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 2000. To appear.

[HR99] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL'99*, pages 93–104. ACM Press, 1999.

[LR99] X. Leroy and F. Rouaix. Security properties of typed applets. In *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 147–182. Springer, 1999.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.

[NN00] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL'00*, pages 135–148. ACM Press, 2000.

[NNHJ99] F. Nielson, H. Riis Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *Proc. CONCUR'99*, number 1664 in LNCS, pages 463–477. Springer, 1999.

[SV00] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *13th IEEE Computer Security Foundations Workshop*, 2000.

[VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in *LNCS*. Springer, 1999.

[Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proceedins of FoSSaCS'99*, volume 1784 of *LNCS*, pages 375–390. Springer, 2000.

Synchronized Hyperedge Replacement with Name Mobility

A Graphical Calculus for Mobile Systems

Dan Hirsch[1] * and Ugo Montanari[2] **

[1] Departamento de Computación, Universidad de Buenos Aires,
dhirsch@dc.uba.ar
[2] Dipartimento di Informatica, Universitá di Pisa,
ugo@di.unipi.it

Abstract. The design of software systems that include mobility or dynamic reconfiguration of their components is becoming more frequent. Consequently, it is necessary to have the right tools to handle their description specially in the design phase. With this in mind and understanding the relevance of visual languages at the design level, we present in this paper a graphical model using *Synchronized Hyperedge Replacement Systems* with the addition of name mobility. This method gives a solid foundation for graphical mobile calculi which are well-suited for high level description of distributed and concurrent systems.

1 Introduction

The design of software systems that include mobility or dynamic reconfiguration of their components is becoming more frequent. Consequently, it is necessary to have the right tools to handle their description. More specifically, it is fundamental to be able to cope with these type of requirements in the design phase and specially for software architectures.

With this in mind and understanding the relevance of visual languages specially at the design level, we present in this paper a graphical model using *Synchronized Hyperedge Replacement Systems* with the addition of name mobility. In this way we obtain the good characteristics of a graphical calculus together with the expressive power to describe complex mobile systems. The capability of creation and sharing of ports together with multiple simultaneous synchronizations give us a very powerful tool to specify more complex evolutions, reconfiguring multiples components by identifying specific ports. Apart from the graphical side, we can relate our calculus with π-calculus [6]. The difference is that π-calculus is *sequential* (i.e. it allows only one synchronization at a time) while

* Partially supported by UBACyT Projects TW72 and X094.
** Partially supported by CNR Projects *Metodi per Sistemi Connessi mediante Reti*; by MURST project *Theory of Concurrency* and *Higher Order and Types* ; by TMR Network GETGRATS; and by Esprit Working Groups *APPLIGRAPH*.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 121–136, 2001.
© Springer-Verlag Berlin Heidelberg 2001

synchronized rewriting allows multiple and concurrent synchronizations all over the graph.

In this setting, hypergraphs [8] represent systems and grammars represent system families (software architecture styles) while their dynamic evolution is model by synchronized rewriting. Following a self-organizing approach we use context-free grammars together with synchronized rewriting to model dynamic reconfiguration without the need of central coordination.

Synchronized replacement with name mobility was first introduced in [3] for modelling software architecture reconfigurations. The presentation in [3] was only for Hoare Synchronization and only with the possibility of sharing new nodes (i.e. new names) as in the πI-calculus [9]. Now, we allow to pass both new names and old names in a synchronization and give a revised definition of the synchronization transition systems. The new contributions of this paper are the introduction of the Milner Synchronization, the introduction of bounded nodes for a compositional presentation and to allow the synchronization of old names with new names. These three capabilities are necessary to model the π-calculus in section 4.2. A more detailed presentation of the ideas in this paper and the theorem proofs can be found in [4].

In the area of graph transformation and its application to system modelling there are interesting work (for example, [8,2]) where, in general, system transformations are represented with productions where their left hand sides are non context-free graphs (with exception of [7]). This means that they imply a centralized control that needs to know the complete map of the system, which is not well suited for distributed systems. Also none of these techniques includes any synchronization mechanism with mobility. On the other side, our use of context free productions with synchronization and mobility is a powerful tool for describing self organising distributed systems.

The formalism presented in this paper gives a solid foundation for graphical mobile calculi which are well-suited for high level description of distributed and concurrent systems, reflected by our main practical goal that is, once again, formalizing the description of software architecture styles including their reconfigurations.

2 Hypergraphs and Syntactic Judgements

In this section we introduce the notion of hypergraphs formalizing them as well formed syntactic judgements. For an extensive presentation on the foundations of Hyperedge Replacement Systems we refer to [1].

A *hyperedge*, or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,\ldots}$) and with as many (ordered) tentacles as the rank of its label. A set of *nodes* together with a set of such edges forms a *hypergraph* (or simply a graph) if each edge is connected, by its tentacles, to its *attachment* nodes. A graph is equipped with a set of external nodes identified by distinct names. In figure 1*a* you can see a graph with four edges of rank two and two external nodes (x and y). External nodes can be seen as the connecting points of a graph with its environment (i.e. the context).

Now, we present a definition of graphs as *syntactic judgements*, where nodes correspond to names, external nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE$.

Definition 1 (Graphs as Syntactic Judgements). *Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A* syntactic judgement *(or simply a judgement) is of the form $\Gamma \vdash G$ where,*

1. *$\Gamma \subseteq \mathcal{N}$ is a set of names (the external nodes of the graph).*
2. *G is a term generated by the grammar $G ::= L(\boldsymbol{x}) \mid G|G \mid (\nu y)G \mid nil$ where $\boldsymbol{x}$ is a vector of names, L is an edge label with $rank(L) = |\boldsymbol{x}|$ and y is a name.*

Let fn(G) denote the set of all free names of G, i.e. all names in G not bound by a ν operator. We demand that $fn(G) \subseteq \Gamma$.

We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2.

Definition 2 (Structural Congruence and Well-Formed Judgements).

- *Structural Congruence $\equiv$ on syntactic judgements obey axioms in Table 1.*
- *The* well-formed judgements *for constructing graphs over LE and $\mathcal{N}$ are those generated by applying the rules in Table 1 up to axioms of structural congruence.*

Table 1. Well-formed judgments

Structural Axioms

$(AG1)\ (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3)$ $(AG2)\ G_1|G_2 \equiv G_2|G_1$

$(AG3)\ G|nil \equiv G$ $(AG4)\ \nu x.\nu y.G \equiv \nu y.\nu x.G$

$(AG5)\ \nu x.G \equiv G\ if x \notin fn(G)$ $(AG6)\ \nu x.G \equiv \nu y.G\{y/x\}\ if\ y \notin fn(G)$

$(AG7)\ \nu x.(G_1|G_2) \equiv (\nu x.G_1)|G_2\ if\ x \notin fn(G_2)$

Syntactic Rules

$$(RG1)\ \frac{}{x_1, \ldots, x_n \vdash nil} \quad (RG2)\ \frac{L \in LE_m \quad y_i \in \{x_j\}}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}$$

$$(RG3)\ \frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2} \quad (RG4)\ \frac{\Gamma, x \vdash G}{\Gamma \vdash \nu x.G}$$

Axioms *(AG1)*, *(AG2)* and *(AG3)* define the associativity, commutativity and identity over *nil* for operation $|$, respectively. Axioms *(AG4)* and *(AG5)* state that the nodes of a graph can be hidden only once and in any order, and axioms *(AG6)* and *(AG7)* define alpha conversion of a graph with respect to its bounded names and the interplay between hiding and the operator for parallel composition, respectively.

Rule *(RG1)* creates a graph with no edges and n nodes and rule *(RG2)* creates a graph with n nodes and one edge labelled by L and with m tentacles (note that there can be repetitions among nodes in $\boldsymbol{y}$, i.e. some tentacles can be attached to the same node). Rule *(RG3)* allows to put together (using $|$) two graphs that share the same set of external nodes. Finally, rule *(RG4)* allows to hide a node from the environment.

If necessary, thanks to axiom *(AG4)*, we will write νX, with $X = \bigcup x_i$, to abbreviate $\nu x_1.\nu x_2 \ldots \nu x_n$. Note that using the axioms, for any judgement we can always have an equivalent normal form $\Gamma \vdash \nu X.G$, with G a subterm containing only composition of edges. It is clear from the above definitions that Γ and X can be made disjoint sets of nodes using the axioms and that $nodes(G) \subseteq (\Gamma \cup X)$. We can state the following correspondence theorem.

Theorem 1 (Correspondence of Graphs and Judgements). *Well-formed syntactic judgements up to structural axioms are isomorphic to graphs up to isomorphism.*

Ring Example. We use graphs to represent systems. In this context, edges are components and nodes are ports of communication. External nodes are connecting ports to the environment. Edges sharing a node mean that there is a communication link among them. So, let us take the graph in figure 1*a* that represents a ring of four components with two connecting ports. Edges representing components are drawn as boxes attached to their corresponding ports. The label of an edge is the name of the component and the arrow indicates the order of the attachment nodes. In this case we only have edges with two tentacles. Names in filled nodes identify external nodes and empty circles are bound nodes. Figure 1*b* shows how the corresponding well-formed judgement is obtained. Note that *(RG3)* needs the same set of names Γ in both premises.

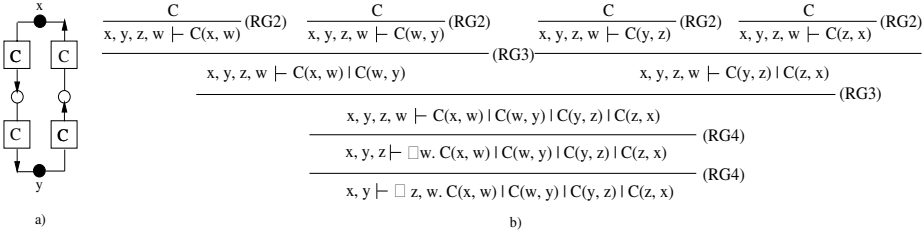


Fig. 1. The graph and the corresponding judgement for the ring example

3 Synchronized Hyperedge Replacement with Name Mobility

Synchronized edge replacement is obtained using graph rewriting combined with constraint solving. More specifically, we use *context-free* productions with actions that are used to coordinate the simultaneous rewriting of various productions.

The following definitions present an extension to synchronized replacement systems where we allow the declaration and creation of names on nodes and use

synchronized rewriting for name mobility. In this way it is possible to specify reconfigurations over the graphs by changing the connections between edges.

3.1 Synchronized Replacement Systems as Syntactic Judgements

A *context-free edge replacement production* rewrites a single edge into an arbitrary graph. Productions will be written as $L \to R$. A production $p = (L \to R)$ can be applied to a graph G yielding H ($G \Rightarrow_p H$) if there is an occurrence of an edge labeled by L in G. A result of applying p to G is a graph H which is obtained from G by removing an edge with label L, and embedding a fresh copy of R in G by coalescing its external nodes with the corresponding attachment nodes of the replaced hyperedge. This notion of edge replacement yields the basic steps in the derivation process of an edge replacement grammar.

To model coordinated rewriting, it is necessary to add some labels to the nodes in productions. Assuming to have an alphabet of actions *Act*, then we associate actions to some of the nodes. In this way, each rewrite of an edge must synchronize actions with (a number of) its adjacent edges and then all the participants will have to move as well (how many depends on the synchronization policy). It is clear that coordinated rewriting will allow the propagation of synchronization all over the graph where productions are applied.

A *synchronized edge replacement grammar*, or simply a grammar, consists of an initial graph and a set of productions. A derivation is obtained by starting with the initial graph and applying a sequence of rewriting rules, each obtained by synchronizing possibly several productions.

Now, for adding to productions the capability of sharing nodes we let a production to declare new names for the nodes it creates and to share these names and/or other existing names with the rest of the graph using the synchronization process. This is done in a production by adding to the action in a node, a tuple of names that it wants to communicate. Therefore, the synchronization of a rewriting rule has to match not only actions, but also the tuples of names. After the matching is obtained and the productions applied, the declared names that were matched are used to obtain the final graph by merging the corresponding nodes.

As is done in π-calculus, we allow to merge new nodes with other nodes (new or old). Merging among already existing nodes is not allowed. Relaxing this constraint, would permit fusions of nodes in the style of the *fusion*-calculus [10]. Instead, if we consider a syntactic restriction in which we allow merging new nodes only, we have the style of the πI-calculus [9]. These policies of which nodes are shared are independent of the synchronization mechanisms applied.

To formalize synchronized rewriting we use, as in section 2, judgements and define the notion of *transitions*.

Definition 3 (Transitions). *Let $\mathcal{N}$ be a fixed infinite set of names and* Act *a ranked set of actions, where each action $a \in Act$ is associated with an arity (indicating the number of nodes it can share). We define a* transition *as:*

$$\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'$$

$$\textit{with } \Lambda : \Gamma \multimap (Act \times \mathcal{N}^*) \qquad \Delta = \{z \mid \exists x.\ \Lambda(x) = (a, \boldsymbol{y}),\ z \notin \Gamma,\ z \in set(\boldsymbol{y})\}$$

A transition is represented as a logical sequent which says that G is rewritten into G' satisfying a *set of requirements* Λ. The free nodes of graph G' must include the free nodes of G and those new nodes (Δ) that are used in synchronization. Note that Δ is determined by the Γ and Λ of the corresponding transition.

The set of requirements $\Lambda \subseteq \Gamma \times Act \times \mathcal{N}^*$ is defined as a partial function in its first argument, i.e. if $(x, a, \boldsymbol{y}) \in \Lambda$ we write $\Lambda(x) = (a, \boldsymbol{y})$ with $arity(a) = |\boldsymbol{y}|$. With $\Lambda(x) \uparrow$ we mean that the function is not defined for x, i.e. that there is no requirement in Λ with x as first argument. Function $set(\boldsymbol{y})$ returns the set of names in vector $\boldsymbol{y}$. The definition of Λ as a function means that all edges in G_1 attached to node x that are participating in a synchronization, must satisfy the conditions of the corresponding synchronization algebra. The function is partial since not all nodes need to be loci of synchronization.

Note that to share only new nodes, it is enough to impose on Λ the condition that names of vectors $\boldsymbol{y}$ should not be in Γ ($set(\boldsymbol{y}) \cap \Gamma = \emptyset$). Then, Δ does not depend on Γ and can be written as, $\Delta = \bigcup_{xa\boldsymbol{y} \in \Lambda} set(\boldsymbol{y})$.

Definition 4 (Productions). *A* synchronized production*, or simply a production, is a special transition of the form,*

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda} x_1, \ldots, x_n, \Delta \vdash G$$

The context-free character of productions is here made clear by the fact that the graph to be rewritten consists of a single edge with distinct nodes. Productions are defined as general schemas that are applied over different graphs, so they will be alpha convertible with respect to the names in Δ. In this way, the new names can be arranged to avoid name clashing and a correct synchronization.

Definition 5 (Grammars). *Let $\mathcal{N}$ be a fixed infinite set of names, LE a ranked alphabet of labels and* Act *a ranked set of actions. A* grammar *consists of an initial graph $\Gamma_0 \vdash G_0$ and a set $\mathcal{P}$ of productions.*

A derivation *is a finite or infinite sequence of the form $\Gamma_0 \vdash G_0 \xrightarrow{\Lambda_1} \Gamma_1 \vdash G_1 \xrightarrow{\Lambda_2} \ldots \xrightarrow{\Lambda_n} \Gamma_n \vdash G_n \ldots$, where $\Gamma_{i-1} \vdash G_{i-1} \xrightarrow{\Lambda_i} \Gamma_i \vdash G_i$, $i = 1 \ldots n$ is a transition in the set $T(\mathcal{P})$ of transitions generated by $\mathcal{P}$. Transitions $T(\mathcal{P})$ are generated by $\mathcal{P}$ applying the transition rules of the chosen synchronization mechanism, as defined in the next sections.*

3.2 Hoare Synchronization

The first synchronization mechanism we present is *Hoare Synchronization*, where each rewrite of an edge must share on each attachment node the same action with all the edges connected to that node.

For example, consider n edges which share one node, such that no other edge is attached to that node, and let us take one production for each of these edges. Each of these productions has an action on that node (a_i for $i = 1 \ldots n$). If $a_i \neq a_j$ (for some i, j), then the n edges cannot be rewritten together (using these productions). If all a_i are the same, then they can move, via the context-sensitive rewriting rule obtained by merging the n context-free productions. The use of

synchronized graph productions in a rewriting system implies the application of several productions where all edges to be rewritten and sharing a node must apply productions that satisfy the same actions.

Given that Hoare synchronization requires that all edges sharing a node must participate in the synchronization, but since not all nodes need to be loci of synchronization, an identity action ε is defined which is required in a node by all the productions which do not synchronize on that node. We impose the condition that the identity action has arity zero, so if it is imposed on a node then no name can be shared on that node. In particular, to model edges which do not move in a transition we need productions with identity actions on their external nodes, where an edge with label L is rewritten to itself. This is called the *id production* $id(L)$. Then, the set $\mathcal{P}$ of productions must include productions $id(L)$ for all symbols L in LE. The corresponding judgements are as follows,

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\{x_1\varepsilon<>, \ldots, x_n\varepsilon<>\}} x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n)$$

For any relation $R \subseteq \Gamma \times Act \times \mathcal{N}^*$ we define $n(R) = \bigcup_{(x,a,\boldsymbol{y})\in R} set(\boldsymbol{y})$ and will call a mapping $\rho : \Delta \to n(R)$ the most general unifier[1] (mgu) of R (with $\Delta = n(R) \cap \Gamma$) whenever $\rho(R)$ is a function in its first argument and if, of all ρ' with this property, ρ identifies the minimal number of names.

The *mgu* is necessary to resolve the identification of names (i.e. nodes) that is consequence of a synchronization operation and to avoid name capture.

Definition 6 (Hoare Transition System). *Let $< G_0, \mathcal{P} >$ be a grammar. All transitions $T(\mathcal{P})$ using Hoare synchronization are obtained from the transition rules in Table 2.*

Rule *(ren)* is necessary to allow to apply a production to graphs with edges that may have several tentacles attached to the same node (this is done by ξ that is a possibly non-injective substitution), and also to adequate the free names to the names of the graph where the production is applied. Notice that $\rho(\xi(\Lambda))$ is still a function and requirements on nodes identified by ξ must coincide. Also, isolated nodes are added (those in Γ) with no requirement on them. Remember that for any transition, as presented in definition 3, Δ is uniquely identified by the corresponding Γ and Λ.

Rule *(com)* is the one that completes the synchronization process. Given that all edges must participate, Hoare synchronization is modeled as the union of the synchronization requirements ($\rho(\Lambda_1 \cup \Lambda_2)$) where ρ assures that the rule can only be applied when the requirements on all the nodes are satisfied and the shared nodes are actually identified. Condition $\Delta_1 \cap \Delta_2 = \emptyset$ avoids name capture.

Rule *(open)* allows to share with the environment a node that was originally bounded. This rule may be used for sharing a port of communication that was

[1] The mapping ρ is exactly the most general unifier of the equations $(a = b) \wedge (\boldsymbol{y} = \boldsymbol{z})$ (whenever $(x, a, \boldsymbol{y}), (x, b, \boldsymbol{z}) \in R$) and is unique up to injective renaming. It does not exist if there are tuples $(x, a, \boldsymbol{y}), (x, b, \boldsymbol{z}) \in R$ with $a \neq b$ or if the equations $\boldsymbol{y} = \boldsymbol{z}$ imply an equation $v = w$ with v, w different old names. Thus the external nodes (i.e., $x \in \Gamma$) that appear in $n(R)$ are considered constants. In this way new names are unified with either new or old names, but it is not possible to have a unification among old names (two different constants cannot be unified).

Table 2. Transition Rules for Hoare Synchronization

$$(ren)\ \frac{x_1,..,x_n \vdash L(x_1,..,x_n) \xrightarrow{\Lambda} x_1,..,x_n, \Delta \vdash G \in \mathcal{P}}{\xi(x_1,..,x_n), \Gamma \vdash \xi(L(x_1,..,x_n)) \xrightarrow{\rho(\xi(\Lambda))} \xi(x_1,..,x_n), \Gamma, \Delta' \vdash \rho(\xi(G))}$$

$$\text{where } \rho = mgu(\xi(\Lambda)) \text{ and } (\xi(x_i) \in \mathcal{N}/(\Delta \cup \Gamma)) \wedge (\xi(y) = y \text{ for } y \in \Delta)$$

$$(com)\ \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda_1} \Gamma, \Delta_1 \vdash G'_1 \quad \Gamma \vdash G_2 \xrightarrow{\Lambda_2} \Gamma, \Delta_2 \vdash G'_2}{\Gamma \vdash G_1|G_2 \xrightarrow{\rho(\Lambda_1 \cup \Lambda_2)} \Gamma, \Delta \vdash \rho(G'_1|G'_2)}$$

$$\text{where } \Delta_1 \cap \Delta_2 = \emptyset \text{ and } \rho = mgu(\Lambda_1, \Lambda_2)$$

$$(open)\ \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \cup \{(x,a,\boldsymbol{y})\}} \Gamma, x, \Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda} \Gamma, \Delta' \vdash \nu Z.G'}\ x \in n(\Lambda)$$

$$(hide)\ \frac{\Gamma, x \vdash G \xrightarrow{\Lambda \cup \{(x,a,\boldsymbol{y})\}} \Gamma, x, \Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda} \Gamma, \Delta' \vdash \nu x, Z.G'}\ x \notin n(\Lambda)$$

$$\text{where } Z = set(\boldsymbol{y}) \backslash (\Delta' \cup \Gamma)$$

local among some components and that now they want to allow others to communicate with them by that port. Note as we are opening name x we still have to keep bounded those names that are only shared by x (i.e. set Z). Notice that $\{x\} \cup \Delta = \Delta' \cup Z$.

Also, rule *(open)* is used for what is called an *extrusion* allowing the creation of privately shared ports. Extrusion allows to export and share bounded nodes. But once synchronization is completed, it hides away those private names that were synchronized meaning that the names are still bound, but their scope has grown. Extrusion is usually (as in Milner synchronization and π-calculus) done by using together rules *(open)* and *(close)*. But in the case of Hoare synchronization, where many edges have to synchronize in a shared node, a *(close)* rule is not very useful because it cannot be sure when to hide the private names that are extruded. It is more reasonable to use the *(com)* and at the end of the whole operation use rule *(hide)* on the corresponding names.

Rule *(hide)* deals with hiding of names. It indicates that we do not only have to hide the wanted name, but also all the names shared by synchronization only on that name (i.e. set Z). Note that there is little difference with rule *(open)*, which is the fact that for rule *(hide)* the node to be hidden (x) must not be shared by other nodes. In this case, $\Delta = \Delta' \cup Z$.

Example. In this example an instance of the ring architecture style starts with a ring configuration and at some point in its evolution is reconfigured to a star.

Figure 2*a* shows the grammar. The initial graph together with production **Brother** construct rings. Production **Star Reconfiguration** is used to reconfigure a ring into a star by creating a new node *(w)* and synchronizing using action r. The new node is distributed among components to identify it as the center of the star. Requirements $(x, r, < w >)$ and $(y, r, < w >)$ are represented

graphically imposing the pair $(r, < w >)$ on nodes x and y on the right hand side, meaning that the rewriting step is only possible if requirements are satisfied.

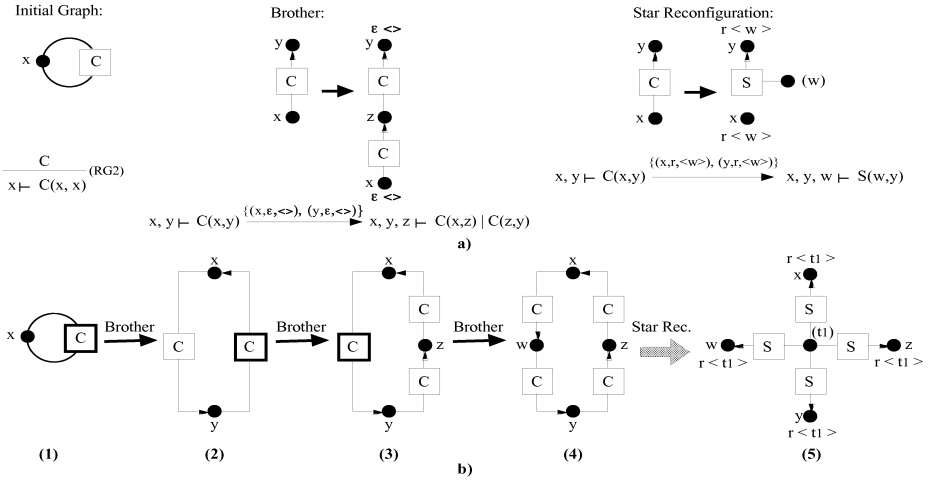


Fig. 2. Ring grammar with star reconfiguration

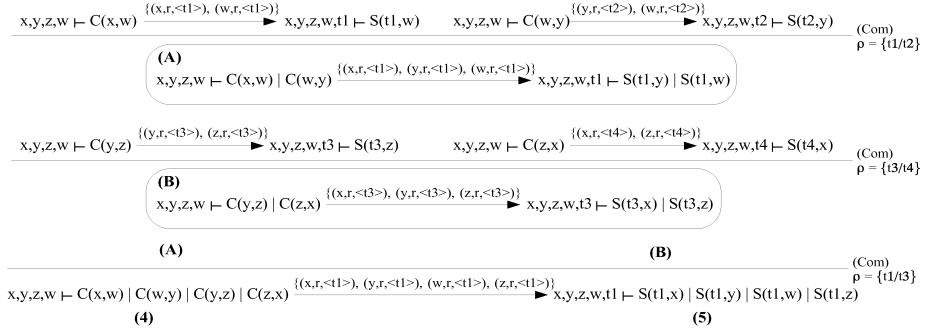


Fig. 3. Proof of transition between graphs (4) and (5) in figure 2*b*

Figure 2*b* shows a possible derivation where a ring of four components is reconfigured (thick arrow). Components with thick border indicate the component where rule **Brother** is applied. Figure 3 shows part of the proof that corresponds to the final step of the derivation in Figure 2*b*. Due to space limitations we omit the **id** productions and the application of rule *ren* for the proof.

This simple example shows how the approach can be used to specify complex reconfigurations including the combination of different styles. In [3] another example of reconfiguration can be found based on a real case of a Remote Medical Care System.

3.3 Milner Synchronization

In this section we define the *Milner Synchronization*. This synchronization mechanism only allows, in a node, to synchronize actions from two of all edges sharing

that node, and only those two edges will be rewritten as a consequence of that synchronization.

In this case, the set of actions Act is formed by two disjoint sets of actions and coactions (Act^+ and Act^-), and a special silent action (τ with $arity(\tau) = 0$). For each action $a \in Act^+$ there is a coaction $\bar{a} \in Act^-$. A requirement of the form $(x, \bar{a}, \boldsymbol{y})$ represents an output of names in $\boldsymbol{y}$ via port x with action a and a requirement of the form $(x, a, \boldsymbol{y})$ represents an input of names in $\boldsymbol{y}$ via port x with action a. A synchronization will result of the matching of an action and its corresponding coaction with the resulting unification of their shared names as it was done for Hoare Synchronization. Given that after synchronizing two requirements we are sure that the synchronization in that node is finished, the corresponding tuples are replaced by a silent action and an empty list of names.

Note that what we are defining is a general Milner synchronization where simultaneous synchronizations are allowed, so the π-calculus synchronization mechanism is a special case where only one synchronization at a time is allowed. For Milner synchronization we do not need an idle action ϵ.

Definition 7 (Milner Transition System). *Let $< G_0, \mathcal{P} >$ be a grammar. All transitions $T(\mathcal{P})$ using Milner synchronization are obtained from the transition rules in Table 3 starting from the set of productions $\mathcal{P}$ over the initial graph G_0.*

Table 3. Transition Rules for Milner Synchronization

$$(par)\ \frac{\Gamma_1 \vdash G_1 \xrightarrow{\Lambda_1} \Gamma_1, \Delta_1 \vdash G'_1 \qquad \Gamma_2 \vdash G_2 \xrightarrow{\Lambda_2} \Gamma_2, \Delta_2 \vdash G'_2}{\Gamma_1, \Gamma_2 \vdash G_1 | G_2 \xrightarrow{\Lambda_1 \cup \Lambda_2} \Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash G'_1 | G'_2}$$

$$\text{where } (\Gamma_1, \Delta_1) \cap (\Gamma_2, \Delta_2) = \emptyset$$

$$(merge)\ \frac{\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'}{\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\Lambda'} \sigma(\Gamma), \Delta' \vdash \nu Z.\rho(\sigma(G'))}$$

$$\text{where } (\sigma(x) \in (\mathcal{N}/\Delta)\ for\ x \in \Gamma)\ \wedge\ (\sigma(y) = y\ for\ y \in \Delta) \text{ and}$$
$$[(\sigma x = \sigma y,\ \Lambda(x)\downarrow,\ \Lambda(y)\downarrow, x \neq y)\ \text{ implies}$$
$$((\forall v.\sigma v = \sigma x,\ v \neq x,\ v \neq y \Rightarrow \Lambda(v)\uparrow),\ \Lambda(x) = (a, \boldsymbol{v}),\ \Lambda(y) = (\bar{a}, \boldsymbol{w}),\ a \neq \tau)]$$
$$\rho = mgu_{\Delta}\{\sigma \boldsymbol{v} = \sigma \boldsymbol{w} |\ \sigma x = \sigma y, \{(x, a, \boldsymbol{v}), (y, b, \boldsymbol{w})\} \subseteq \Lambda\}$$
$$\Lambda'(z) = \begin{cases} (\tau, <>) & if\ \sigma x = \sigma y = z,\ x \neq y,\ \Lambda(x)\downarrow,\ \Lambda(y)\downarrow \\ (\rho(\sigma(\Lambda)))(z) & otherwise \end{cases}$$
$$Z = \rho(\sigma(\Gamma \cup \Delta))/(\sigma(\Gamma) \cup \Delta')$$

$$(res)\ \frac{\Gamma, x \vdash G \xrightarrow{\Lambda} \Gamma, x, \Delta \vdash G'}{\Gamma \vdash \nu x.G \xrightarrow{\Lambda'} \Gamma, \Delta' \vdash \nu Z.G'}$$

$$\text{where } (\Lambda(x)\uparrow\ \vee\ \Lambda(x) = (\tau, <>)) \text{ and } \Lambda' = \Lambda/\{(x, \tau, <>)\} \text{ and } Z = \begin{cases} \emptyset & if\ x \in n(\Lambda) \\ \{x\} & otherwise \end{cases}$$

Rule *(par)* juxtaposes two disjoint transitions without synchronization. Also, to allow the application of a transition over a subgraph of a bigger one we

need to include the identity transitions for any graph (i.e. transitions with no requirements and that rewrite a graph in itself).

Rule *(merge)* is the responsible of identifying nodes. First, substitution σ identifies nodes (typically after the use of rule *(par)*) and then, by way of the *mgu* ρ, the synchronizations of the matched names are resolved. The conditions on the rule avoid name capture ($(\sigma(x) \in (\mathcal{N}/\Delta)\ for\ x \in \Gamma)\ \wedge\ (\sigma(y) = y\ for\ y \in \Delta)$) and assure that only two edges synchronize ($\forall v.\sigma v = \sigma x,\ v \neq x,\ v \neq y \Rightarrow \Lambda(v)\uparrow$). As result of the synchronization we have the set of requirements Λ' where the synchronizing tuples are replaced by silent actions. Set Δ' includes only those new names that are still being shared by other external nodes that have not been synchronized yet.

Rule *merge* takes care of two types of communication. The first one is when an existing node is shared and identified with some new nodes (an old node is merged with new nodes). The result of the synchronization works as a usual *(com)* rule (i.e synchronization occurs but no name is restricted, see for example rule *(Com')* in section 4.1). The second type of communication is when only new nodes are identified, which usually corresponds to what is called a *(Close)* rule. In this case when the rule is used together with rule *(res)* they cause an *extrusion*. These rules allow to export and share bounded nodes. But once synchronization is completed, rule *(merge)* hides away those private names that were synchronized (Z) meaning that the names are still bound, but their scope has grown.

Rule *(res)* takes account of four cases. The first two ($\Lambda(x)\uparrow$ or $\Lambda(x) = (\tau, <>)$ with x the name to extrude, i.e. $x \in n(\Lambda)$) correspond to what is usually called an *(Open)* rule that works in a similar way as for Hoare synchronization. They are used to export bounded nodes (in these cases $Z = \emptyset$). The other cases are for the bounding operation that in Milner Synchronization is called restriction. In the first case the rule restricts a node not participating in a synchronization ($\Lambda(x)\uparrow$) and in the second one the rule is applied over nodes where a synchronization has taken place because we are sure that it is complete ($\Lambda(x) = (\tau, <>)$). A node that can still participate in a synchronization ($\Lambda(x) \neq (\tau, <>)$) cannot be bound.

4 A Translation for π-Calculus

With the goal of studying the expressive power of the approach we are presenting in this section a translation for π-calculus using Synchronized Replacement Systems with Milner Synchronization. We first introduce the ordinary definition of π-calculus with its usual operational semantics and then present the translation.

4.1 The π-Calculus

The π-calculus [6] is a value passing process algebra. Many different versions of the π-calculus have appeared in the literature. The π-calculus we present here is synchronous, *monadic* and with guarded recursion.

Definition 8 (π-Calculus Syntax). *Let $\mathcal{N}$ be a countable set of names. The syntax of π-calculus* agents*, ranged over by $P, Q, \ldots$, are defined by the syntax:*

$$P ::= nil \mid \pi.P \mid P|P \mid P+P \mid \boldsymbol{\nu} x.\, P \mid [x{=}y]P \mid rec\, x.\, P$$

In order we have, inaction, prefix, parallel composition, non-deterministic choice, restriction, match and recursion. Prefixes, *ranged over by* π, *are defined as* $\pi ::= \bar{x}y \mid x(y)$. *They correspond to the output action and input action. The occurrences of* y *in* $x(y).P$ *and* $\boldsymbol{\nu} y.\, P$ *are bound;* free names *of agent* P *are defined as usual and we denote them with* $\mathrm{fn}(P)$. *Also, we denote with* $\mathrm{n}(P)$ *and* $\mathrm{n}(\pi)$ *the sets of (free and bound) names of agent* P *and prefix* π *respectively.*

Also, we require that any free occurrence of x *in* $rec\, x.\, P$ *must be in the scope of a prefix (guarded recursion).*

An agent P is sequential if its top operator is a prefix, the non-deterministic choice or (guarded) recursion. If σ is a name substitution, we denote with $P\sigma$ the agent P whose free names have been replaced according to substitution σ, in a capture-free way.

Definition 9 (π-Calculus Structural Congruence). *We define π-calculus agents up to a* structural congruence $\equiv$*; it is the smallest congruence that satisfies axioms in Table 4.*

Table 4. π-Calculus Structural Axioms

(alpha)	$P \equiv Q$ if P and Q are alpha equivalent with respect to bounded names
(sum)	$P{+}nil \equiv P \qquad P{+}Q \equiv Q{+}P \qquad P{+}(Q{+}R) \equiv (P{+}Q){+}R$
(par)	$P\|nil \equiv P \qquad P\|Q \equiv Q\|P \qquad P\|(Q\|R) \equiv (P\|Q)\|R$
(res)	$\boldsymbol{\nu} x.\, nil \equiv nil \qquad \boldsymbol{\nu} x.\boldsymbol{\nu} y.\, P \equiv \boldsymbol{\nu} y.\boldsymbol{\nu} x.\, P \qquad \boldsymbol{\nu} x.\, (P\|Q) \equiv P\|\boldsymbol{\nu} x.\, Q$ if $x \notin \mathrm{fn}(P)$
(match)	$[x{=}x]P \equiv P \qquad [x{=}y]nil \equiv nil$

In what follows, we will silently identify the agents that are structurally congruent. We remark that $P \equiv Q$ implies $P\sigma \equiv Q\sigma$ and $\mathrm{fn}(P) = \mathrm{fn}(Q)$; so, it is possible to define the effect of a substitution and to define the free names also for agents up to structural equivalence.

At this point, it is necessary to comment on the differences between π-calculus and synchronized rewriting that will affect the definition of the translation. For π-calculus we have an interleaving operational semantics that allows only a sequential evolution of agents. On the other side, we are using synchronized graph rewriting which is a distributed concurrent model allowing for multiple and simultaneous synchronizations and rewriting. In spite of the fact that the translation function that we are defining in section 4.2 does not allows multiple synchronization on one edge, it is still possible to have concurrent independent transition steps.

Therefore, as we want to obtain the mapping of π-calculus to the more expressive universe of graph rewriting we have to adequate the definition of the π-calculus operational semantics to a distributed context. For this we need to define a transition relation that gives some more information about τ actions. The standard operational semantics of the π-calculus is defined via labeled transitions $P \xrightarrow{\alpha} P'$ with α an action, where $P \xrightarrow{\tau} P'$ indicates that agent P goes

to P' by an internal action. This is done without the need of specifying on which port the internal action takes place and is due to the fact that as being sequential it is the only action occurring. In a distributed concurrent context we need to know where the τ action is taking place, as more than one action can happen at the same time. Then, we define a new transition relation which differs from the standard one only in the τ action. Now, we can have $P \overset{x\tau}{\longmapsto} P'$ indicating the node (x) where the synchronization occurs, and the usual $P \overset{\tau}{\longmapsto} P'$ for the cases where the τ action is taking place under the scope of a restriction. We refer to [6] for further explanations of the standard transition relation.

Definition 10 (π-Calculus Operational Semantics). *The operational semantics of the π-calculus is defined via labeled transitions $P \overset{\alpha}{\longmapsto} P'$, where P is the starting agent, P' is the target one and α is an action.*

The actions *an agent can perform are defined by the following syntax:*

$$\alpha ::= \tau \mid x\tau \mid x(z) \mid \bar{x}y \mid \bar{x}(z)$$

and are called respectively synchronization *(first two actions),* bound input*,* free output *and* bound output *actions; x and y are free names of α (*$\mathrm{fn}(\alpha)$*), whereas z is a bound name (*$\mathrm{bn}(\alpha)$*); moreover* $\mathrm{n}(\alpha) = \mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha)$*. The transitions for the* operational semantics *are defined by the rules of Table 5.*

Table 5. π-Calculus Operational Semantics

(Out) $\bar{x}y.P \overset{\bar{x}y}{\longmapsto} P$ $\qquad$ *(Inp)* $x(y).P \overset{x(y)}{\longmapsto} P$

(Sum) $\dfrac{P \overset{\alpha}{\longmapsto} P'}{P+Q \overset{\alpha}{\longmapsto} P'}$ $\qquad$ *(Par)* $\dfrac{P \overset{\alpha}{\longmapsto} P'}{P|Q \overset{\alpha}{\longmapsto} P'|Q}$ if $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$

(Com') $\dfrac{P \overset{\bar{x}y}{\longmapsto} P' \quad Q \overset{x(y)}{\longmapsto} Q'}{P|Q \overset{x\tau}{\longmapsto} P'|Q'}$ $\qquad$ *(Close')* $\dfrac{P \overset{\bar{x}(y)}{\longmapsto} P' \quad Q \overset{x(y)}{\longmapsto} Q'}{P|Q \overset{x\tau}{\longmapsto} \boldsymbol{\nu} y.\,(P'|Q')}$

(Open) $\dfrac{P \overset{\bar{x}y}{\longmapsto} P'}{\boldsymbol{\nu} y.\,P \overset{\bar{x}(y)}{\longmapsto} P'}$ if $x \neq y$ $\qquad$ *(Rec)* $\dfrac{P[rec\,x.\,P/x] \overset{\alpha}{\longmapsto} P'}{rec\,x.\,P \overset{\alpha}{\longmapsto} P'}$

(Res') $\dfrac{P \overset{\alpha}{\longmapsto} P'}{\boldsymbol{\nu} x.\,P \overset{\alpha}{\longmapsto} \boldsymbol{\nu} x.\,P'}$ if $x \notin \mathrm{n}(\alpha)$ $\qquad$ $\dfrac{P \overset{x\tau}{\longmapsto} P'}{\boldsymbol{\nu} x.\,P \overset{\tau}{\longmapsto} \boldsymbol{\nu} x.\,P'}$

(Match) $\dfrac{P \overset{\alpha}{\longmapsto} P'}{[x{=}x]P \overset{\alpha}{\longmapsto} P'}$ $\qquad$ *(Cong)* $\dfrac{P \equiv P' \quad P \overset{\alpha}{\longmapsto} Q \quad Q \equiv Q'}{P' \overset{\alpha}{\longmapsto} Q'}$

The differences of the standard operational semantics with the rules of Table 5 are in rules *(Com')*, *(Close')* and *(Res')* for the τ action. In rule *(Res')* we add a second case to the original one for restricting a node where a synchronization takes place. The prefix $x(z)$ for bound input means *input some name along link named x and call it y.* A free output $\bar{x}y$ means *output the name y along the link named x.* A bound output $\bar{x}(z)$ corresponds to the emission of a private name of an agent to the environment: in this way, the channel becomes public and can be used for further communications between the agent and the environment.

Process $[x{=}y]P$ behaves like P if x and y are the same name, it behaves like the inactive process otherwise. As we already mentioned, rule *(Open')* shares a private name with the environment and together with rule *(Close')* causes an extrusion. Rule *(Rec)* describes the transition for a recursive process.

4.2 Translation

Now we present a translation function of π-calculus to synchronized replacement Systems and state the correspondence theorems. Proofs of can be found in [4].

Definition 11 (Translation Function for π-Calculus). *Let $\mathcal{N}$ be a fixed infinite set of names. We define a translation function $[\![-]\!]_\Gamma$ for π-calculus agents. The translation is defined with respect to a set of names ($\Gamma \subset \mathcal{N}$) in Table 6.*

Table 6. Translation Function for π-calculus

1. $[\![nil]\!]_\Gamma = \Gamma \vdash nil$ **2.** $[\![\pi.P]\!]_\Gamma = \Gamma \vdash L_{\pi.P}(ord(\mathrm{fn}(\pi.P)))$ if $\mathrm{fn}(\pi.P) \subseteq \Gamma$

3. $[\![P{+}Q]\!]_\Gamma = \Gamma \vdash L_{P+Q}(ord(\mathrm{fn}(P{+}Q)))$ *if* $\mathrm{fn}(P{+}Q) \subseteq \Gamma$ *and* $P, Q \not\equiv nil$
$[\![P{+}Q]\!]_\Gamma = [\![Q{+}P]\!]_\Gamma = [\![P]\!]_\Gamma$ *if* $\mathrm{fn}(P) \subseteq \Gamma$ *and* $Q \equiv nil$

4. $[\![rec\, x.\, P]\!]_\Gamma = \Gamma \vdash L_{rec\, x.\, P}(ord(\mathrm{fn}(rec\, x.\, P)))$ **5.** $[\![[x{=}y]P]\!]_\Gamma = [\![nil]\!]_\Gamma$ *if* $x \neq y$
$[\![[x{=}x]P]\!]_\Gamma = [\![P]\!]_\Gamma$

6. $\dfrac{[\![P]\!]_\Gamma = \Gamma \vdash G_P \quad [\![Q]\!]_\Gamma = \Gamma \vdash G_Q}{[\![P|Q]\!]_\Gamma = \Gamma \vdash G_P | G_Q}$ **7.** $\dfrac{[\![P]\!]_{\Gamma,x} = \Gamma, x \vdash G}{[\![\boldsymbol{\nu} x.\, P]\!]_\Gamma = \Gamma \vdash \nu x.G}$

Productions

8. $\dfrac{P \overset{\bar{x}y}{\longmapsto} Q \quad P \text{ sequential}}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(out,x,<y>)\}} [\![Q]\!]_{\mathrm{fn}(P)}}$ **9.** $\dfrac{P \overset{x\tau}{\longmapsto} Q \quad P \text{ sequential}}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(\tau,x,<>)\}} [\![Q]\!]_{\mathrm{fn}(P)}}$

10. $\dfrac{P \overset{\tau}{\longmapsto} Q \quad P \text{ sequential}}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\emptyset} [\![Q]\!]_{\mathrm{fn}(P)}}$ **11.** $\dfrac{P \overset{\bar{x}(y)}{\longmapsto} Q \quad P \text{ sequential}}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(out,x,<y>)\}} [\![Q]\!]_{\mathrm{fn}(P)\cup\{y\}}}$

12. $\dfrac{P \overset{x(y)}{\longmapsto} Q \quad P \text{ sequential}}{[\![P]\!]_{\mathrm{fn}(P)} \xrightarrow{\{(in,x,<y>)\}} [\![Q]\!]_{\mathrm{fn}(P)\cup\{y\}}}$

The edges that are created by the translation have labels that correspond to the uppermost level of sequential subterms of the agent to be translated. More specifically, each edge label represents a sequential subterm up to structural axioms, so two labels corresponding to equivalent agents are considered the same label. Function *ord* returns an ordered sequence of the agent free names that represents the attachment nodes of the corresponding edge. A transition in the π-calculus will be represented as a transition of the corresponding judgement (up to alpha conversion of bounded names). Then, productions are generated based on these sequential agents with the corresponding label as the left hand side of the production and the target agent as the right hand side. Actions are translated as requirements on the transitions. For π-calculus we have one action

for inputs (*in*) and one coaction for outputs (*out*). Then, the evolution of the agent is modeled using the Milner transition system introduced in table 3. Note that for the cases with action τ, the translation produces a production with τ and an empty list of names (9) or a production with no requirements (10). Intuitively, this last case corresponds to a judgement proof where restriction rule *(res)* was applied. This correspondence is formally proved in theorem 4. An example of a sequential agent that produces a τ transition can be $\bar{\alpha}|\alpha + \beta$.

Theorem 2. *For every π-calculus agent a finite set of productions is generated by the translation function $[\![-]\!]_\Gamma$.*

Theorem 3 (Correspondence of π-Calculus Agents and Judgements). *There is a bijective correspondence of π-calculus agents (up to structural axioms) and well-formed syntactic judgements (up to structural axioms and isolated nodes) with respect to the translation function $[\![-]\!]_\Gamma$.*

For the next theorem we define a *basic judgement transition* as a transition over a graph using exactly one production or the synchronization of two productions. These mean that either one agent only makes a transition (without synchronization) or only two agents make a transition by synchronization.

As we already mentioned, we are mapping π-calculus that has a sequential operational semantics (one transition at a time) to a distributed concurrent context. So it is clear that there are transitions for judgements (the concurrent ones) that cannot be obtained in π-calculus. Then, the following theorem states that a transition step in π-calculus is done, if and only if, there is a basic judgement transition between the corresponding translations. Basic transitions correspond to sequential steps of π-calculus. Note that the correspondence theorem is proved for the standard operational semantics of π-calculus. Also, we have to say that the theorem below is sufficient to prove the semantic correspondence given that theorem 3 already proved the bijective correspondence of the translation from agents to judgements (i.e. graphs). In this way, we are sure that any graph resulting from a transition has a corresponding agent.

Definition 12 (Basic Transition). *Transition $\Gamma \vdash G \xrightarrow{\Lambda} \Gamma, \Delta \vdash G'$ is* basic *if*

- *it is a production (Λ contains at most one requirement by definition) or*
- *set Λ contains one requirement and if the proof uses rule* merge *it must be a synchronization or*
- *set Λ is empty, the proof is as before with a synchronization and there is an application of rule* (res) *over the synchronizing name (it is the only rule that can remove a τ).*

Theorem 4 (Semantic Correspondence). *For any π-calculus agents P and Q, $P \xrightarrow{\alpha} Q$, if and only if, there is a basic transition $[\![P]\!]_\Gamma \xrightarrow{\{[\![\alpha]\!]_\Gamma\}} [\![Q]\!]_\Gamma$, with $\mathrm{fn}(P),\ \mathrm{fn}(Q),\ \mathrm{fn}(\alpha) \subseteq \Gamma$.*

5 Conclusions and Future Work

This paper presents a new graphical mobile calculus based on synchronized hyperedge replacement with name mobility. We formalize the approach using syntactic judgements and present the transition system for Hoare and Milner synchronizations. It is worth notice that to "implement" other synchronization algebras it is only needed to change the corresponding transition system. Also, for studying the expressive power of the approach we present a translation for π-calculus using synchronized replacement with Milner synchronization.

It is clear from the results that our calculus is more expressive (and general) than π-calculus allowing us to have higher level primitives. Although, this implies an increment of the complexity of the implementation which have to be evaluated as a tradeoff of the practical problem of implementing the multiple synchronization mechanisms and the specific applications of interest.

We can also mention that [5] presents a bisimilarity for synchronized graph rewriting with name mobility (based on the work of [3]) proving it to be a congruence. Also they introduce a so-called *format* which is a syntactic condition on productions ensuring that bisimilarity is a congruence. This last result is original not only for graph rewriting, but also for mobility in general.

It is our intention to continue the study of the expressive power of this model and to generate a prototype implementing these ideas. Also we want to investigate techniques to analyze system properties over the graph derivations such as, invariant checking, reachability and static analysis, and modular reasoning.

References

1. Drewes, F., Kreowski, H-J., Hable, A.: Hyperedge Replacement Graph Grammars. Chapter 2, In [8]. 1997.
2. Ehrig, H., Kreowski, H.-J., Montanari, U. and Rozenberg, G., Editors: Handbook of Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution, volume 3 , World Scientific, 1999.
3. Hirsch, D., Inverardi, P., Montanari, U.: Reconfiguration of Software Architecture Styles with Name Mobility. In Proceedings of 4th international Conference, Coordination 2000, LNCS, volume 1906, 2000.
4. Hirsch, D., Montanari, U.: Synchronized Hyperedge Replacement with Name Mobility. Technical Report TR-01-001, Department of Computer Science, Universidad de Buenos Aires, 2001. http://www.dc.uba.ar/people/proyinv/tr.html
5. König, B., Montanari, U.: Observational Equivalence for Synchronized Graph Rewriting with Mobility. Submitted for publication. 2001.
6. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, 1999.
7. Montanari, U., Pistore, M. and Rossi, F.: Modeling Concurrent, Mobile and Coordinated Systems via Graph Transformations. In [2]. 1999.
8. Rozenberg, G., editor: Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, volume I. World Scientific, 1997.
9. Sangiorgi, D.: π-calculus, Internal Mobility and Agent-passing Calculi. Theoretical Computer Science 167(2), 1996.
10. Victor B.: The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. PhD Thesis, Uppsala University, Dept. of Computer Science, June 1998.

Dynamic Input/Output Automata: A Formal Model for Dynamic Systems
(Extended Abstract)

Paul C. Attie[1,2] and Nancy A. Lynch[2]

[1] College of Computer Science, Northeastern University, Cullinane Hall, 360 Huntington Avenue, Boston, Massachusetts 02115.
attie@ccs.neu.edu
[2] MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA.
lynch@theory.lcs.mit.edu

Abstract. We present a mathematical state-machine model, the *Dynamic I/O Automaton (DIOA) model*, for defining and analyzing *dynamic systems* of interacting components. The systems we consider are dynamic in two senses: (1) components can be created and destroyed as computation proceeds, and (2) the events in which the components may participate may change. The new model admits a notion of *external system behavior*, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior, and a notion of *simulation* from one dynamic system to another, which can be used to prove that one system implements the other.
The DIOA model was defined to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. It can also be used for other forms of dynamic systems, such as systems described by means of object-oriented programs, and systems containing services with changing access permissions.

1 Introduction

Many modern distributed systems are *dynamic*: they involve changing sets of components, which get created and destroyed as computation proceeds, and changing capabilities for existing components. For example, programs written in object-oriented languages such as Java involve objects that create new objects as needed, and create new references to existing objects. Mobile agent systems involve agents that create and destroy other agents, travel to different network locations, and transfer communication capabilities.

To describe and analyze such distributed systems rigorously, one needs an appropriate *mathematical foundation*: a state-machine-based framework that allows modeling of individual components and their interactions and changes. The framework should admit standard modeling methods such as parallel composition and levels of abstraction, and standard proof methods such as invariants

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 137–151, 2001.
© Springer-Verlag Berlin Heidelberg 2001

and simulation relations. At the same time, the framework should be simple enough to use as a basis for distributed algorithm analysis.

Static mathematical models like I/O automata [7] could be used for this purpose, with the addition of some extra structure (special Boolean flags) for modeling dynamic aspects. For example, in [8], dynamically-created transactions were modeled as if they existed all along, but were "awakened" upon execution of special *create* actions. However, dynamic behavior has by now become so prevalent that it deserves to be modeled directly. The main challenge is to identify a small, simple set of constructs that can be used as a basis for describing most interesting dynamic systems.

In this paper, we present our proposal for such a model: the *Dynamic I/O Automaton (DIOA) model*. Our basic idea is to extend the I/O automaton model with special *create* actions, and combine such extended automata into global *configurations*. The DIOA model admits a notion of external system behavior, based on sets of traces. It also features a *parallel composition* operator for dynamic systems, which respects external behavior and satisfies standard execution projection and pasting results, and a notion of *simulation relation* from one dynamic system X to another dynamic system Y, which can be used to prove that X implements Y.

We defined the DIOA model initially to support the analysis of *mobile agent systems*, in a joint project with researchers at Nippon Telephone and Telegraph. Creation and destruction of agents are modeled directly within the DIOA model. Other important agent concepts such as changing locations and capabilities are described in terms of changing signatures, using additional structure. Our preliminary work on modeling and analyzing agent systems appeared in last year's NASA workshop on formal methods for agent systems [1]. We are currently considering the use of DIOA to model and analyze object-oriented programs; here, creation of new objects is modeled directly, while addition of references is modeled as a signature change.

Related work: Most approaches to the modeling of dynamic systems are based on a process algebra, in particular, the π-calculus [9] or one of its variants. Such approaches [4,5,10] model dynamic aspects by introducing channels and/or locations as basic notions. Our model is more primitive than these approaches, for example, it does not include channels and their transmission as basic notions. Our approach is also different in that it is primarily a (set-theoretic) mathematical model, rather than a formal language and calculus. We expect that notions such as channel and location will be built upon the basic model using additional layers (as we do for modeling agent mobility in terms of signature change). Also, we ignore issues (e.g., syntax) that are important when designing a programming language (the "precondition-effect" notation in which we present an example is informal, and is not part of our model). Another difference with process-algebraic approaches is that we use a simulation notion for refinement, rather than bisimulation. This allows us more latitude in refinement, as our example will demonstrate. Finally, our model has a well-defined notion of projection onto a subsystem. This is a crucial pre-requisite for compositional reasoning, and is usually missing from process-algebraic approaches.

The paper is organized as follows. Section 2 presents the DIOA model. Section 3 presents execution projection and pasting results, which provide the basis for compositional reasoning in our model. Section 4 proposes an appropriate notion of forward simulation for DIOA. Section 5 discusses how mobility and locations can be modeled in DIOA. Section 6 presents an example: an agent whose purpose is to traverse a set of databases in search of a satisfactory airline flight, and to purchase such a flight if it finds it. Section 7 discusses further research and concludes. Proofs are provided in the full version [2], which is available on-line.

2 The Dynamic I/O Automaton Model

To express dynamic aspects, DIOA augments the I/O automaton model with:

1. Variable signatures: The signature of an automaton is a function of its state, and so can change as the automaton makes state transitions. In particular, an automaton "dies" by changing its signature to the empty set, after which it is incapable of performing any action. We call this new class of automata *signature I/O automata*, henceforth referred to simply as "automata," or abbreviated as SIOA.
2. Create actions: An automaton A can "create" a new automaton B by executing a create action
3. Two-level semantics: Due to the introduction of create actions, the semantics of an automaton is no longer accurately given by its transition relation. The effect of create actions must also be considered. Thus, the semantics is given by a second class of automata, called *configuration automata*. Each state of a configuration automaton consists of the collection of signature I/O automata that are currently "awake," together with the current local state of each one.

2.1 Signature I/O Automata

We assume the existence of a set $\mathcal{A}$ of unique SIOA identifiers, an underlying universal set $Auts$ of SIOA, and a mapping $aut : \mathcal{A} \mapsto Auts$. $aut(A)$ is the SIOA with identifier A. We use "the automaton A" to mean "the SIOA with identifier A". We use the letters A, B, possibly subscripted or primed, for SIOA identifiers.

In our model, each automaton A has a *universal signature* $usig(A)$. The actions that A may execute (in any of its states) are drawn from $usig(A)$. In a particular state s, the executable actions are drawn from a fixed (but varying with s) sub-signature of $usig(A)$, denoted by $sig(A)(s)$, and called the *state signature*. Thus, the "current" signature of A is a function of its current state that is always constrained to be a sub-signature of A's universal signature.

As in the I/O atomaton model, the actions of a signature (either universal or state) are partitioned into (sets of) input, output, and internal actions: $usig(A) = \langle uin(A), uout(A), uint(A) \rangle$. Additionally, the output actions are partitioned into regular outputs and create outputs: $uout(A) = \langle uoutregular(A), ucreate(A) \rangle$. Likewise, $sig(A)(s) = \langle in(A)(s), out(A)(s), int(A)(s) \rangle$, and $out(A)(s) = \langle outregular(A)(s), create(A)(s) \rangle$.

For any signature component, the $\hat{}$ operator yields the union of sets of actions within the signature, e.g., $\hat{out}(A)(s) = outregular(A)(s) \cup create(A)(s)$, and $\hat{sig}(A)(s) = in(A)(s) \cup outregular(A)(s) \cup create(A)(s) \cup int(A)(s)$.

A create action a has a single attribute: $target(a)$, the identifier of the automaton that is to be created.

Definition 1 (Signature I/O Automaton). *A* signature I/O automaton $aut(A)$ *consists of the following components and constraints on those components:*

- *A fixed universal signature* $usig(A)$ *as discussed above.*
- *A set* $states(A)$ *of states.*
- *A nonempty set* $start(A) \subseteq states(A)$ *of start states.*
- *A mapping* $sig(A) : states(A) \mapsto 2^{uin(A)} \times \{2^{uoutregular(A)} \times 2^{ucreate(A)}\} \times 2^{uint(A)}$.
- *A transition relation* $steps(A) \subseteq states(A) \times usig(A) \times states(A)$.
- *The following constraints:*
  1. $\forall (s, a, s') \in steps(A) : a \in \hat{sig}(A)(s)$.
  2. $\forall s, \forall a \in in(A)(s), \exists s' : (s, a, s') \in steps(A)$
  3. $\hat{sig}(A)(s) \neq \emptyset$ *for any start state* s.

Constraint 1 requires that any executed action be in the signature of the start state. Constraint 2 is the input enabling requirement of I/O automata. Constraint 3 requires that start states have a nonempty signature, since otherwise, the newly created automaton will be unable to execute any action. Thus, this is no restriction in practice, and its use simplifies our definitions.

If $(s, a, s') \in steps(A)$, we also write $s \xrightarrow{a}_A s'$. For sake of brevity, we write $states(A)$ instead of $states(aut(A))$, i.e., the components of an automaton are identified by applying the appropriate selector function to the automaton identifier, rather than the automaton itself. In the sequel, we shall sometimes write a create action as $\mathsf{create}(A, B)$, where A is the identifier of the automaton executing $\mathsf{create}(A, B)$, and B is the target automaton identifier. This is a notational convention only, and is not part of our model.

2.2 Configuration Automata

Suppose $\mathsf{create}(A, B)$ is an action of A. As with any action, execution of $\mathsf{create}(A, B)$ will, in general, cause a change in the state of A. However, we also want the execution of $\mathsf{create}(A, B)$ to have the effect of creating the SIOA B. To model this, we must keep track of the set of "alive" SIOA, i.e., those that have been created but not destroyed (we consider the automata that are initially present to be "created at time zero"). Thus, we require a transition relation over sets of SIOA. We also need to keep track of the current global state, i.e., the tuple of local states of every SIOA that is alive. Thus, we replace the notion of global state with the notion of "configuration," and use a transition relation over configurations.

Definition 2 (Simple configuration, Compatible simple configuration). *A* simple configuration *is a finite set* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *where* A_i *is a signature I/O automaton identifier,* $s_i \in states(A_i)$*, for* $1 \leq i \leq n$*, and* $A_i \neq A_j$ *for* $1 \leq i, j \leq n, i \neq j$*.*

A simple configuration $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ *is* compatible *iff, for all* $1 \leq i, j \leq n, i \neq j$*:*

1. $\hat{sig}(A_i)(s_i) \cap int(A_j)(s_j) = \emptyset$*,* $\hat{out}(A_i)(s_i) \cap \hat{out}(A_j)(s_j) = \emptyset$*, and*
2. $create(A_i)(s_i) \cap \hat{sig}(A_j)(s_j) = \emptyset$*.*

Thus, in addition to the usual I/O automaton compatibility conditions [7], we require that a create action of one SIOA cannot be in the signature of another.

If $n = 0$, then the configuration is *empty*. Let $C = \{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ be a compatible simple configuration. Then we define $auts(C) = \{A_1, \ldots, A_n\}$, $outregular(C) = \bigcup_{1 \leq i \leq n} outregular(A_i)(s_i)$, $create(C) = \bigcup_{1 \leq i \leq n} create(A_i)(s_i)$, $in(C) = \bigcup_{1 \leq i \leq n} in(A_i)(s_i) - outregular(C)$, $int(C) = \bigcup_{1 \leq i \leq n} int(A_i)(s_i)$.

Definition 3 (Transitions of a simple configuration). *The* transitions *that a compatible simple configuration* $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\}$ $(n > 0)$ *can execute are as follows:*

1. *non-*create *action*
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$
   $\{\langle A_1, s'_1\rangle, \ldots, \langle A_n, s'_n\rangle\} - \{\langle A_j, s'_j\rangle : 1 \leq j \leq n \text{ and } \hat{sig}(A_j)(s_j) = \emptyset\}$
   if
   $a \in \hat{sig}(A_1)(s_1) \cup \ldots \cup \hat{sig}(A_n)(s_n)$,
   $a \notin create(A_1)(s_1) \cup \ldots \cup create(A_n)(s_n)$*, and*
   for all $1 \leq i \leq n$ *: if* $a \in \hat{sig}(A_i)(s_i)$ *then* $s_i \xrightarrow{a}_{A_i} s'_i$*, otherwise* $s'_i = s_i$*.*
   Transitions not arising from a create action enforce synchronization by matching action names, as in the basic I/O automaton model. Also, all involved automata may change their current signature, and automata whose new signature is empty are destroyed.
2. create *actions*
   a) create *action whose target does not exist a priori*
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s_i\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s'_i\rangle, \ldots, \langle A_n, s_n\rangle, \langle B, t\rangle\} - \{\langle A_i, s'_i\rangle : \hat{sig}(A_i)(s'_i) = \emptyset\}$
   if
   $1 \leq i \leq n$*,* $a \in create(A_i)(s_i)$*,* $s_i \xrightarrow{a}_{A_i} s'_i$*,* $target(a) = B$*,*
   $B \notin \{A_1, \ldots, A_n\}$*, and* $t \in start(B)$*.*
   Execution of a in a simple configuration where its target B is not present results in the creation of B, which initially can be in any of its start states t. $\langle B, t\rangle$ is added to the current configuration. The automaton A_i executing a changes state and signature according to its transition relation and signature mapping, and all other automata remain in the same state. If A_i's new signature is empty, then A_i is destroyed.
   b) create *action whose target automaton already exists*
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s_i\rangle, \ldots, \langle A_n, s_n\rangle\} \xrightarrow{a}$
   $\{\langle A_1, s_1\rangle, \ldots, \langle A_i, s'_i\rangle, \ldots, \langle A_n, s_n\rangle\} - \{\langle A_i, s'_i\rangle : \hat{sig}(A_i)(s'_i) = \emptyset\}$

if
$1 \leq i \leq n$, $a \in create(A_i)(s_i)$, $s_i \xrightarrow{a}_{A_i} s_i'$, $target(a) \in \{A_1, \ldots, A_n\}$.
Execution of a in a simple configuration where its target is already present results only in a state and signature change to the automaton A_i executing a. All other automata remain in the same state. If A_i's new signature is empty, then A_i is destroyed.

If a simple configuration is empty, or is not compatible, then it cannot execute any transitions.

If C and D are simple configurations and $\pi = a_1, \ldots, a_n$ is a finite sequence of $n \geq 1$ actions, then define $C \xrightarrow{\pi} D$ iff there exist simple configurations $C_0, \ldots, C_n$ such that $C = C_0 \xrightarrow{a_1} C_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} C_{n-1} \xrightarrow{a_n} C_n = D$.

In anticipation of composition, we define.

Definition 4 (Configuration).

1. *A simple configuration is a configuration*
2. *If $C_1, \ldots, C_n$ are configurations ($n > 0$), then so is $\langle C_1, \ldots, C_n \rangle$*
3. *The only configurations are those generated by the above two rules*

We extend *auts* to configurations by defining $auts(\langle C_1, \ldots, C_n \rangle) = auts(C_1) \cup \ldots \cup auts(C_n)$ for a configuration $\langle C_1, \ldots, C_n \rangle$.

The entire behavior that a given configuration is capable of is captured by the notion of *configuration automaton*.

Definition 5 (Configuration automaton). *A configuration automaton X is a state-machine with four components.*

1. *a nonempty set of start configurations, $start(X)$*
2. *a set of configurations, $states(X) \supseteq start(X)$*
3. *a signature mapping $sig(X)$, where for each $C \in states(X)$,*
   a) $sig(X)(C) = \langle in(X)(C), out(X)(C), int(X)(C) \rangle$
   b) $out(X)(C) = \langle outregular(X)(C), outcreate(X)(C) \rangle$
   c) $int(X)(C) = \langle intregular(X)(C), intcreate(X)(C) \rangle$
   d) *$in(X)(C)$, $outregular(X)(C)$, $outcreate(X)(C)$, $intregular(X)(C)$, and $intcreate(X)(C)$ are sets of actions.*
4. *a transition relation, $steps(X) = \{(C, a, D) \mid C, D \in states(X)$ and $a \in \hat{sig}(X)(C)\}$*

We usually use "configuration" rather than "state" when referring to states of a configuration automaton. Definition 5 allows an arbitrary transition relation between the configurations of a configuration automaton. However, these configurations are finite nested tuples, with the basic elements being SIOA. The SIOA transitions totally determine the transitions that a given configuration can execute. Hence, we introduce *proper configuration automata* (rules CA1–CA4 below), which respect the transition behavior of configurations.

Definition 6 (Mutually compatible configurations). *Let X, Y be configuration automata. Let $C \in states(X)$, $D \in states(Y)$. Then C and D are* mutually compatible *iff*

1. $auts(C) \cap auts(D) = \emptyset$,
2. $\hat{sig}(X)(C) \cap \hat{int}(Y)(D) = \emptyset$, $\hat{int}(X)(C) \cap \hat{sig}(Y)(D) = \emptyset$, $\hat{out}(X)(C) \cap \hat{out}(Y)(D) = \emptyset$, *and*
3. $outcreate(X)(C) \cap \hat{sig}(Y)(D) = \emptyset$, $\hat{sig}(X)(C) \cap outcreate(Y)(D) = \emptyset$.

Definition 7 (Compatible configuration). *Let C be a configuration. If C is simple, then C is compatible (or not) according to Definition 2. If $C = \langle C_1, \ldots, C_n \rangle$, then C is compatible iff (1) each C_i is compatible, and (2) each pair in $\{C_1, \ldots, C_n\}$ are mutually compatible.*

Definition 8 (Configuration transitions). *The* transitions *that a compatible configuration C can execute are as follows:*

1. *If C is simple, then the transitions are those given by Definition 3*
2. *If $C = \langle C_1, \ldots, C_n \rangle$, then $\langle C_1, \ldots, C_n \rangle \xrightarrow{a} \langle D_1, \ldots, D_n \rangle$ iff*
   a) $a \in \hat{sig}(C_1) \cup \ldots \cup \hat{sig}(C_n)$
   b) *for* $1 \leq i \leq n$: *if* $a \in \hat{sig}(C_i)$ *then* $C_i \xrightarrow{a} D_i$, *otherwise* $C_i = D_i$.

Definition 9 (Closure). *Let $\mathcal{C}$ be a set of compatible configurations $\mathcal{C}$. $X = closure(\mathcal{C})$ is the state-machine given by:*

1. $start(X) = \mathcal{C}$
2. $states(X) = \{D \mid \exists C \in \mathcal{C}, \exists \pi : C \xrightarrow{\pi} D\}$
3. $steps(X) = \{(C, a, D) \mid C \xrightarrow{a} D \text{ and } C, D \in states(X)\}$
4. $sig(X)$, *where for each* $C \in states(X)$, $sig(X)(C)$ *is given by:*
   a) $outregular(X)(C) = outregular(C)$
   b) $outcreate(X)(C) = create(C)$
   c) $in(X)(C) = in(C)$
   d) $intregular(X)(C) = int(C)$
   e) $intcreate(X)(C) = \emptyset$

Rule CA1: Let X be as in Definition 9. If every configuration of X is compatible, then X is a proper configuration automaton.

config($\mathcal{C}$) is the automaton induced by all the configurations reachable from some configuration in $\mathcal{C}$, and the transitions between them.

Definition 10 (Composition of proper configuration automata). *Let $X_1, \ldots, X_n$, be proper configuration automata. Then $X = X_1 \parallel \cdots \parallel X_n$ is the state-machine given by:*

1. $start(X) = start(X_1) \times \cdots \times start(X_n)$
2. $states(X) = states(X_1) \times \cdots \times states(X_n)$
3. $steps(X)$ *is the set of all* $(\langle C_1, \ldots, C_n \rangle, a, \langle D_1, \ldots, D_n \rangle)$ *such that*
   a) $a \in \hat{sig}(X_1)(C_1) \cup \ldots \cup \hat{sig}(X_n)(C_n)$, *and*
   b) *if* $a \in \hat{sig}(X_i)(C_i)$, *then* $C_i \xrightarrow{a}_{X_i} D_i$, *otherwise* $C_i = D_i$
4. $sig(X)$, *where for each* $C = \langle C_1, \ldots, C_n \rangle \in states(X)$, $sig(X)(C)$ *is given by:*

a) $outregular(X)(C) = outregular(X_1)(C_1) \cup \ldots \cup outregular(X_n)(C_n)$
b) $outcreate(X)(C) = outcreate(X_1)(C_1) \cup \ldots \cup outcreate(X_n)(C_n)$
c) $in(X)(C) = (in(X_1)(C_1) \cup \ldots \cup in(X_1)(C_1)) - outregular(X)(C)$
d) $intregular(X)(C) = intregular(X_1)(C_1) \cup \ldots \cup intregular(X_n)(C_n)$
e) $intcreate(X)(C) = intcreate(X_1)(C_1) \cup \ldots \cup intcreate(X_n)(C_n)$

Rule CA2: Let X be as in Definition 10. If every configuration of X is compatible, then X is a proper configuration automaton.

Definition 11 (Action hiding). *Let X be a proper configuration automaton and Σ a set of actions. Then $X \setminus \Sigma$ is the state-machine given by:*

1. $start(X \setminus \Sigma) = start(X)$
2. $states(X \setminus \Sigma) = states(X)$
3. $steps(X \setminus \Sigma) = steps(X)$
4. $sig(X \setminus \Sigma)$, *where for each* $C \in states(X \setminus \Sigma)$, $sig(X \setminus \Sigma)(C)$ *is given by:*
   a) $outregular(X \setminus \Sigma)(C) = outregular(X)(C) - \Sigma$
   b) $outcreate(X \setminus \Sigma)(C) = outcreate(X)(C) - \Sigma$
   c) $in(X \setminus \Sigma)(C) = in(X)(C)$
   d) $intregular(X \setminus \Sigma)(X)C = intregular(X)(C) \cup (outregular(X)(C) \cap \Sigma)$
   e) $intcreate(X \setminus \Sigma)(X)C = intcreate(X)(C) \cup (outcreate(X)(C) \cap \Sigma)$

Rule CA3: If X is a proper configuration automaton, then so is $X \setminus \Sigma$.

The automata generated by rules CA1, CA2 are called closure automata, composed automata, respectively.

Rule CA4: The only configuration automata are those that are generated by rules CA1–CA3.

Definition 12 (Execution, trace). *An execution fragment α of a configuration automaton X is a (finite or infinite) sequence $C_0 a_1 C_1 a_2 \ldots$ of alternating configurations and actions such that $(C_{i-1}, a_i, C_i) \in steps(X)$ for each triple (C_{i-1}, a_i, C_i) occurring in α. Also, α ends in a configuration if it is finite. An* execution *of X is an execution fragment of X whose first configuration is in $start(X)$. $execs(X)$ denotes the set of executions of configuration automaton X.*

Given an execution fragment $\alpha = C_0 a_1 C_1 a_2 \ldots$, the trace of α (denoted $trace(\alpha)$) is the sequence that results from

1. *replacing each C_i by its external signature $ext(X)(C_i)$, and then*
2. *removing all a_i such that $a_i \notin \hat{ext}(X)(C_{i-1})$, i.e., a_i is an internal action of C_{i-1}, and then*
3. *replacing every finite, maximal sequence of identical external signatures by a single instance.*

$traces(X)$, the set of traces of a configuration automaton X, is the set $\{\beta \mid \exists \alpha \in execs(X) : \beta = trace(\alpha)\}$.

We write $C \xrightarrow{\alpha}_X C'$ iff there exists an execution fragment α (with $|\alpha| \geq 1$) of X starting in C and ending in C'. When α contains a single action a (and so $(C, a, C') \in steps(X)$) we write $C \xrightarrow{a}_X C'$.

2.3 Clone-Freedom

Our semantics allows the creation of several SIOA with the same identifier, provided they are "contained" in different closure automata (which could then be composed); we preclude this within the same closure automaton because the SIOA would not be distinguishable from our point of view. We also find it desirable that SIOA in different closure automata also have different identifiers, i.e., that identifiers are really unique (which is why we introduced them in the first place). Thus, we make the following assumption.

Definition 13 (Clone-freedom assumption). *For any proper configuration automaton X, and any reachable configuration C of X, there is no action $a \in outcreate(X)(C) \cup intcreate(X)(C)$ such that $target(a) \in auts(C)$ and $\exists C' : C \xrightarrow{a} C'$.*

This assumption does not preclude reasoning about situations in which an SIOA A_1 cannot be distinguished from another SIOA A_2 *by the other SIOA in the system.* This could occur, e.g., due to a malicious host which "replicates" agents that visit it. We distinguish between such replicas at the level of reasoning by assigning unique identifiers to each. These identifiers are not available to the other SIOA in the system, which remain unable to tell A_1 and A_2 apart (e.g., in the sense of the "knowledge" [6] about A_1, A_2 that they possess).

3 Compositional Reasoning

To confirm that our model provides a reasonable notion of concurrent composition, which has expected properties, and to enable compositional reasoning, we establish execution "projection" and "pasting" results for compositions.

Definition 14 (Execution projection). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in states(X)$ and $\forall j > 0, a_j \in \hat{sig}(X)(C_{j-1})$. Then $\alpha \lceil X_i$ $(1 \leq i \leq n)$ is the sequence resulting from:*

1. *replacing each C_j by its i'th component $C_{j,i}$, and then*
2. *removing all $a_j C_{j,i}$ such that $a_j \notin \hat{sig}(X_i)(C_{j-1,i})$.*

Our execution projection results states that the projection of an execution (of a composed configuration automaton $X = X_1 \parallel \cdots \parallel X_n$) onto a component X_i, is an execution of X_i.

Theorem 1 (Execution projection). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. If $\alpha \in execs(X)$ then $\alpha \lceil X_i \in execs(X_i)$.*

Our execution pasting result requires that a candidate execution α of a composed automaton $X = X_1 \parallel \cdots \parallel X_n$ must project onto an actual execution of every component X_i, and also that every action of α not involving X_i does not change the configuration of X_i. In this case, α will be an actual execution of X.

Theorem 2 (Execution pasting). *Let $X = X_1 \parallel \cdots \parallel X_n$ be a proper configuration automaton. Let α be a sequence $C_0 a_1 C_1 a_2 C_2 \ldots C_{j-1} a_j C_j \ldots$ where $\forall j \geq 0, C_j = \langle C_{j,1}, \ldots, C_{j,n} \rangle \in states(X)$ and $\forall j > 0, a_j \in \hat{sig}(X)(C_{j-1})$. Furthermore, suppose that*

1. *for all $1 \leq i \leq n : \alpha \lceil X_i \in execs(X_i)$,*
2. *for all $j > 0$: if $a_j \notin \hat{sig}(X_i)(C_{j-1,i})$ then $C_{j-1,i} = C_{j,i}$*

Then, $\alpha \in execs(X)$.

4 Simulation

Since the semantics of a system is given by its configuration automaton, we define a notion of forward simulation from one configuration automaton to another. Our notion requires the usual matching of every transition of the implementation by an execution fragment of the specification. It also requires that corresponding configurations have the same external signature. This gives us a reasonable notion of refinement, in that an implementation presents to its environment only those interfaces (i.e., external signatures) that are allowed by the specification.

Definition 15 (Forward simulation). *Let X and Y be configuration automata. A* forward simulation *from X to Y is a relation f over $states(X) \times states(Y)$ that satisfies:*

1. *if $C \in start(X)$, then $f[C] \cap start(Y) \neq \emptyset$,*
2. *if $C \xrightarrow{a}_X C'$ and $D \in f[C]$, then there exists $D' \in f[C']$ such that*
   a) *$D \xrightarrow{\alpha_1}_Y D_1 \xrightarrow{a}_Y D_2 \xrightarrow{\alpha_2}_Y D'$,*
   b) *$ext(Y)(D_3) = ext(X)(C)$ for all D_3 along α_1 (including D, D_1),*
   c) *$ext(Y)(D_4) = ext(X)(C')$ for all D_4 along α_2 (including D_2, D').*

We say $X \leq Y$ if a forward simulation from X to Y exists. Our notion of correct implementation with respect to safety properties is given by trace inclusion, and is implied by forward simulation.

Theorem 3. *If $X \leq Y$ then $traces(X) \subseteq traces(Y)$.*

5 Modeling Dynamic Connection and Locations

We stated in the introduction that we model both the dynamic creation/moving of connections, and the mobility of agents, by using dynamically changing external interfaces. The guiding principle here is the notion that an agent should only interact directly with either (1) another co-located agent, or (2) a channel one of whose ends is co-located with the agent. Thus, we restrict interaction according to the current locations of the agents.

We adopt a logical notion of location: a location is simply a value drawn from the domain of "all locations." To codify our guiding principle, we partition the set of SIOA into two subsets, namely the set of agent SIOA, and the set of channel

SIOA. Agent SIOA have a single location, and represent agents, and channel SIOA have two locations, namely their current endpoints. We assume that all configurations are compatible, and codify the guiding principle as follows: for any configuration, the following conditions all hold, (1) two agent SIOA have a common external action only if they have the same location, (2) an agent SIOA and a channel SIOA have a common external action only if one of the channel endpoints has the same location as the agent SIOA, and (3) two channel SIOA have no common external actions.

6 Example: A Travel Agent System

Our example is a simple flight ticket purchase system. A client requests to buy an airline ticket. The client gives some "flight information," f, e.g., route and acceptable times for departure, arrival etc., and specifies a maximum price $f.mp$ they can pay. f contains all the client information, including mp, as well as an identifier that is unique across all client requests. The request goes to a static (always existing) "client agent," who then creates a special "request agent" dedicated to the particular request. That request agent then visits a (fixed) set of databases where the request might be satisfied. If the request agent finds a satisfactory flight in one of the databases, i.e., a flight that conforms to f and has price $\leq mp$, then it purchases some such flight, and returns a flight descriptor fd giving the flight, and the price paid ($fd.p$) to the client agent, who returns it to the client. The request agent then terminates.

The agents in the system are: (1) *ClientAgt*, who receives all requests from the client, (2) $ReqAgt(f)$, responsible for handling request f, and (3) $DBAgt_d, d \in \mathcal{D}$, the agent (i.e., front-end) for database d, where $\mathcal{D}$ is the set of all databases in the system. In writing automata, we shall identify automata using a "type name" followed by some parameters. This is only a notational convenience, and is not part of our model.

We first present a specification automaton, and then the client agent and request agents of an implementation (the database agents provide a straightforward query/response functionality, and are omitted for lack of space). When writing sets of actions, we make the convention that all free variables are universally quantified over their domains, so, e.g., $\{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(\mathit{fd}, \mathit{ok}?)\}$ within action $\mathsf{select}_d(f)$ below really denotes $\{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(\mathit{fd}, \mathit{ok}?) \mid \mathit{fd} \in \mathcal{F}, \mathit{flts} \subseteq \mathcal{F}, \mathit{ok}? \in \mathit{Bool}\}$.

In the implementation, we enforce locality constraints by modifying the signature of $ReqAgt(f)$ so that it can only query a database d if it is currently at location d (we use the databse names for their locations). We allow $ReqAgt(f)$ to communicate with *ClientAgt* regardless of its location. A further refinement would insert a suitable channel between $ReqAgt(f)$ and *ClientAgt* for this communication (one end of which would move along with $ReqAgt(f)$), or would move $ReqAgt(f)$ back to the location of *ClientAgt*.

We use "state variables" *in* and *outreg* to denote the current sets of *in*, *outregular* and *int* actions in the SIOA state signature (these are the only components of the signature that vary). For brevity, the universal signature declara-

tion groups actions into input, output and internal only. The actual action types are declared in the "precondition-effect" action descriptions.

Specification: *Spec*

Universal Signature

Input:
  $\mathsf{request}(f)$, where $f \in \mathcal{F}$
  $\mathsf{inform}_d(f, \mathit{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\mathit{flts} \subseteq \mathcal{F}$
  $\mathsf{conf}_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $f, fd \in \mathcal{F}$, and $ok? \in \mathit{Bool}$
  $\mathsf{select}_d(f)$, where $d \in \mathcal{D}$ and $f \in \mathcal{F}$
  $\mathsf{adjustsig}(f)$, where $f \in \mathcal{F}$
  initially: $\{\mathsf{request}(f) : f \in \mathcal{F}\} \cup \{\mathsf{select}_d(f) : d \in \mathcal{D}, f \in \mathcal{F}\}$
Output:
  $\mathsf{query}_d(f)$, where $d \in \mathcal{D}$ and $f \in \mathcal{F}$
  $\mathsf{buy}_d(f, \mathit{flts})$, where $d \in \mathcal{D}$, $f \in \mathcal{F}$, and $\mathit{flts} \subseteq \mathcal{F}$
  $\mathsf{response}(f, fd, ok?)$, where $f, fd \in \mathcal{F}$ and $ok? \in \mathit{Bool}$
  initially: $\{\mathsf{response}(f, fd, ok?) : f, fd \in \mathcal{F}, ok? \in \mathit{Bool}\}$
Internal:
  initially: $\emptyset$

State

$\mathit{status}_f \in \{\text{notsubmitted}, \text{submitted}, \text{computed}, \text{replied}\}$, status of request f, initially notsubmitted
$\mathit{trans}_{f,d} \in \mathit{Bool}$, true iff the system is currently interacting with database d on behalf of request f, initially false
$\mathit{okflts}_{f,d} \subseteq \mathcal{F}$, set of acceptable flights that has been found so far, initially empty
$\mathit{resps} \subseteq \mathcal{F} \times \mathcal{F} \times \mathit{Bool}$, responses that have been calculated but not yet sent to client, initially empty
$x_{f,d} \in \mathcal{N}$, bound on the number of times database d is queried on behalf of request f before a negative reply is returned to the client, initially any natural number greater than zero

Actions

Input $\mathsf{request}(f)$
Eff: $\mathit{status}_f \leftarrow$ submitted

Input $\mathsf{select}_d(f)$
Eff: $\mathit{in} \leftarrow$
  $(\mathit{in} \cup \{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(fd, ok?)\}) -$
  $\{\mathsf{inform}_{d'}(f, \mathit{flts}), \mathsf{conf}_{d'}(fd, ok?) : d' \neq d\}$;
  $\mathit{outreg} \leftarrow$
  $(\mathit{outreg} \cup \{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd)\}) -$
  $\{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, fd) : d' \neq d\}$

Outregular $\mathsf{query}_d(f)$
Pre: $\mathit{status}_f =$ submitted $\wedge\ x_{f,d} > 0$
Eff: $x_{f,d} \leftarrow x_{f,d} - 1$;
  $\mathit{trans}_{f,d} \leftarrow \mathit{true}$

Input $\mathsf{inform}_d(f, \mathit{flts})$
Eff: $\mathit{okflts}_{f,d} \leftarrow \mathit{okflts}_{f,d} \cup$
  $\{fd : fd \in \mathit{flts} \wedge fd.p \leq f.mp\}$

Outregular $\mathsf{buy}_d(f, \mathit{flts})$
Pre: $\mathit{status}_f =$ submitted $\wedge$
  $\mathit{flts} = \mathit{okflts}_{f,d} \neq \emptyset \wedge \mathit{trans}_{f,d}$
Eff: *skip*

Input $\mathsf{conf}_d(f, fd, ok?)$
Eff: $\mathit{trans}_{f,d} \leftarrow \mathit{false}$;
  if $ok?$ then
    $\mathit{resps} \leftarrow \mathit{resps} \cup \{\langle f, fd, \mathit{true}\rangle\}$;
    $\mathit{status}_f \leftarrow$ computed
  else
    if $\forall d : x_{f,d} = 0$ then
      $\mathit{resps} \leftarrow \mathit{resps} \cup \{\langle f, \bot, \mathit{false}\rangle\}$;
      $\mathit{status}_f \leftarrow$ computed
    else
      skip

Outregular $\mathsf{response}(f, fd, ok?)$
Pre: $\langle f, fd, ok?\rangle \in \mathit{resps} \wedge \mathit{status}_f =$ computed
Eff: $\mathit{status}_f \leftarrow$ replied

Input $\mathsf{adjustsig}(f)$
Eff: $\mathit{in} \leftarrow \mathit{in} -$
    $\{\mathsf{inform}_d(f, \mathit{flts}), \mathsf{conf}_d(f, fd, ok?)\}$;
  $\mathit{outreg} \leftarrow \mathit{outreg} -$
    $\{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd)\}$

We now give the client agent and request agents of the implementation. The initial configuration consists solely of the client agent *ClientAgt*.

Client Agent: *ClientAgt*

Universal Signature

Input:
  request(f), where $f \in \mathcal{F}$
  req-agent-response($f, fd, ok?$), where $f, fd \in \mathcal{F}$, and $ok? \in Bool$
Output:
  response($f, fd, ok?$), where $f, fd \in \mathcal{F}$ and $ok? \in Bool$
Internal:
  create($ClientAgt, ReqAgt(f)$), where $f \in \mathcal{F}$

State

$reqs \subseteq \mathcal{F}$, outstanding requests, initially empty
$created \subseteq \mathcal{F}$, outstanding requests for whom a request agent has been created, but the response has not yet been returned to the client, initially empty
$resps \subseteq \mathcal{F} \times \mathcal{F} \times Bool$, responses not yet returned to client, initially empty

Actions

Input request(f)
Eff: $reqs \leftarrow reqs \cup \{\langle f \rangle\}$

Create create($ClientAgt, ReqAgt(f)$)
Pre: $f \in reqs \wedge f \notin created$
Eff: $created \leftarrow created \cup \{f\}$

Input req-agent-response($f, fd, ok?$)
Eff: $resps \leftarrow resps \cup \{\langle f, fd, ok? \rangle\}$;
  $done \leftarrow done \cup \{f\}$

Outregular response($f, fd, ok?$)
Pre: $\langle f, fd, ok? \rangle \in resps$
Eff: $resps \leftarrow resps - \{\langle f, fd, ok? \rangle\}$

ClientAgt receives requests from a client (not portrayed), via the request input action. *ClientAgt* accumulates these requests in *reqs*, and creates a request agent $ReqAgt(f)$ for each one. Upon receiving a response from the request agent, via input action req-agent-response, the client agent adds the response to the set *resps*, and subsequently communicates the response to the client via the response output action. It also removes all record of the request at this point.

Request Agent: $ReqAgt(f)$ where $f \in \mathcal{F}$

Universal Signature

Input:
  inform$_d(f, flts)$, where $d \in \mathcal{D}$ and $flts \subseteq \mathcal{F}$
  conf$_d(f, fd, ok?)$, where $d \in \mathcal{D}$, $fd \in \mathcal{F}$, and $ok? \in Bool$
  move$_f(c, d)$, where $d \in \mathcal{D}$
  move$_f(d, d')$, where $d, d' \in \mathcal{D}$ and $d \neq d'$
  terminate($ReqAgt(f)$)
  initially: $\{\text{move}_f(c, d),$ where $d \in \mathcal{D}\}$
Output:
  query$_d(f)$, where $d \in \mathcal{D}$
  buy$_d(f, flts)$, where $d \in \mathcal{D}$ and $flts \subseteq \mathcal{F}$
  req-agent-response($f, fd, ok?$), where $fd \in \mathcal{F}$ and $ok? \in Bool$
  initially: $\emptyset$
Internal:
  initially: $\emptyset$

State

$location \in c \cup \mathcal{D}$, location of the request agent, initially c, the location of *ClientAgt*
$status \in \{\text{notsubmitted}, \text{submitted}, \text{computed}, \text{replied}\}$, status of request f, initially notsubmitted
$trans_d \in Bool$, true iff $ReqAgt(f)$ is currently interacting with database d (on behalf of request f), initially false
$DBagents \subseteq \mathcal{D}$, databases that have not yet been queried, initially the list of all databases $\mathcal{D}$
$donedb \in Bool$, boolean flag, initially *false*
$done \in Bool$, boolean flag, initially *false*
$tkt \in \mathcal{F}$, the flight ticket that $ReqAgt(f)$ purchases on behalf of the client, initially $\perp$
$okflts_d \subseteq \mathcal{F}$, set of acceptable flights that $ReqAgt(f)$ has found so far, initially empty

Actions

Input $\mathsf{move}_f(c, d)$
Eff: $location \leftarrow d$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_d(f, flts), \mathsf{conf}_d(f, fd, ok?)\}$;
$outreg \leftarrow \{\mathsf{query}_d(f), \mathsf{buy}_d(f, fd),$
$\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;
$int \leftarrow \emptyset$

Outregular $\mathsf{query}_d(f)$
Pre: $location = d \wedge d \in DBagents \wedge tkt = \perp$
Eff: $DBagents \leftarrow DBagents - \{d\}$;
$trans_d \leftarrow true$

Input $\mathsf{inform}_d(f, flts)$
Eff: $okflts_d \leftarrow okflts_d \cup$
$\{fd : fd \in flts \wedge fd.p \leq f.mp\}$;
if $okflts_d = \emptyset$ then
$trans_d \leftarrow false$;
$int \leftarrow \{\mathsf{move}_f(d, d') :$
$d' \in DBagents - \{d\}\}$

Outregular $\mathsf{buy}_d(f, flts)$
Pre: $location = d \wedge flts = okflts_d \neq \emptyset \wedge$
$tkt = \perp \wedge trans_d \wedge status = \text{submitted}$
Eff: $skip$

Input $\mathsf{conf}_d(f, fd, ok?)$
Eff: $trans_d \leftarrow false$;
if $ok?$ then
$tkt \leftarrow fd$;
$status \leftarrow \text{computed}$
else
if $DBagents = \emptyset$ then
$status \leftarrow \text{computed}$
else
$skip$

Input $\mathsf{move}_f(d, d')$
Eff: $location \leftarrow d'$;
$donedb \leftarrow false$;
$in \leftarrow \{\mathsf{inform}_{d'}(f, flts), \mathsf{conf}_{d'}(f, fd, ok?)\}$;
$outreg \leftarrow \{\mathsf{query}_{d'}(f), \mathsf{buy}_{d'}(f, fd),$
$\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)\}$;
$int \leftarrow \emptyset$

Outregular $\mathsf{req\text{-}agent\text{-}response}(f, fd, ok?)$
Pre: $status = \text{computed} \wedge$
$[\ (fd = tkt \neq \perp \wedge ok?) \vee$
$(DBagents = \emptyset \wedge fd = \perp \wedge \neg ok?)$
$]$
Eff: $status \leftarrow \text{replied}$;
$in \leftarrow \emptyset$;
$outreg \leftarrow \emptyset$;
$int \leftarrow \emptyset$

ReqAgt(f) handles the single request f, and then terminates itself. *ReqAgt*(f) has initial location c (the location of *ClientAgt*) traverses the databases in the system, querying each database d using $\mathsf{query}_d(f)$. Database d returns a set of flights that match the schedule information in f. Upon receiving this ($\mathsf{inform}_d(f, flts)$), *ReqAgt*$(f)$ searches for a suitably cheap flight (the $\exists fd \in flts : fd.p \leq f.mp$ condition in $\mathsf{inform}_d(f, flts)$). If such a flight exists, then *ReqAgt*(f) attempts to buy it ($\mathsf{buy}_d(f, flts)$ and $\mathsf{conf}_d(f, fd, ok?)$). If successfull, then *ReqAgt*(f) returns a positive response to *ClientAgt* and terminates. *ReqAgt*(f) can return a negative response if it queries each database once and fails to buy a flight.

We note that the implementation refines the specification (provided that all actions except $\mathsf{request}(f)$ and $\mathsf{response}(f, fd, ok?)$ are hidden) even though the implementation queries each database exactly once before returning a negative response, whereas the specification queries each database some finite number of times before doing so Thus, no reasonable bisimulation notion could be established between the specification and the implementation. Hence, the use of a simulation, rather than a bisimulation, allows us much more latitude in refining a specification into an implementation.

7 Further Research and Conclusions

There are many avenues for further work. Our most immediate concern is to establish trace projection and pasting results analogous to the execution projection and pasting results given above. These will then allow us to establish substitutivity results of the form: if $traces(X_1) \subseteq traces(X_2)$, then

$traces(X_1 \parallel Y) \subseteq traces(X_2 \parallel Y)$. We shall also investigate ways of allowing a target SIOA of some create action to be replaced by a more refined SIOA. Let $X[B_2]$ be the configuration automaton resulting when some create action (of some SIOA A in X) has target B_2, and let $X[B_1]$ be the configuration automaton that results when this target is changed to B_1. We would like to establish: if $traces(B_1) \subseteq traces(B_2)$, then $traces(X[B_1]) \subseteq traces(X[B_2])$.

Agent systems should be able to operate in a dynamic environment, with processor failures, unreliable channels, and timing uncertainties. Thus, we need to extend our model to deal with fault-tolerance and timing. We shall also extend the framework of [3] for verifying liveness properties to our model. This should be relatively straightforward, since [3] uses only properties of forward simulation that should also carry over to our setting.

Acknowledgments. The first author was supported in part by NSF CAREER Grant CCR-9702616.

References

1. Tadashi Araragi, Paul Attie, Idit Keidar, Kiyoshi Kogure, Victor Luchangco, Nancy Lynch, and Ken Mano. On formal modeling of agent computations. In *NASA Workshop on Formal Approaches to Agent-Based Systems*, Apr. 2000. To appear in Springer LNCS.
2. Paul Attie and Nancy Lynch. Dynamic input/output automata: a formal model for dynamic systems. Technical report, Northeastern University, Boston, Mass., 2001. Available at http://www.ccs.neu.edu/home/attie/pubs.html.
3. P.C. Attie. Liveness-preserving simulation relations. In *Proceedings of the 18'th Annual ACM Symposium on Principles of Distributed Computing*, pages 63–72, 1999.
4. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.
5. Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96), Springer-Verlag, LNCS 1119*, pages 406–421, Aug. 1996.
6. Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. In *Proceedings of the 3'rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984.
7. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. CWI-Quarterly, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Also, Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.
8. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
9. R. Milner. *Communicating and mobile systems: the π-calculus*. Addison-Wesley, Reading, Mass., 1999.
10. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

Probabilistic Information Flow in a Process Algebra

Alessandro Aldini

Università di Bologna, Dipartimento di Scienze dell'Informazione,
Mura Anteo Zamboni 7, 40127 Bologna, Italy,
aldini@cs.unibo.it
http://www.cs.unibo.it/~aldini

Abstract. We present a process algebraic approach for extending to the probabilistic setting the classical logical information flow analysis of computer systems. In particular, we employ a calculus for the analysis of probabilistic systems and a notion of probabilistic bisimulation in order to define classical security properties, such as nondeterministic noninterference (NNI) and nondeducibility on compositions (NDC), in the probabilistic setting. We show how to (i) extend the results known for the nondeterministic case, (ii) analyse insecure nondeterministic behaviors, and (ii) reveal probabilistic covert channels which may be not observable in the nondeterministic case. Finally, we show that the expressiveness of the calculus we adopt makes it possible to model concurrent systems in order to derive also performance measures.

1 Introduction

There is a lot of work in the security community which aims at proposing formal definitions related to confidentiality in real systems. One of the main techniques used for verifying the non-occurrence of unauthorized disclosure of information is the analysis of the information flow among the different components of the system (see, e.g., [33,23,18]). A well established approach used to conduct such an analysis is based on an extentional characterization usually known as noninterference [17]. In particular, the use of process algebras to formalize the idea of noninterference has received increased attention in recent years (see, e.g., [28,12, 26,29]). Most of such process algebraic approaches address the problem of defining and analysing information flows in a nondeterministic setting. In particular, in [12] Focardi and Gorrieri promote the classification of a set of properties capturing the idea of information flow and noninterference. More precisely, they employ an extension of CCS [25] where events are partitioned into two different levels of confidentiality (low level and high level), thus allowing the whole flow of information between the two different levels to be controlled. The main idea underlying the verification of a given security property consists in deriving two models P' and P'' from the same model of the system in such a way that their definition depends on the particular security property we intend to check. Then

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 152–168, 2001.
© Springer-Verlag Berlin Heidelberg 2001

the verification of the property simply consists in checking the semantic equivalence between P' and P''. The most interesting and intuitive security properties capturing information flows are the Non Deducibility on Compositions (*NDC*) and the Nondeterministic Non Interference (*NNI*). For instance, the NDC property can be described as follows: $\forall \Pi \in HighUsers.\ E \,|\, \Pi \approx E\ w.r.t.\ LowUsers$, where $|$ stands for the parallel composition operator and $\approx$ is the equivalence relation. Such formula says that a system E is *NDC* secure if, from the low level user standpoint, the behavior of the system is invariant w.r.t. the composition with every high level user Π. The above properties describe the nondeterministic behavior of systems, and often this assumption is more than enough to reveal insecure information flows. However, in many cases, a detailed and closer to the implementation description of a system should include additional aspects (such as time and probabilities) as a possibly observable behavior (e.g. in order to check the existence of probabilistic covert channels). To this aim, in this paper we propose a process algebraic approach for the analysis of security properties in a probabilistic setting which extends the approach of [12]. The motivation of this work is twofold.

On the one hand, an approach that considers the probabilistic aspect of systems may contribute to reveal new information flows that do not arise when considering only the nondeterministic behavior of systems. For instance, a low level user of a system could observe in different contexts the same set of events but with different probability distributions associated with each event. More precisely we show that, when considering more concrete information, a system which is secure according to the possibilistic version of *NDC* (*NNI*) can become insecure when passing to the probabilistic version of *NDC* (*NNI*).

On the other hand, by introducing probabilities we can give a probabilistic measure to the insecure behaviors captured in the nondeterministic setting. For instance, a low level user may check if the probability of observing an insecure behavior of a system is beyond a threshold for which he considers the system to be secure "enough". Another reason for considering probabilities is that more concrete models allow the modeler to describe in the same specification different aspects of the same system and, e.g., to analyse on the same model both performance related properties and information flow security properties.

The process algebra we employ in order to apply the above ideas is quite different from the CCS-like calculus proposed in [12,13,14], the main reason being that our language is particularly adequate to combine powerful mechanisms like probabilistic internal/external choices, asynchronous execution of parallel processes, multiway synchronization, and an asymmetric master-slave cooperation discipline, which are very suitable to describe the behavior of real systems (see e.g. [2]). Moreover, fully specified systems modeled with such a language give rise to fully probabilistic systems, so that a Markov Chain can be derived and easily analysed to get performance measures of the system. As far as the equivalence relation we adopt is concerned, we point out that the possibilistic noninterference like properties of [12,13] are defined with respect to different equivalences, among which the weak bisimulation seems to be particularly appropriate to deal

with some kind of information flows. In this paper we resort to a probabilistic weak bisimulation, and based on this assumption, we define the probabilistic version of Bisimulation *NDC* and Bisimulation Strong *NNI*.

The paper is organized as follows. After an overview of related work, we describe in Sect. 2 the probabilistic calculus by introducing the underlying model (an extension of classical labeled transition systems), the syntax, and the semantics of the algebra. We then present the notion of probabilistic weak bisimulation (Sect. 3) and the bisimulation based security properties (Sect. 4). Finally, in Sect. 5 we report some conclusion. Due to space limitations, we omit the proofs of the results; the interested reader is referred to [1].

1.1 Related Work

A significant amount of work has been done in order to extend the relation among potential insecure information flows and different aspects of concurrent systems such as time and probabilities. As an example, programs which execute several concurrent processes not only introduce nondeterminism, but also the potential for a malicious party to observe the internal timing behaviors of programs, via the effect that the running time of commands might have on the scheduler choice of when various concurrent alternatives can be executed (see, e.g., [32,14,11]). On the other hand, in [33,18] the authors claim that real systems may exhibit probabilistic covert channels that are not ruled out by standard nondeterministic security models; in particular, the author of [18] proposes a probabilistic version of Millen's synchronous state machine [24] on which two security properties are rephrased. On the same subject, the authors of [31] formalize the idea of confidentiality operationally for a simple imperative language with dynamic thread creation, by capturing the probabilistic information flow that arises from the scheduling of concurrent threads. In the same line of the above discussion, in [22] the author considers the notion of Probabilistic Noninterference (PNI) in the setting of a model of probabilistic dataflow, by showing the compositionality of PNI in such a context and that a simpler nonprobabilistic notion of noninterference, called Nondeducibility on Strategies [33], is an instantiation of PNI. Moreover, the authors of [8] resort to a possibilistic information flow analysis of a Probabilistic Idealised Algol to check for probabilistic interference, and in [9] a probabilistic security analysis is proposed in the framework of a declarative programming language. Finally, in [19] the authors set out a modal logic for reasoning about multilevel security of probabilistic systems. However, to the best of our knowledge, no approach has been proposed for extending with probabilities the information flow theory in process algebras.

2 A Probabilistic Calculus

In this section we introduce a smooth extension of the probabilistic process algebra proposed in [3,6] for modeling and analysing probabilistic systems. Such a

calculus adopts a mixture of the generative and reactive approaches [16] by considering an *asymmetric* form of synchronization where a process which behaves generatively may synchronize only with processes which behave reactively. The integration of the generative and reactive approaches has been naturally obtained by designating some actions, called *generative* actions, as predominant over the other ones, called *reactive* actions (denoted by a subscript $*$), and by imposing that generative actions can synchronize with reactive actions only. We see the reactive actions as *incomplete* actions which must synchronize with generative actions of another system component in order to form a complete system. A system is considered to be *fully specified* only when it gives rise to a probabilistic transition system which is purely generative, in the sense that it does not include reactive transitions. Fully specified systems are therefore fully probabilistic systems from which a Markov Chain can be derived (by discarding actions from transition labels) that can be easily analysed to get performance measures.

In the following, we first describe the models generated by terms of the calculus, and then we extend the basic language with the operators needed for the description of security properties. Finally, we equip the algebra with a probabilistic weak bisimulation equivalence by following the same line of [5].

2.1 The Model

Generative-reactive transition systems [3] are composed of transitions labeled with an action, which can be either generative or reactive, and a probability. Formally, we denote the set of action types by $AType$, ranged over by a, b, ... As usual $AType$ includes the special type τ denoting internal actions. We denote the set of reactive actions by $RAct = \{a_* \mid a \in AType - \{\tau\}\}$ and the set of generative actions by $GAct = AType$. The set of actions is denoted by $Act = RAct \cup GAct$, ranged over by π, π', ... Transitions leaving a state are grouped in several bundles. We have a single generative bundle composed of all the transitions labeled with a generative action and several reactive bundles, each one referring to a different action type a and composed of all the transitions labeled with a_*. A bundle of transitions expresses a probabilistic choice. On the contrary the choice among bundles is performed non-deterministically.

Definition 1. *A Generative-Reactive Transition System GRTS is a quadruple* $(S, AType, T, s_0)$ *with* S *a set of states and* s_0 *the initial one,* $AType$ *a set of action types,* $T \in \mathcal{M}(S \times Act \times]0,1] \times S)$ *a multiset* [1] *of probabilistic transitions, such that*

1. $\forall s \in S, \forall a_* \in RAct.\ \sum\{\!|\, p \mid \exists t \in S : (s, a_*, p, t) \in T \,|\!\} \in \{0, 1\}$
2. $\forall s \in S.\ \sum\{\!|\, p \mid \exists a \in GAct, t \in S : (s, a, p, t) \in T \,|\!\} \in \{0, 1\}$ □

The first requirement defines the reactive bundles and the second requirement defines the unique generative bundle. Both requirements say that for each state

[1] We use "$\{\!|$" and "$|\!\}$" as brackets for multisets and $\mathcal{M}(S)$ to denote the collection of multisets over set S.

the probabilities of the transitions composing a bundle, if there are any, sum up to 1 (otherwise the summation over empty multisets is defined equal to 0). Graphically, transitions of the same bundle are grouped by an arc, and the probability of a transition is omitted when equal to 1 (see Fig. 1).

2.2 The Language

In this section we introduce an extension of the calculus proposed in [3,6], by adding the restriction and hiding operators. Let *Const* be a set of constants, ranged over by $A, B, \ldots$. The set $\mathcal{L}$ of process terms is generated by the syntax:

$$P ::= \underline{0} \mid \pi.P \mid P +^p P \mid P \parallel_S^p P \mid P[a \to b]^p \mid P\backslash L \mid P/_a^p \mid A$$

where $S, L \subseteq AType - \{\tau\}$, $a, b \in AType - \{\tau\}$, and $p \in]0,1[$. The set $\mathcal{L}$ is ranged over by $P, Q, \ldots$. We denote by $\mathcal{G}$ the set of guarded and closed terms of $\mathcal{L}$. Moreover, we denote two disjoint sets $AType_H$ and $AType_L$ of high and low level action types which form a covering of $AType - \{\tau\}$, such that $a \in GAct$ and $a_* \in RAct$ are high (low) level actions if $a \in AType_H$ ($a \in AType_L$). Let $\mathcal{G}_H = \{P \in \mathcal{G} \mid sort(P) \subseteq AType_H\}$ be the set of high level terms (i.e. including high level actions only). An informal overview of the operators is as follows.

$\underline{0}$ represents a terminated or deadlocked term having no transitions. The prefix operator $\pi.P$ performs the action π with probability 1 and then behaves like P. Constants A are used to specify recursive systems. In general, when defining an algebraic specification, we assume a set of constants defining equations of the form $A \triangleq P$ to be given.

The alternative composition operator $P +^p Q$ represents a probabilistic choice between the generative actions of P and Q and between the reactive actions of P and Q of the same type. As far as generative actions are concerned, $P +^p Q$ executes a generative action of P with probability p and a generative action of Q with probability $1 - p$. In the case one process P or Q cannot execute generative actions, $P +^p Q$ chooses a generative action of the other process with probability 1 (similarly as in [4]). As far as reactive actions of a given type a are concerned, $P +^p Q$ chooses between the reactive actions a_* of P and Q according to probability p, by following the same mechanism. As an example, in Fig. 1(a) we report the *GRTS*s generated by the terms[2] $a +^p b$ and $b_* +^p b_*$ representing purely probabilistic choices made according to p. On the other hand, in Fig. 1(b) terms $a +^p b_*$ and $a_* +^p b_*$ represent purely nondeterministic choices, where p is not considered. Finally, Fig. 1(c) shows a mixed probabilistic and nondeterministic system, corresponding to the term $(a +^{p'} b_*) +^p (b +^{p''} b_*)$, where p' and p'' are not considered.

The parallel composition operator $P \parallel_S^p Q$ is based on a CSP like synchronization policy, where processes P and Q are required to synchronize over actions of type in the set S, and locally execute all the other actions. A synchronization between two actions of type a may occur only if either they are both reactive actions a_* (and the result is a reactive action a_*), or one of them is a generative

[2] We abbreviate terms $\pi.\underline{0}$ by omitting the final $\underline{0}$.

action a and the other one is a reactive action a_* (and the result is a generative action a). The generative actions of P (Q) executable by $P \|^p_S Q$ are such that either their type a is not in S, or a is in S and Q (P) can perform some reactive action a_*. In particular, as standard when restricting actions in the generative model [16], the probabilities of executing such actions are proportionally redistributed so that their overall probability sums up to 1. The choice among the generative actions of P and Q executable by $P \|^p_S Q$ is made according to probability p, by following the same probabilistic mechanism seen for alternative composition. In the case of synchronizing generative actions a of P (Q), their probability is further redistributed among the reactive actions a_* executable by Q (P), according to the probability they are chosen in Q (P). As far as reactive actions of a given type $a \notin S$ are concerned, $P \|^p_S Q$ may perform all the reactive actions a_* executable by P or Q and the choice among them is made according to probability p, by following the same probabilistic mechanism seen for alternative composition. As far as reactive actions of a given type $a \in S$ are concerned, if both P and Q may execute some reactive action a_*, the choice of the two actions a_* of P and Q forming the actions a_* executable by $P \|^p_S Q$ is made according to the probability they are *independently* chosen by P and Q.

The relabeling operator $P[a \to b]^p$ turns actions of type a into actions of type b. The parameter p expresses the probability that reactive actions b_* obtained by relabeling actions a_* of P are executed with respect to the actions b_* previously performable by P. As an example, consider the second *GRTS* of Fig. 1(b), corresponding to the process $P \stackrel{\Delta}{=} a_* +^q b_*$, where the choice is purely nondeterministic. If we apply the relabeling operator $P[a \to b]^p$ we obtain the process represented by the second *GRTS* of Fig. 1(a), where the semantics of $P[a \to b]^p$ is a probabilistic choice between the action b_* obtained by relabeling the action a_* and the other action b_*, performed according to probabilities p and $1 - p$, respectively. In this way the probabilistic information p provided in the operator $P[a \to b]^p$ guarantees that the relabeling operator does not introduce non-determinism between reactive actions of the same type. Parameter p is, instead, not used when relabeling generative actions because the choice between generative actions of type a and b in P is already probabilistic.

The restriction operator $P \backslash L$ prevents the execution of the actions with type in L. In this work, we have introduced this additional operator in order to make simple the definition of the security properties. In fact, it can be obtained by resorting to the parallel operator, because $P \backslash L = P \|^p_L \underline{0}$, for each $p \in]0, 1[$.

The hiding operator $P/^p_a$ turns generative and reactive actions of type a into generative actions τ. The parameter p expresses the probability that actions τ

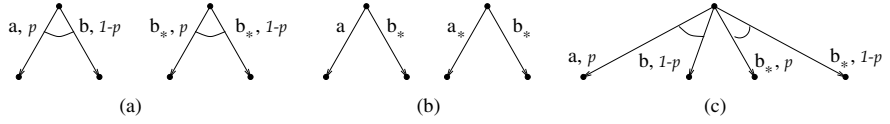


Fig. 1. Some examples of *GRTSs* derived from alternative composition

obtained by hiding reactive actions a_* of P are executed with respect to the generative actions previously enabled by P (by following the same probabilistic mechanism seen for relabeling). As an example, let us consider the processes[3] $P \stackrel{\Delta}{=} l.P +^q h.P$ and $P' \stackrel{\Delta}{=} l.P' +^q h_*.P'$. When hiding the high level actions of such processes (e.g. in order to verify the noninterference property), we want to obtain two processes equivalent to $Q \stackrel{\Delta}{=} l.Q +^q \tau.Q$, because both generative and reactive high level actions are not observable by a low level user. In particular, when hiding P', we have that the nondeterministic choice becomes a probabilistic choice. Actually, we consider the effect of hiding the reactive action h_* as the execution of a synchronization between h_* and an external generative action h that gives rise to an internal generative action τ. As we will see in Sect. 4, the process P' turns out to be secure if the probability distribution of the hidden actions is not able to change the probabilistic behavior of the low level view of the process itself, meaning that the probability of the external high level action h (which synchronizes with h_*) is not meaningful, because it does not alter the probabilistic behavior of the low view of the system.

The formal semantics of our calculus maps terms onto *GRTS*s, where each label is composed of an action and a probability. The *GRTS* deriving from a term $\mathcal{G}$ is defined by the operational rules in Tables 1 and 2, where in addition to rules undersigned with l, which refer to the local moves of the lefthand process P, we consider also the symmetrical rules taking into account the local moves of the righthand process Q, obtained by exchanging the roles of terms P and Q in the premises and by replacing p with $1-p$ in the label of the derived transitions. We use $P \stackrel{\pi}{\longrightarrow}$ to stand for $\exists\, p, P' : P \stackrel{\pi,\, p}{\longrightarrow} P'$, meaning that P can execute action π, and $P \stackrel{G}{\longrightarrow}$ to stand for $\exists\, a \in G : P \stackrel{a}{\longrightarrow}$, $G \subseteq AType$, meaning that P can execute a generative action of type belonging to set G. We assume the sets $G_{S,P}, G_L \subseteq AType$, with $S, L \subseteq AType - \{\tau\}$ and $P \in \mathcal{G}$, to be defined as follows: $G_{S,P} = \{a \in AType \,|\, a \notin S \vee (a \in S \wedge P \stackrel{a_*}{\longrightarrow})\}$ and $G_L = \{a \in AType \,|\, a \notin L\}$. $G_{S,Q}$ ($G_{S,P}$) is the set of types of the generative transitions of P (Q) executable by $P \parallel_S^p Q$ and G_L is the set of types of the generative transitions of P executable by $P \backslash L$. Since we consider a restricted set of executable actions, we redistribute the probabilities of the generative transitions of P (Q) executable by $P \parallel_S^p Q$ and $P \backslash L$ so that their overall probability sums up to 1 [16]. To this aim in semantics rules we employ the function $\nu_P(G) : \mathcal{P}(AType) \longrightarrow]0, 1]$, with $P \in \mathcal{G}$, defined as $\nu_P(G) = \sum \{\!|\, p \,|\, \exists P', a \in G : P \stackrel{a,\, p}{\longrightarrow} P' \,|\!\}$ that computes the sum of the probabilities of the generative transitions executable by P whose type belongs to the set G. Hence $\nu_P(G_{S,Q})$ ($\nu_Q(G_{S,P})$) and $\nu_P(G_L)$ compute the overall probability of the generative transitions of P (Q) executable by $P \parallel_S^p Q$ and $P \backslash L$, respectively. Finally, we employ the following abbreviations to denote the hiding of high level actions.

Definition 2. *Let "P/L", where L is a finite sequence $\langle a_1^{p_1}, \ldots, a_n^{p_n} \rangle$ of actions $a_i \neq \tau$ with an associated probability p_i, stand for the expression $P/^{p_1}_{a_1} \ldots /^{p_n}_{a_n}$,*

[3] We denote with $l, l', \ldots$ low level types and with $h, h', \ldots$ high level types.

Table 1. Operational semantics for the basic calculus

$$(gr1)\ \pi.P \xrightarrow{\pi,1} P$$

$$(r1_l)\ \frac{P \xrightarrow{a_*,q} P' \quad Q \xrightarrow{a_*}}{P +^p Q \xrightarrow{a_*,p\cdot q} P'} \qquad (r2_l)\ \frac{P \xrightarrow{a_*,q} P' \quad Q \not\xrightarrow{a_*}}{P +^p Q \xrightarrow{a_*,q} P'}$$

$$(g1_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \xrightarrow{GAct}}{P +^p Q \xrightarrow{a,p\cdot q} P'} \qquad (g2_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \not\xrightarrow{GAct}}{P +^p Q \xrightarrow{a,q} P'}$$

$$(r3_l)\ \frac{P \xrightarrow{a_*,q} P' \quad Q \xrightarrow{a_*}}{P \parallel^p_S Q \xrightarrow{a_*,p\cdot q} P' \parallel^p_S Q}\ a \notin S \qquad (r4_l)\ \frac{P \xrightarrow{a_*,q} P' \quad Q \not\xrightarrow{a_*}}{P \parallel^p_S Q \xrightarrow{a_*,q} P' \parallel^p_S Q}\ a \notin S$$

$$(r5)\ \frac{P \xrightarrow{a_*,q} P' \quad Q \xrightarrow{a_*,q'} Q'}{P \parallel^p_S Q \xrightarrow{a_*,q\cdot q'} P' \parallel^p_S Q'}\ a \in S$$

$a \notin S$:

$$(g3_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \xrightarrow{G_{S,P}}}{P \parallel^p_S Q \xrightarrow{a,p\cdot q/\nu_P(G_{S,Q})} P' \parallel^p_S Q} \qquad (g4_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \not\xrightarrow{G_{S,P}}}{P \parallel^p_S Q \xrightarrow{a,q/\nu_P(G_{S,Q})} P' \parallel^p_S Q}$$

$a \in S$:

$$(g5_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \xrightarrow{a_*,q'} Q' \quad Q \xrightarrow{G_{S,P}}}{P \parallel^p_S Q \xrightarrow{a,p\cdot q'\cdot q/\nu_P(G_{S,Q})} P' \parallel^p_S Q'} \qquad (g6_l)\ \frac{P \xrightarrow{a,q} P' \quad Q \xrightarrow{a_*,q'} Q' \quad Q \not\xrightarrow{G_{S,P}}}{P \parallel^p_S Q \xrightarrow{a,q'\cdot q/\nu_P(G_{S,Q})} P' \parallel^p_S Q'}$$

$$(r6)\ \frac{P \xrightarrow{a_*,q} P' \quad P \xrightarrow{b_*}}{P[a \to b]^p \xrightarrow{b_*,p\cdot q} P'[a \to b]^p} \qquad (r7)\ \frac{P \xrightarrow{a_*,q} P' \quad P \not\xrightarrow{b_*}}{P[a \to b]^p \xrightarrow{b_*,q} P'[a \to b]^p}$$

$$(r8)\ \frac{P \xrightarrow{b_*,q} P' \quad P \xrightarrow{a_*}}{P[a \to b]^p \xrightarrow{b_*,(1-p)\cdot q} P'[a \to b]^p} \qquad (r9)\ \frac{P \xrightarrow{b_*,q} P' \quad P \not\xrightarrow{a_*}}{P[a \to b]^p \xrightarrow{b_*,q} P'[a \to b]^p}$$

$$(r10)\ \frac{P \xrightarrow{c_*,q} P'}{P[a \to b]^p \xrightarrow{c_*,q} P'[a \to b]^p}\ c \notin \{a, b\}$$

$$(g7)\ \frac{P \xrightarrow{a,q} P'}{P[a \to b]^p \xrightarrow{b,q} P'[a \to b]^p} \qquad (g8)\ \frac{P \xrightarrow{c,q} P'}{P[a \to b]^p \xrightarrow{c,q} P'[a \to b]^p}\ a \neq c$$

$$(gr2)\ \frac{P \xrightarrow{\pi,q} P'}{A \xrightarrow{\pi,q} P'}\ \text{if } A \triangleq P$$

Table 2. Operational semantics for restriction and hiding

$$(r11)\ \frac{P \xrightarrow{a_*,q} P'}{P\backslash L \xrightarrow{a_*,q} P'\backslash L}\ a \notin L \qquad (g9)\ \frac{P \xrightarrow{a,q} P'}{P\backslash L \xrightarrow{a,q/\nu_P(G_L)} P'\backslash L}\ a \notin L$$

$$(r12)\ \frac{P \xrightarrow{a_*,q} P' \quad P \xrightarrow{GAct}}{P/^p_a \xrightarrow{\tau,p\cdot q} P'/^p_a} \qquad (r13)\ \frac{P \xrightarrow{a_*,q} P' \quad P \xrightarrow{GAct} \not\rightarrow}{P/^p_a \xrightarrow{\tau,q} P'/^p_a} \qquad (r14)\ \frac{P \xrightarrow{b_*,q} P'}{P/^p_a \xrightarrow{b_*,q} P'/^p_a}\ a \neq b$$

$$(g10)\ \frac{P \xrightarrow{b,q} P' \quad P \xrightarrow{a_*}}{P/^p_a \xrightarrow{b,(1-p)\cdot q} P'/^p_a}\ a \neq b \qquad (g11)\ \frac{P \xrightarrow{b,q} P' \quad P \xrightarrow{a_*} \not\rightarrow}{P/^p_a \xrightarrow{b,q} P'/^p_a}\ a \neq b$$

$$(g12)\ \frac{P \xrightarrow{a,q} P' \quad P \xrightarrow{a_*}}{P/^p_a \xrightarrow{\tau,(1-p)\cdot q} P'/^p_a} \qquad (g13)\ \frac{P \xrightarrow{a,q} P' \quad P \xrightarrow{a_*} \not\rightarrow}{P/^p_a \xrightarrow{\tau,q} P'/^p_a}$$

hiding the actions with types $a_1, \ldots, a_n$. Let "$P/AType_H$", where we assume the set of high level actions $AType_H$ to be the set $\{h_1, \ldots, h_n\}$, stand for the expression $P/^{p_1}_{h_1} \ldots /^{p_n}_{h_n}$, for any choice of the associated probabilities $p_1, \ldots, p_n$.

Theorem 1. *If P is a process of $\mathcal{G}$, then the operational semantics of P (composed of the terms reachable from P according to the operational rules of Tables 1 and 2) is a GRTS.*

3 Equivalence

In this section we define a weak bisimulation equivalence for generative-reactive transition systems. Let us consider the *GRTS* $(S, AType, T, s_0)$. We define function $Pr : S \times Act \times S \longrightarrow [0,1]$ by $Pr(s,\pi,s') = \sum\{\!|\, p \,|\, (s,\pi,p,s') \in T \,|\!\}$, and $Pr(s,\pi,C) = \sum_{s' \in C} Pr(s,\pi,s')$, $C \subseteq S$. An execution fragment is a finite sequence $\sigma = s_0 \xrightarrow[\pi_1,p_1]{} s_1 \xrightarrow[\pi_2,p_2]{} \ldots s_k$, such that $s_i \in S$, $\pi_i \in Act$ and $p_i > 0$ for each $i \in 1, \ldots, k$. We denote $Pr(\sigma) = Pr(s_0,\pi_1,s_1) \cdot \ldots \cdot Pr(s_{k-1},\pi_k,s_k)$. An execution is an infinite sequence $\sigma' = s_0 \xrightarrow[\pi_1,p_1]{} s_1 \xrightarrow[\pi_2,p_2]{} \ldots$ where $s_i \in S$, $\pi_i \in Act$ and $p_i > 0$ for each $i \in \mathbf{N}$. Finally, let $\sigma\uparrow$ denote the set of executions σ' such that $\sigma \leq_{prefix} \sigma'$ where *prefix* is the usual prefix relation on sequences. Assuming the basic notions of probability theory (see e.g. [20]) we define the probability space on the executions starting in a given state $s \in S$. Let $Exec(s)$ be the set of executions starting in s, and $ExecFrag(s)$ the set of execution fragments starting in s. Moreover, let $\Sigma(s)$ be the smallest sigma field on $Exec(s)$ such that it

contains the basic cylinder $\sigma\uparrow$ where $\sigma \in \mathit{ExecFrag}(s)$. The probability measure Prob is the unique measure on $\Sigma(s)$ such that $\mathit{Prob}(\sigma\uparrow) = Pr(\sigma)$.

In the following, $\hat{a}$ stands for a if $a \in GAct - \{\tau\}$ and for ε if $a = \tau$, $C \subseteq \mathcal{G}$, and $P \in \mathcal{G}$. Now let us consider $\mathit{Exec}(\tau^*\hat{a}, C)$ the set of executions σ' that lead to a term in C via a sequence belonging to the set of sequences $\tau^*\hat{a} \subseteq GAct^*$. Let $\mathit{Exec}(P, \tau^*\hat{a}, C) = \mathit{Exec}(\tau^*\hat{a}, C) \cap \mathit{Exec}(P)$, where $\mathit{Exec}(P)$ is the set of executions starting from P. The probability $\mathit{Prob}(P, \tau^*\hat{a}, C) = \mathit{Prob}(\mathit{Exec}(P, \tau^*\hat{a}, C))$ is defined as follows:

$$\mathit{Prob}(P, \tau^*\hat{a}, C) = \begin{cases} 1 & \text{if } a = \tau \wedge P \in C \\ \sum_{Q \in \mathcal{G}} \mathit{Prob}(P, \tau, Q) \cdot \mathit{Prob}(Q, \tau^*\hat{a}, C) & \text{if } a = \tau \wedge P \notin C \\ \sum_{Q \in \mathcal{G}} \mathit{Prob}(P, \tau, Q) \cdot \mathit{Prob}(Q, \tau^*\hat{a}, C) & \\ \quad + \mathit{Prob}(P, \hat{a}, C) & \text{otherwise} \end{cases}$$

The definition of weak bisimulation for *GRTS*s is similar to the ideas presented in [5], where the classical relation $\approx$ of [25] is replaced by the function Prob in order to consider the probability of reaching each state. Moreover, the authors of [5] describe an algorithm that computes weak bisimulation equivalence classes in time $\mathcal{O}(n^3)$ and space $\mathcal{O}(n^2)$.

Definition 3. *A relation $R \subseteq \mathcal{G} \times \mathcal{G}$ is a probabilistic weak bisimulation if $(P, Q) \in R$ implies for all $C \in \mathcal{G}/R$*

- $\mathit{Prob}(P, \tau^*\hat{a}, C) = \mathit{Prob}(Q, \tau^*\hat{a}, C)$ *for all* $a \in GAct$
- $\mathit{Prob}(P, a_*, C) = \mathit{Prob}(Q, a_*, C)$ *for all* $a_* \in RAct$

Two terms $P, Q \in \mathcal{G}$ are weakly bisimulation equivalent, denoted $P \approx_{PB} Q$, if there exists a weak bisimulation R containing the pair (P, Q). Note that if $P \in C$ then $\mathit{Prob}(P, \tau^*, C) = 1$. It is worth noting that the authors of [5] define both weak and branching bisimulation for fully probabilistic transition systems and show that these two relations coincide in such a probabilistic case (so that we can use $\tau^*\hat{a}$ instead of $\tau^*\hat{a}\tau^*$ in Definition 3). Moreover, it is easy to see that the definition of probabilistic weak bisimulation extends the classical notion of weak bisimulation. This means that the properties we define in Sect. 4 capture all the information flows which arise by analysing the nondeterministic behavior of a system via the classical approach.

4 Security Properties

In this section we present some information flow security properties, by extending the *noninterference* theory proposed in [12,13] in a probabilistic framework. In this paper we just consider the properties based on the weak bisimulation described in the previous section. As put in evidence in [13], the reason for resorting to this kind of equivalence stems from some lacks typical of other notions of equivalence, such as the trace equivalence. In general, the notion of bisimulation is finer than trace equivalence and is able to detect a higher number of

insecure behaviors (e.g. trace equivalence is not able to detect high level deadlocks). Moreover, as far as other notions of equivalence are concerned, in [13] the authors show that failure/testing equivalences are not interesting for systems with some high level loops or with τ loops.

4.1 Probabilistic Noninterference

We start by defining a probabilistic extension of the Bisimulation Strong Nondeterministic Noninterference (*BSNNI*), which says that a process P is *BSNNI* if the process $P\backslash AType_H$, where no high level activity is allowed, behaves like the process $P/AType_H$, where all the high level activities are hidden [12]. We call such an extended property Bisimulation Strong Probabilistic Noninterference (*BSPNI*).

Definition 4. $P \in BSPNI \Leftrightarrow P/AType_H \approx_{PB} P\backslash AType_H$

We point out that due to Definition 2, the low behavior of a process P that is *BSPNI* does not depend on a particular probability distribution of the hidden high level actions. This condition reinforces the security property and justifies our choice of hiding the reactive actions into τ actions, because it guarantees that the probability of a potential synchronization between a reactive high level action of P and a corresponding generative high level action of its environment cannot alter the probability distribution of the low level behavior of P.

For instance, let us consider the term $P \stackrel{\Delta}{=} l.P +^p h.l.P$, representing a process which can do a low level action preceded (or not) by a high level action. The high level does not interfere with the low level, because a low level user can just observe the action l with no information about the high level behavior; indeed we have $P/AType_H \approx_{PB} (l.P +^p h.l.P)/AType_H \approx_{PB} l.P +^p \tau.l.P \approx_{PB} l.P \approx_{PB} P\backslash AType_H$. The same example obtained by replacing h with h_* is still secure, as $P/AType_H \approx_{PB} P\backslash AType_H$ for any choice of p. This means that in $P \stackrel{\Delta}{=} l.P +^p h_*.l.P$ any potential interaction of P with its environment by means of a synchronization via a high level action h cannot alter the probability distribution of the event l and, as a consequence, the behavior of the system observable by a low level user. Now we show through some examples how the probabilistic version of noninterference can be employed in order to study the relation between the information flow and the probabilistic behavior of the system.

4.2 Probabilistic Measure of Insecure Nondeterministic Behaviors

Modeling the probabilistic behavior of a system allows us to give a more concrete, closer to the implementation description of the system, that may then reveal new aspects of the potential insecure information flows. More precisely, by introducing probabilities we can give a probabilistic measure to the insecure behaviors captured in the nondeterministic setting. Indeed, while in the nondeterministic case we can just deduce that a system is insecure because of an insecure information flow, in the probabilistic setting we can add that the system

reveals such an information flow with a certain probability. Hence, we can define different levels of security depending on the probability of observing insecure behaviors that an user of the system can tolerate.

For instance, let us consider an abstraction of an access monitor which handles read and write commands on a single-bit low-level variable. The low level read commands are represented by the reactive action $r0_*$ and $r1_*$, and the low level write commands are represented by the generative actions $w0$ and $w1$. We suppose that a high level user can explicitly change the value of the variable from 0 to 1, through the action h. The access monitor is described by the system[4] $P \triangleq h.P' +^q (r0_*.P + w1.P')$ with $P' \triangleq r1_*.P' + w0.P$. The high level user can interfere with any low level user by altering the low view of the system. This because intuitively a low level user can observe the sequence $r0_*.r1_*$, i.e. he first reads the value 0 and then the value 1, without observing a write command on the variable made by another low level user. Formally, we have that in $P/AType_H$ an internal τ transition (obtained by hiding the action h) leads to P' with probability q, whereas such a transition is not enabled in $P\backslash AType_H$. This situation is enough to construct a covert channel from high level to low level. However, such a system (with q close to 0 in the definition of P) and the secure version of the same system (obtained by removing the undesirable insecure component $h.P'$ from the definition of P) behave almost (up to small fluctuations) the same. From the user standpoint, a negligible risk (equal to a small ϵ) of observing an undesirable information flow may be tolerable, especially if the cost of a completely secure system is significantly greater than the cost of the system P. Formally, the notion of bisimulation can be enriched in order to tolerate ϵ-fluctuations which make the security condition less restrictive. As an example, in [7] the authors promote a pseudometric for probabilistic transition systems which quantifies the similarity of the behavior of probabilistic systems which are not bisimilar, and that can be easily exploited also in this context.

4.3 Capturing Probabilistic Information Flows

An important reason for extending the classical possibilistic theory of noninterference is that real systems may exhibit probabilistic covert channels that are not captured by standard nondeterministic security models. Potential insecure behaviors may be revealed by checking the probabilistic version of security properties such as the NNI. Now we show that our approach can be used in order to rule out such finer undesirable insecure behaviors.

An example of a probabilistic covert channel is inspired to the following insecure program proposed by Sabelfeld and Sands in [30,31]. They show that the program $Prog : \ h := h \, mod \, 100; (l := h +^{1/2} l := rand(99))$ has no information flow if we only consider the possible behaviors of the system, but the final value of l will reveal information about h when considering statistical inferences derived from the relative frequency of outcomes of repeated computations. Similarly, let us consider the process $P \triangleq (l' +^p l'') +^r h.(l' +^q l'')$, where a low level

[4] We omit the parameter of the probabilistic choice operator if it is not meaningful.

user can observe either an action l' or an action l'' (for the sake of simplicity we just consider two observable events, namely l' and l'', whereas in *Prog* a low level user can observe 100 different values). The component $(l' +^p l'')$ says that the probability distribution of the two low level actions is guided by parameter p (the counterpart in *Prog* is represented by the assignment $l := rand(99)$). The component $h.(l' +^q l'')$ says that, given an high level event of type h, the probability distribution of the two low level actions is guided by parameter q (the counterpart in *Prog* is represented by the assignment $l := h$). The nondeterministic version of this process is *BSNNI*, because the high level behavior does not alter what a low level user can observe, but in the probabilistic setting this is not the case (see Fig. 2). In fact, the probability of observing an event l' w.r.t. an event l'' changes depending on the behavior of the high level part exactly as the value of l (in program *Prog*) reveals h with a certain probability. In particular, we observe that P is *BSPNI* if and only if $p = q$, because in this case the high level behavior does not alter the probability distribution of the two low level events. Finally, it is worth noting that the insecure behavior of such an example is not captured by classical security properties such as *BNDC* [13] (and therefore the *lazy security* property of [28]), Strong *BNDC* [12], and Strong *BSNNI* [13].

A similar example is given by the system $Q \triangleq (l'_*.Q +^{p'} l''_*.Q) \|^{p''}_{\{l',l''\}} ((l'.Q +^p l''.Q) \|^r_{\bar{0}} h.(l'.Q +^q l''.Q))$ where parameters p' and p'' are not meaningful and the probability distribution of the low level actions depends on the behavior of the high level user. It is worth noting that Q is fully specified, in that it does not contain nondeterministic choices. As a consequence we can derive a Markov Chain from the *GRTS* underlying Q and, e.g., we can compute the throughput of the actions l' and l'' via standard techniques (see e.g. [21]).

As another example, let us consider the term $P \triangleq (l.\underline{0} +^p l.l'.\underline{0}) +^q l.h.l'.\underline{0}$ that is *BNDC* and, therefore, *BSNNI* (see [13]). In the probabilistic framework, the high level action h interferes with the probability of observing either a single l or the sequence $l.l'$. In particular in $P\backslash AType_H$ a low level user observes either the single event l with probability $p \cdot q + (1 - q)$ or the sequence $l.l'$ with probability $(1 - p) \cdot q$. On the other hand, in $P/AType_H$ a low level user observes either the single event l with probability $p \cdot q$ or the sequence $l.l'$ with probability $1 - p \cdot q$. As a consequence $P\backslash AType_H \not\approx_{PB} P/AType_H$ and the term P turns out to be insecure in the probabilistic setting.

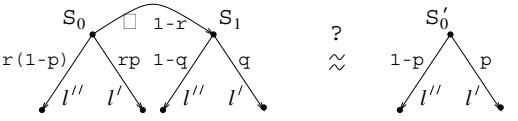


Fig. 2. $S_0 \approx_B S_1 \approx_B S'_0$, but $S_0 \not\approx_{PB} S_1 \not\approx_{PB} S'_0$, except for the case $p = q$

4.4 Nondeducibility on Composition

Sometimes the noninterference property is not enough to capture all the potential insecure behaviors of a system. For this reason, other additional properties have been suggested in order to overcome the lacks of the noninterference property. Among the different proposals, we consider the so called Non Deducibility on Composition (*NDC*), saying that the system behavior is invariant w.r.t. the composition with every high level user. Formally, P is *BNDC* if and only if $\forall\, \Pi \in \mathit{High\ Users},\ (P \mid \Pi)\backslash \mathit{AType}_H \approx_B P/\mathit{AType}_H$. Now we extend the notion of such a property in the probabilistic setting, by defining the Probabilistic Bisimulation *NDC* (for short, *PBNDC*).

Definition 5. $P \in \mathit{PBNDC} \Leftrightarrow P/\mathit{AType}_H \approx_{PB} ((P \parallel_S^p \Pi)/S)\backslash \mathit{AType}_H, \forall\, \Pi \in \mathcal{G}_H, p \in]0,1[, S \subseteq \mathit{AType}_H$.

It is worth noting that due to the particular parallel operator we adopt, it is necessary (i) to hide the high level actions belonging to S which succeed in synchronizing in $P \parallel_S^p \Pi$ and then (ii) to purge the system of the remaining high level actions. As in the nondeterministic framework, *PBNDC* is at least as strong as *BSPNI*.

Proposition 1. $\mathit{PBNDC} \subset \mathit{BSPNI}$

For instance, let us consider the process $P \triangleq l.\underline{0} +^p h.h.l.\underline{0}$. It is simple to see that this process is *BSPNI* since $P/\mathit{AType}_H \approx_{PB} l.\underline{0} +^p \tau.\tau.l.\underline{0} \approx_{PB} l.\underline{0}$ and $P\backslash \mathit{AType}_H = (l.\underline{0} +^p h.h.l.\underline{0})\backslash \mathit{AType}_H \approx_{PB} l.\underline{0}$. But, if we consider the process $\Pi \triangleq h_*.\underline{0}$ we get that $(P \parallel_{\{h\}}^q \Pi)/\mathit{AType}_H)\backslash \mathit{AType}_H \approx_{PB} l.\underline{0} +^p \tau.\underline{0} \not\approx_{PB} l.\underline{0} \approx_{PB} P/\mathit{AType}_H$.

The above example shows that *BSPNI* is not able to detect some potential deadlock due to high level activities, exactly as put in evidence in [13]. Therefore we resort to the *PBNDC* property in order to capture these finer undesirable behaviors. It is worth noting that, as reported in [13,14], the above definition of *PBNDC* is difficult to use because of the universal quantification on high level processes. For this reason, we propose the probabilistic version of the *SBSNNI* property (described in [13]), called Strong *BSPNI* (*SBSPNI*).

Definition 6. $P \in \mathit{SBSPNI} \Leftrightarrow \forall P' \in \mathit{Der}(P) : P' \in \mathit{BSPNI}$

The definition requires the system to be *BSPNI* in every derivative of P. Also in this framework the following property is valid.

Proposition 2. $\mathit{SBSPNI} \subset \mathit{PBNDC}$

Unfortunately, differently from *SBSNNI*, we have that *SBSPNI* is not compositional w.r.t. parallel composition. This because different processes composed in parallel can alter the probability distribution each other. As an example of such a lack, let us consider the system $P \triangleq l'.\underline{0} \parallel_\emptyset^p (l''.\underline{0} +^q h.l''.\underline{0})$. Both the left component and right one are *SBSPNI*. However, the possible execution of h

changes the probability of observing l', so from the low level user point of view, the probability distribution of the two observable sequences $l'.l''$ and $l''.l'$ is altered by the high level user, who hence can create a probabilistic covert channel. Obviously this would not be the case if $l' = l''$ (in such a case $P \stackrel{\Delta}{=} l.\underline{0} \parallel^{p}_{\emptyset} l.\underline{0} +^{q} h.l.\underline{0}$ is *SBSPNI*).

5 Conclusion

In this paper we have investigated the problem of extending the noninterference theory of [12,13] to the probabilistic case. In particular, the probabilistic version of the bisimulation noninterference and the bisimulation nondeducibility on compositions turned out to be able to capture information flows which would not be caught by the strongest classical nonprobabilistic notions of noninterference (e.g. the *SBNDC*, that is the most restrictive property proposed in [12]).

The aim of our approach is twofold. On the one hand, if the system is fully specified (i.e. the corresponding *GRTS* does not include nondeterministic choices), we can derive a Markov Chain and get performance measures of the system. On the other hand, the same model can be analysed in order to (i) give a probabilistic measure to the insecure nondeterministic behaviors, and (ii) reveal potential information flows that arise only if the model captures the probabilistic aspect of the system.

Among the possible extensions of such a work, it could be interesting (i) to extend the calculus with a notion of time in order to analyse on the same model the information flows deriving from both temporal and probabilistic behaviors, and (ii) to consider other notions of equivalences, such as testing equivalence for probabilistic processes. Moreover it may be meaningful to contrast our probabilistic notion of noninterference with other approaches cited in the related work. In particular, it would be most interesting to compare our ideas with those works (see, e.g., [27]) where weaknesses of security properties (like the BNDC) are pointed out. Finally, the next step of this study is the adoption of the proposed approach in the area of network security (see, e.g., [10]) for the analysis of cryptographic protocol properties in a probabilistic setting. In particular the aim is to extend to our setting the process algebraic approach of [15] proposed for analysing security protocols in the nondeterministic case.

Acknowledgements. This research has been funded by Progetto MURST and by a grant from Microsoft Research Europe.

References

1. A. Aldini, *"Probabilistic Information Flow in a Process Algebra"*, Tech. Rep. UBLCS-2001-06, University of Bologna, Italy, 2001
2. A. Aldini, M. Bernardo, R. Gorrieri, M. Roccetti, *"Comparing the QoS of Internet Audio Mechanisms via Formal Methods"*, ACM Transactions on Modelling and Computer Simulation, ACM Press, Vol. 11, N. 1, 2001

3. A. Aldini, M. Bravetti, *"An Asynchronous Calculus for Generative-Reactive Probabilistic Systems"*, in Proc. of 8th Int. Workshop on Process Algebra and Performance Modeling, pp. 119-138, 2000
4. J.C.M. Baeten, J.A. Bergstra, S.A. Smolka, *"Axiomatizing Probabilistic Processes: ACP with Generative Probabilities"*, in Information and Comp. 121:234-255, 1995
5. C. Baier and H. Hermanns, *"Weak Bisimulation for Fully Probabilistic Processes"*, in Proc. of CAV'97, Springer LNCS 1254, pp. 119-130, 1997
6. M. Bravetti, A. Aldini, *"Discrete Time Generative-reactive Probabilistic Processes with Different Advancing Speeds"*, Tech. Rep. UBLCS-2000-03, University of Bologna, Italy, to appear in Theoretical Computer Science, 2001
7. F. van Breugel, J. Worrell, *"Towards Quantitative Verification of Probabilistic Systems (extended abstract)"*, to appear in Proc. of 28th International Colloquium on Automata, Languages and Programming, Springer LNCS, 2001
8. D. Clark, C. Hankin, S. Hunt, R. Nagarajan, *"Possibilistic Information Flow is safe for Probabilistic Non-Interference"*, in Work. on Issues in the Theory of Security, 2000
9. A. Di Pierro, C. Hankin, H. Wiklicky, *"Probabilistic Security Analysis in a Declarative Framework"*, in Work. on Issues in the Theory of Security, 2000
10. A. Durante, R. Focardi, R. Gorrieri, *"A Compiler for Analysing Cryptographic Protocols Using Non-Interference"*, in ACM TOSEM, special issue on Software Engineering & Security, Volume 9(4), pages 489-530, 2000
11. N. Evans, S. Schneider, *"Analysing Time Dependent Security Properties in CSP Using PVS"*, in Proc. Symposium on Research in Computer Security, pp. 222-237, Springer LNCS 1895, 2000
12. R. Focardi, R. Gorrieri, *"A Classification of Security Properties"*, Journal of Computer Security, 3(1):5-33, 1995
13. R. Focardi, R. Gorrieri, *"The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties"*, IEEE Trans. Sof. Eng., 27:550-571, 1997
14. R. Focardi, R. Gorrieri, F. Martinelli, *"Information Flow Analysis in a Discrete-Time Process Algebra"*, in Proc. of 13th IEEE Computer Security Foundations Work., pp. 170-184, 2000
15. R. Focardi, R. Gorrieri, F. Martinelli, *"Non Interference for the Analysis of Cryptographic Protocols"*, in Proc. of 27th International Colloquium on Automata, Languages and Programming, Springer LNCS 1853, pp. 354-372, 2000
16. R.J. van Glabbeek, S.A. Smolka, B. Steffen, *"Reactive, Generative and Stratified Models of Probabilistic Processes"*, in Information and Comp. 121:59-80, 1995
17. J.A. Goguen, J. Meseguer, *"Security Policy and Security Models"*, in Proc. IEEE Symposium on Security and Privacy, pp. 11-20, IEEE CS Press, 1982
18. J.W. Gray III, *"Toward a Mathematical Foundation for Information Flow Security"*, Journal of Computer Security, 1:255-294, 1992
19. J.W. Gray III, P.F. Syverson, *"A Logical Approach to Multilevel Security of Probabilistic Systems"*, in Proc. IEEE Computer Society Symposium on Research in Security and Privacy, pp. 164-176, 1992
20. P.R. Halmos, *"Measure Theory"*, Springer-Verlag, 1950
21. R.A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971
22. J. Jürjens, *"Secure information flow for concurrent processes"*, in Proc. of Int. Conf. on Concurrency Theory, Springer LNCS 1877, pp. 395-409, 2000
23. J. McLean, *"Security Models and Information Flow"*, in Proc. of IEEE Symp. on Research in Security and Privacy, pp. 180-187, 1990

24. J.K. Millen, *"Hookup Security for Synchronous Machines"*, in Proc. of 3rd IEEE Computer Security Foundations Work., pp. 84-90, 1990
25. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989
26. A.W. Roscoe, *"CSP and Determinism in Security Modelling"*, in Proc. of IEEE Symposium on Security and Privacy, pp. 114-127, 1995
27. A.W. Roscoe, G.M. Reed, R. Forster, *"The successes and failures of behavioural models"*, in Millenial Perspectives in Computer Science, 2000
28. A. W. Roscoe, J.C.P. Woodcock, L. Wulf, *"Noninterference through Determinism"*, in Proc. European Symposium on Research in Computer Security, pp. 33-53, Springer LNCS 875, 1994
29. P.Y.A. Ryan, S. Schneider, *"Process Algebra and Noninterference"*, in Proc. of 12th IEEE Computer Security Foundations Work., pp. 214-227, 1999
30. A. Sabelfeld, D. Sands, *"A Per Model of Secure Information Flow in Sequential Programs"*, in Proc. of 8th European Symposium on Programming, Springer LNCS 1576, pg. 40-58, 1999
31. A. Sabelfeld, D. Sands, *"Probabilistic Noninterference for Multi-threaded Programs"*, in Proc. of 13th IEEE Computer Security Foundations Work., 2000
32. S. Schneider, *"Concurrent and Real-Time Systems: the CSP Approach"*, J. Wiley & Sons, Inc., 1999
33. J.T. Wittbold, D.M. Johnson, *"Information Flow in Nondeterministic Systems"*, in Proc. of IEEE Symp. on Research in Security and Privacy, pp. 144-161, 1990

Symbolic Computation of Maximal Probabilistic Reachability*

Marta Kwiatkowska, Gethin Norman, and Jeremy Sproston

School of Computer Science, University of Birmingham,
Birmingham B15 2TT, United Kingdom.
{M.Z.Kwiatkowska,G.Norman,J.Sproston}@cs.bham.ac.uk

Abstract. We study the *maximal reachability probability* problem for infinite-state systems featuring both nondeterministic and probabilistic choice. The problem involves the computation of the maximal probability of reaching a given set of states, and underlies decision procedures for the automatic verification of probabilistic systems. We extend the framework of symbolic transition systems, which equips an infinite-state system with an algebra of symbolic operators on its state space, with a symbolic encoding of probabilistic transitions to obtain a model for an infinite-state probabilistic system called a *symbolic probabilistic system*. An exact answer to the maximal reachability probability problem for symbolic probabilistic systems is obtained algorithmically via iteration of a refined version of the classical predecessor operation, combined with intersection operations. As in the non-probabilistic case, our state space exploration algorithm is semi-decidable for infinite-state systems. We illustrate our approach with examples of probabilistic timed automata, for which previous approaches to this reachability problem were either based on unnecessarily fine subdivisions of the state space, or which obtained only an upper bound on the exact reachability probability.

1 Introduction

Many systems, such as control, real-time, and embedded systems, give rise to *infinite-state* models. For instance, embedded systems can be modelled in formalisms characterised by a finite number of control states (representing a digital controller) interacting with a finite set of real-valued variables (representing an analogue environment). Motivated by the demand for automatic verification techniques for infinite-state systems, a number of results concerning the decidability of problems such as reachability, model checking and observational equivalence have been presented: isolated results concerning models such as timed automata [3], hybrid automata [2] and data independent systems [22] have been subject to unifying theories [1,10] and, in some cases, have provided the basis of efficient analysis tools, such as the timed automata model checker Uppaal [17].

In this paper, we consider a *probabilistic* model for infinite-state systems. For examples of infinite-state systems exhibiting probabilistic behaviour, consider

* Supported in part by the EPSRC grants GR/M04617 and GR/N22960.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 169–183, 2001.
© Springer-Verlag Berlin Heidelberg 2001

the real-time algorithm employed in the root contention protocol of IEEE1394 (FireWire) [20], probabilistic lossy channels [12] and open queueing networks [8]. Our system model also admits nondeterministic choice, which allows the modelling of asynchronous systems, and permits the *underspecification* of aspects of a system, including probabilistic attributes. We focus on the *maximal reachability probability* problem for probabilistic systems, concerning the computation of the maximal probability with which a given set of states is reachable. In the same way that reachability underlies the verification of temporal modalities in the non-probabilistic context, probabilistic reachability provides the foundation for probabilistic model checking of temporal modalities [6,5].

To reason about properties of infinite-state systems, an implicit, *symbolic* means to describe infinite state sets is required. The operations required on such state sets include boolean and *predecessor* operations, which together enable model checking of reachability properties by *backwards exploration* through the state space. Our first contribution concerns the extension of symbolic transition systems [10], which are infinite-state systems equipped with an algebra of such operations, with a (discrete) probabilistic transition relation. Observe that, in the context of quantitative reachability properties, it is not enough to know whether a state makes a transition to another, as encoded in the traditional predecessor operation: the probability of the transition must also be known. Our approach, which is specifically designed for the computation of maximal reachability probabilities, is to encode the transitions of a probabilistic system into a number of *types* (giving a family of *typed predecessor operations*), and the probabilistic branching of the system into a set of distributions over transition types called *distribution templates*. The resulting model, which consists of symbolic encodings of both states *and* transitions, together with an algebra of operations including the typed predecessor operations, is called a *symbolic probabilistic system*.

Our second contribution concerns the computation of the maximal reachability probability for certain classes of symbolic probabilistic systems by reduction to a finite-state problem. First, a state space exploration algorithm successively iterates typed predecessor and intersection operations, starting from the target set. The typed predecessor operations characterise the sets of states which can make a transition of a particular type to a previously generated set of states. To reason about the probabilistic branching structure of the system, we compute sets of states in which transitions of *multiple* types are enabled through intersections of state sets. If the state space exploration algorithm terminates, then a finite set of state sets is returned. Together, the transition types available in each of these state sets, and the distribution templates, allow us to construct a finite-state probabilistic system with an equal maximal reachability probability to that of the symbolic probabilistic system.

The state space analysis algorithm is closed under typed predecessor and intersection operations, and does not take *differences* between state sets; therefore, it differs from partition refinement algorithms. Our approach keeps the number of operations on the state space to a minimum, while retaining sufficient information for the computation of the maximal reachability probability.

In particular, noting that many symbolic approaches describe state sets in terms of constraints, our algorithm avoids propagating constraints arising from difference operations. To our knowledge, reasoning about reachability probabilities using a combination of predecessor and intersection operations is novel.

Related work. Approaches to infinite-state systems with discrete probability distributions include model checking methods for probabilistic lossy channel systems [12]. Two verification methods for probabilistic timed automata are presented in [15]. The first uses the "region graph" of [3] to compute *exact* reachability probabilities, but suffers from the state explosion problem (in particular, the size of the verification problem is sensitive to the magnitudes of the model's timing constraints, which is not true of our technique). The second uses forwards reachability, but, in contrast to our technique, only computes an *upper bound* on the actual maximal probability. Verification methodologies for infinite-state systems with *continuous* distributions are given in [4,7,14].

Plan of the paper. Section 2 defines symbolic probabilistic systems, and describes how they are used to represent probabilistic timed automata [15]. We present the semi-decidable algorithm to generate a finite-state representation of a symbolic probabilistic system in Section 3. Section 4 offers a critique of the analysis method, and suggests directions for future research.

2 Symbolic Probabilistic Systems

2.1 Preliminaries

A discrete probability *distribution* (*subdistribution*) over a finite set Q is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$ ($\sum_{q \in Q} \mu(q) \leq 1$). For a possibly uncountable set Q', let $\mathsf{Dist}(Q')$ ($\mathsf{SubDist}(Q')$) be the set of distributions (subdistributions) over finite subsets of Q'.

Recall that a *transition system* is a pair (S, δ) comprising a set S of states and a transition function $\delta : S \to 2^S$. A *state transition* $s \to t$ from a given state s is determined by a nondeterministic choice of target state $t \in \delta(s)$. In contrast, a (nondeterministic-) *probabilistic system* $\mathbb{S} = (S, \mathit{Steps})$ includes a probabilistic transition function $\mathit{Steps} : S \to 2^{\mathsf{Dist}(S)}$. A *probabilistic transition* $s \xrightarrow{\mu} t$ is made from a state $s \in S$ by first nondeterministically selecting a distribution μ from the set $\mathit{Steps}(s)$, and second by making a probabilistic choice of target state t according to μ, such that $\mu(t) > 0$. A *path* of a probabilistic system is a finite or infinite sequence of probabilistic transitions of the form $\omega = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} s_2 \cdots$. For a path ω and $i \in \mathbb{N}$, we denote by $\omega(i)$ the $(i+1)$th state of ω, and if ω is finite, $\mathit{last}(\omega)$ the last state of ω.

We now introduce *adversaries* which resolve the nondeterminism of a probabilistic system [21]. Formally, an *adversary* of $\mathbb{S}$ is a function A mapping every finite path ω to a distribution $\mu \in \mathsf{Dist}(S)$ such that $\mu \in \mathit{Steps}(\mathit{last}(\omega))$. Let $\mathit{Adv}_{\mathbb{S}}$ be the set of adversaries of $\mathbb{S}$. For any $A \in \mathit{Adv}_{\mathbb{S}}$, let Path^A_{ful} denote the

set of infinite paths associated with A. Then, in the standard way, we define the measure $Prob^A$ over $Path^A_{ful}$ [13].

The *maximal reachability probability* is the maximum probability with which a given set of states of a probabilistic system can be reached from a particular state. Formally, for the probabilistic system $\mathbb{S} = (S, Steps)$, state $s \in S$, and set $U \subseteq S$ of target states, the maximal reachability probability $ProbReach(s, U)$ of reaching U from s is defined as

$$ProbReach(s, U) \stackrel{\text{def}}{=} \sup_{A \in Adv_{\mathbb{S}}} Prob^A\{\omega \in Path^A_{ful} \mid \omega(0) = s \wedge \exists i \in \mathbb{N} \,.\, \omega(i) \in U\}.$$

The maximal reachability probability can be obtained as the solution to a linear programming problem in the case of finite probabilistic systems [6].

Computation of the maximal reachability probability allows one to verify properties of the form "with at least probability 0.99, it is possible to correctly deliver a data packet". By duality, it also applies to the validation of invariance properties such as "with probability at most 0.01, the system aborts". Furthermore, in the context of real–time systems, maximal reachability probability can be used to verify time-bounded reachability properties, also known as *soft* deadlines, such as "with probability 0.975 or greater, it is possible to deliver a message within 5 time units". For a more detailed explanation see [15].

2.2 Symbolic Probabilistic Systems: Definition and Intuition

Symbolic transition systems were introduced in [10] as (possibly infinite-state) transition systems equipped with *symbolic state algebras*, comprising a set of *symbolic states* (each element of which denotes a possibly infinite set of states), boolean, predecessor, emptiness and membership operations on symbolic states. In [10], classes of infinite-state systems for which a finitary structure can be identified by iteration of certain operations of the symbolic state algebra are defined, consequently highlighting the decidability of certain verification problems.

Symbolic probabilistic systems augment the framework of symbolic transition systems with (1) a probabilistic transition relation, (2) a symbolic encoding of probabilistic transitions, and (3) a redefined symbolic state algebra. Given the definition of probabilistic systems in the previous section, point (1) is self-explanatory. For point (2), note that information concerning probabilities is necessary for computation of maximal reachability probabilities. Let $s \to t$ be the state transition *induced* by a probabilistic transition $s \xrightarrow{\mu} t$ by abstracting the distribution μ from the transition. The symbolic representation consists of two steps: first, we encode state transitions induced by the probabilistic transitions of the system within a set of *transition types*. Second, we encode the probabilistic branching structure of the system, which is not represented in the set of transition types, by a set of *distribution templates*, which are distributions over the set of transition types. Finally, for point (3), the predecessor operation of a symbolic transition system is now replaced by a *family of predecessor operations*, each of which is defined according to the state transitions encoded by a transition type. This allow us to identify and reason about sets of states in which state

transitions of different transition types are available; in Section 3, we see that this characteristic is vital to identify a finitary structure on which the system's maximal reachability probability can be computed.

We now give the definition of symbolic probabilistic systems which generalise the symbolic transition systems of [10]. The definition of symbolic states R, extension function $\ulcorner\cdot\urcorner$, and symbolic operators $\mathsf{And}, \mathsf{Diff}, \mathsf{Empty}$ and Member agree with those given for symbolic transition systems, with the only difference being the typed predecessor operations. Conditions 1(a–c) have been added to represent probabilistic systems in such a way as to preserve maximal reachability probabilities, and are explained after the definition. In other contexts, different choices of symbolic representation and operations may be appropriate.

Definition of symbolic probabilistic systems. *A* symbolic probabilistic system $\mathbb{P} = (S, \mathit{Steps}, R, \ulcorner\cdot\urcorner, \mathcal{T}ra, \mathsf{D})$ *comprises: a probabilistic system* (S, Steps)*; a set of symbolic states* R*; an extension function* $\ulcorner\cdot\urcorner : R \to 2^S$*; a set of transition types* $\mathcal{T}ra$*, and, associated with each* $a \in \mathcal{T}ra$*, a transition function* $\delta_a : S \to 2^S$*; and a set of distribution templates* $\mathsf{D} \subseteq \mathsf{Dist}(\mathcal{T}ra)$*, such that the following conditions are satisfied.*

1. *For all states* $s \in S$*, let* $\mathcal{T}ra(s) \subseteq \mathcal{T}ra$ *be such that for any* $a \in \mathcal{T}ra$*:* $a \in \mathcal{T}ra(s)$ *if and only if* $\delta_a(s) \neq \emptyset$*. Then, for all* $t \in S$*:*
   a) *if* $a \in \mathcal{T}ra$ *and* $t \in \delta_a(s)$*, then there exists* $\mu \in \mathit{Steps}(s)$ *such that* $\mu(t) > 0$*;*
   b) *if* $\mu \in \mathit{Steps}(s)$*, then there exists* $\nu \in \mathsf{D}$ *and a vector of states* $\langle t_a \rangle_{a \in \mathcal{T}ra(s)} \in \prod_{a \in \mathcal{T}ra(s)} \delta_a(s)$ *such that:*

$$\sum_{a \in \mathcal{T}ra(s) \wedge t = t_a} \nu(a) = \mu(t);$$

   c) *if* $\nu \in \mathsf{D}$ *and* $\langle t_a \rangle_{a \in \mathcal{T}ra(s)}$ *is a vector of states in* $\prod_{a \in \mathcal{T}ra(s)} \delta_a(s)$*, then there exists* $\mu \in \mathit{Steps}(s)$ *such that:*

$$\mu(t) \geq \sum_{a \in \mathcal{T}ra(s) \wedge t = t_a} \nu(a).$$

2. *There exists a family of computable functions* $\{\mathsf{pre}_a\}_{a \in \mathcal{T}ra}$ *of the form* $\mathsf{pre}_a : R \to R$*, such that, for all* $a \in \mathcal{T}ra$ *and* $\sigma \in R$*:*

$$\ulcorner \mathsf{pre}_a(\sigma) \urcorner = \{ s \in S \mid \exists t \in \delta_a(s) \,.\, t \in \ulcorner \sigma \urcorner \} \,.$$

3. *There is a computable function* $\mathsf{And} : R \times R \to R$ *such that* $\ulcorner \mathsf{And}(\sigma, \tau) \urcorner = \ulcorner \sigma \urcorner \cap \ulcorner \tau \urcorner$ *for each pair of symbolic states* $\sigma, \tau \in R$*.*
4. *There is a computable function* $\mathsf{Diff} : R \times R \to R$ *such that* $\ulcorner \mathsf{Diff}(\sigma, \tau) \urcorner = \ulcorner \sigma \urcorner \setminus \ulcorner \tau \urcorner$ *for each pair of symbolic states* $\sigma, \tau \in R$*.*
5. *There is a computable function* $\mathsf{Empty} : R \to \mathbb{B}$ *such that* $\mathsf{Empty}(\sigma)$ *if and only if* $\ulcorner \sigma \urcorner = \emptyset$ *for each symbolic state* $\sigma \in R$*.*
6. *There is a computable function* $\mathsf{Member} : S \times R \to \mathbb{B}$ *such that* $\mathsf{Member}(s, \sigma)$ *if and only if* $s \in \ulcorner \sigma \urcorner$ *for each state* $s \in S$ *and symbolic state* $\sigma \in R$*.*

We proceed to describe transition types and distribution templates in greater depth.

Transition types. Recall that a transition type encodes a set of state transitions of a symbolic probabilistic system. Hence, for each transition type $a \in \mathcal{T}ra$ there is a transition relation $\delta_a : S \to 2^S$ encoding all of the state transitions of type a. This grouping is *not* necessarily a partition of the state transitions and a given state transition may correspond to more than one type. It follows from the lemma below that every probabilistic transition is represented by a state transition encoded in some transition type, and vice versa.

Lemma 1. *Let $\mathbb{P} = (S, Steps, R, \ulcorner \cdot \urcorner, \mathcal{T}ra, \mathsf{D})$ be a symbolic probabilistic system. For any $s, t \in S$: $\mu(t) > 0$ for some $\mu \in Steps(s)$ if and only if $t \in \delta_a(s)$ for some $a \in \mathcal{T}ra$.*

Distribution templates. Recall that we use the set of distribution templates to encode the actual probabilities featured in the system. Point 1(b) requires that the probabilistic branching structure of the system is represented in the distribution templates. Conversely, condition 1(c) expresses the fact that, in all states, for any transition encoded by a distribution template and transition type, there exists a system transition which assigns an equal or greater probability to all target states. This implies that there may be combinations of distribution templates and transition types which do not correspond to actual probabilistic transitions of the system. However, condition 1(c) together with 1(b) ensures that our model is nevertheless sufficient for the computation of the maximal reachability probability.

Example 1. Consider a system in which the state space takes the form of valuations of a single real-valued variable x. In state $s \in \mathbb{R}$, the variable x can be reset nondeterministically in the intervals (1,3) and (2,4), each with probability 0.5. Consider representing the system as a symbolic probabilistic system, where the set of symbolic states is the set of integer-bounded intervals of $\mathbb{R}$. The above behaviour can then be encoded by transition types a and b, such that $\delta_a(s) = (1, 3)$ and $\delta_b(s) = (2, 4)$, and the distribution template $\nu \in \mathsf{Dist}(\{a, b\})$ given by $\nu(a) = \nu(b) = 0.5$. Now, for any $s' \in (2, 3)$ there exists a distribution $\mu_{s'} \in Steps(s)$ which corresponds to moving from s and resetting x to s' with probability 1. For any such $\mu_{s'}$, the corresponding vector $\langle t_a, t_b \rangle$, described in point 1(b), is given by $t_a = t_b = s'$.

Finiteness of transition types and templates. Observe that the sets of transition types and distribution templates associated with a symbolic probabilistic system may be infinite. However, in Section 3, we restrict the analysis techniques to systems with finite sets of distribution templates and transition types. This assumption implies that the analysis method is appropriate for classes of infinite-state system exhibiting finite *regularity* in probabilistic transitions. For example, the probabilistic lossy channels of [12] cannot be modelled using a finite set of

distribution templates, because the probability of message loss varies with the quantity of data in the unbounded buffer.

2.3 Example: Probabilistic Timed Automata

In this section, we show that probabilistic timed automata [15] can be represented as symbolic probabilistic systems. We assume familiarity with the classical, non-probabilistic timed automaton model [3,11]. For an in-depth introduction to probabilistic timed automata, refer to [15].

Let $\mathcal{X}$ be a set of real-valued variables called *clocks*. Let $Zones(\mathcal{X})$ be the set of *zones* over $\mathcal{X}$, which are conjunctions of atomic constraints of the form $x \sim c$ and $x - y \sim c$, for $x, y \in \mathcal{X}$, $\sim \in \{<, \leq, \geq, >\}$, and $c \in \mathbb{N}$. A point $v \in \mathbb{R}^{|\mathcal{X}|}$ is referred to as a *clock valuation*. The clock valuation v *satisfies* the zone ζ, written $v \models \zeta$, if and only if ζ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value v_x from v.

A *probabilistic timed automaton* is a tuple $\mathsf{PTA} = (L, \mathcal{X}, inv, prob, \langle g_l \rangle_{l \in L})$, where: L is a finite set of *locations*; the function $inv : L \to Zones(\mathcal{X})$ is the *invariant condition*; the function $prob : L \to 2^{\mathsf{Dist}(L \times 2^{\mathcal{X}})}$ is the *probabilistic edge relation* such that $prob(l)$ is finite for all $l \in L$; and, for each $l \in L$, the function $g_l : prob(l) \to Zones(\mathcal{X})$ is the *enabling condition* for l. A state of a probabilistic timed automaton PTA is a pair (l, v) where $l \in L$ and $v \in \mathbb{R}^{|\mathcal{X}|}$. If the current state is (l, v), there is a nondeterministic choice of either letting *time pass* while satisfying the invariant condition $inv(l)$, or making a *discrete* transition according to any distribution in $prob(l)$ whose enabling condition $g_l(p)$ is satisfied. If the distribution $p \in prob(l)$ is chosen, then the probability of moving to the location l' and resetting all of the clocks in the set X to 0 is given by $p(l', X)$.

Example 2. Consider the probabilistic timed automaton PTA modelling a simple probabilistic communication protocol given in Figure 1. The nodes represent the locations: II (sender, receiver both idle); DI (sender has data, receiver idle); SI (sender sent data, receiver idle); and SR (sender sent data, receiver received). As soon as data has been received by the sender, the protocol moves to the location DI with probability 1. In DI, after between 1 and 2 time units, the protocol makes a transition either to SR with probability 0.9 (data received), or to SI with probability 0.1 (data lost). In SI, the protocol will attempt to resend the data after 2 to 3 time units, which again can be lost, this time with probability 0.05.

Before we represent a probabilistic timed automaton as a symbolic probabilistic system, we introduce the following definitions. Let $v \in \mathbb{R}^{|\mathcal{X}|}$ be a clock valuation: for any real $\eta \geq 0$, the clock valuation $v + \eta$ is obtained from v by adding η to the values of each of the clocks; and, for any $X \subseteq \mathcal{X}$, the clock valuation $v[X := 0]$ is obtained from v by resetting all of the clocks in X to 0. Now, for zone ζ and $\eta \geq 0$, let $\zeta + \eta$, be the expression in which each clock $x \in \mathcal{X}$ is replaced syntactically by $x + \eta$ in ζ, and let $[X := 0]\zeta$ be the expression in which each clock $x \in X$ is replaced syntactically by 0 in ζ. The set of *edges* of PTA, denoted

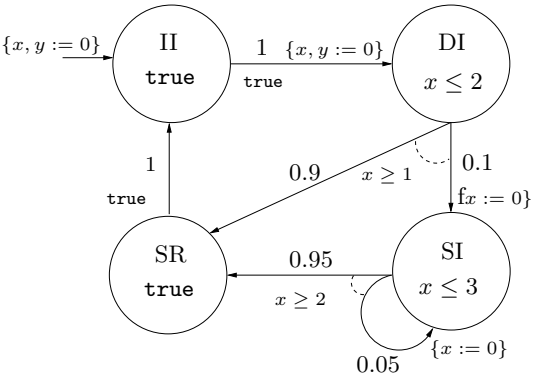


Fig. 1. A probabilistic timed automaton modelling a probabilistic protocol.

by $E_{\mathsf{PTA}} \subseteq L^2 \times 2^{\mathcal{X}} \times \mathit{Zones}(\mathcal{X})$, is defined such that $(l, l', X, \zeta) \in E_{\mathsf{PTA}}$ if and only if there exists $p \in \mathit{prob}(l)$ such that $g_l(p) = \zeta$ and $p(l', X) > 0$.

A probabilistic timed automaton $\mathsf{PTA} = (L, \mathcal{X}, \mathit{inv}, \mathit{prob}, \langle g_l \rangle_{l \in L})$ defines a symbolic probabilistic system $\mathbb{P} = (S, \mathit{Steps}, R, \ulcorner \cdot \urcorner, \mathcal{T}\mathit{ra}, \mathsf{D})$, where:

- (S, Steps) is the infinite-state probabilistic system obtained as a semantical model for probabilistic timed automata in the standard manner [15].
- The set R of symbolic states is given by $L \times \mathit{Zones}(\mathcal{X})$. The extension function $\ulcorner \cdot \urcorner$ is given by $\ulcorner (l, \zeta) \urcorner = \{(l, v) \in S \mid v \models \zeta\}$ for each $(l, \zeta) \in R$.
- The set of transition types $\mathcal{T}\mathit{ra}$ is the set of edges E_{PTA} plus the special type *time* such that, for any edge $(l', l'', X, \zeta') \in E_{\mathsf{PTA}}$, and state $(l, v) \in S$:

$$\delta_{\mathit{time}}(l, v) = \{(l, v + \eta) \mid \eta \geq 0 \wedge \forall\, 0 \leq \eta' \leq \eta \,.\, v + \eta' \models \mathit{inv}(l)\}$$

$$\delta_{(l', l'', X, \zeta)}(l, v) = \begin{cases} \{(l'', v[X := 0])\} & \text{if } l = l' \text{ and } v \models \zeta \\ \emptyset & \text{otherwise.} \end{cases}$$

- The set of distribution templates D is such that $\nu \in \mathsf{D}$ if and only if either:
  1. $\nu(\mathit{time}) = 1$, or
  2. there exists a location $l \in L$ and distribution $p \in \mathit{prob}(l)$ such that, for all transition types $a \in \mathcal{T}\mathit{ra}$:

$$\nu(a) = \begin{cases} p(l', X) & \text{if } a = (l, l', X, g_l(p)) \text{ for some } l' \in L \text{ and } X \subseteq \mathcal{X} \\ 0 & \text{otherwise.} \end{cases}$$

Given $(l, v) \in S$, the set $\delta_{\mathit{time}}(l, v)$ represents the set of states to which a time passage transition can be made, whereas $\delta_{(l', l'', X, \zeta)}(l, v)$ represents the unique state which is reached after crossing the edge denoted by (l', l'', X, ζ), provided that it is available, and the empty symbolic state otherwise. As time passage transitions are always made with probability 1, there exists a distribution template $\nu_{\mathit{time}} \in \mathsf{D}$, such that $\nu_{\mathit{time}}(\mathit{time}) = 1$; each of the other distribution templates in D is derived from a unique distribution of the probabilistic timed automaton.

For any symbolic state $(l, \zeta) \in R$, and any edge $(l', l'', X, \zeta') \in E_{\mathsf{PTA}}$, the typed predecessor operations are defined by:

$$\mathsf{pre}_{time}(l, \zeta) = (l, (\exists \eta \geq 0 \,.\, \zeta + \eta \wedge \forall\, 0 \leq \eta' \leq \eta \,.\, inv(l) + \eta'))$$
$$\mathsf{pre}_{(l', l'', X, \zeta')}(l, \zeta) = \begin{cases} (l', (\zeta' \wedge inv(l') \wedge [X := 0](\zeta \wedge inv(l)))) & \text{if } l = l'' \\ (l, \mathtt{false}) & \text{otherwise.} \end{cases}$$

Observe that these operations are defined in terms of pairs of locations and constraints on clocks. Note that by classical timed automata theory [11], for each $a \in \mathcal{T}ra$, the function pre_a is well defined and computable. Boolean operations, membership and emptiness are also well defined and computable for R. Both of the sets $\mathcal{T}ra$ and D are finite, which follows from the finiteness of L and $prob(l)$ for each $l \in L$.

Points 1(b) and 1(c) of the definition of symbolic probabilistic systems apply to probabilistic timed automata for the following reasons. As explained above, the distribution template ν_{time} encodes time passage transitions of the probabilistic system $(S, Steps)$ and conditions 1(b) and 1(c) follow trivially. The other transitions of PTA consist of choices of enabled distributions. Recall that edges of the probabilistic timed automaton are transition types. First consider condition 1(b): for any $l \in L$ and $p \in prob(l)$, there exists a distribution template $\nu \in \mathsf{D}$ assigning the same probability to the edges induced by p. Then, a probabilistic transition of $(S, Steps)$ corresponding to p will be encoded by this ν. For condition 1(c), recall that each $\nu \in \mathsf{D} \setminus \{\nu_{time}\}$ is derived from a particular $p \in prob(l)$ for some $l \in L$. Then, for the state $(l', v) \in S$, either $l' = l$ and $v \models g_l(p)$, and condition 1(c) follows as in the case of 1(b), or ν assigns probability 0 to all types in $\mathcal{T}ra(s)$, and hence any distribution available in this state will ensure the satisfaction of 1(c).

The translation method can be adapted to classes of *probabilistic hybrid automata* [18,19], which are hybrid automata [2] augmented with a probabilistic edge relation similar to that featured in the definition of probabilistic timed automata, given an appropriate set of symbolic states and algebra of operations. For example, a translation for *probabilistic linear hybrid automata* is immediate, given the above translation and the translation from non-probabilistic linear hybrid automata to symbolic transition systems of [10].

3 Maximal Reachability Probability Algorithm

We now present a *semi-decidable* algorithm (semi-algorithm) solving the maximal reachability probability problem for symbolic probabilistic systems. As mentioned in the previous section, we restrict attention to those symbolic probabilistic systems with *finite* sets of transition types and distribution templates. Note that, even for symbolic probabilistic systems within this class, the algorithm is not guaranteed to terminate.

Let $\mathbb{P} = (S, Steps, R, \ulcorner \cdot \urcorner, \mathcal{T}ra, \mathsf{D})$ be a symbolic probabilistic system such that the sets $\mathcal{T}ra$ and D are finite, and let $F \subseteq R$ be the target set of symbolic states which for which the maximal reachability probability is to be computed.

Symbolic semi-algorithm ProbReach
input: $(R, \mathcal{T}ra, \{\mathsf{pre}_a\}_{a\in\mathcal{T}ra}, \mathsf{And}, \mathsf{Diff}, \mathsf{Empty}, \mathsf{Member})$
target set $F \subseteq R$
$T_0 := F$;
$E := \emptyset$;
for $i = 0, 1, 2, \ldots$ **do**
$T_{i+1} := T_i$
for all $a \in \mathcal{T}ra \wedge \sigma \in T_i$ **do**
$T_{i+1} := \mathsf{pre}_a(\sigma) \cup T_{i+1}$
$T_{i+1} := \{\mathsf{And}(\mathsf{pre}_a(\sigma), \tau) \mid \tau \in T_{i+1}\} \cup T_{i+1}$ $(*)$
$E := \{(\mathsf{pre}_a(\sigma), a, \sigma)\} \cup E$
end for all
until $\ulcorner T_{i+1} \urcorner \subseteq \ulcorner T_i \urcorner$
$(T, E) := \mathsf{ExtendEdges}(T_i, E)$
return (T, E)

Procedure ExtendEdges
input: graph (T, E)
for all $\sigma \in T \wedge (\sigma', a, \tau) \in E$ **do**
if $\ulcorner \sigma \urcorner \subseteq \ulcorner \sigma' \urcorner$ **then**
$E := \{(\sigma, a, \tau)\} \cup E$
end if
end for all
return (T, E)

Fig. 2. Backwards exploration using predecessor and intersection operations

Our first task is to generate a finite graph (T, E), where $T \subseteq R$ and $E \subseteq T \times \mathcal{T}ra \times T$. The nodes of the graph (T, E) will subsequently form the states of a finite-state probabilistic system, and the edges will be used to define the required probabilistic transitions. The symbolic semi-algorithm ProbReach which generates the graph (T, E) is shown in Figure 2.

The algorithm ProbReach proceeds by successive iteration of predecessor and intersection operations. For each $i \in \mathbb{N}$ and for all currently generated symbolic states in the set T_i, the algorithm constructs the set T_{i+1} of symbolic states by adding to T_i the typed predecessors of the symbolic states in T_i, and the intersections of these predecessors with symbolic states in T_i. Furthermore, the edge relation E is expanded to relate the existing symbolic states to their newly generated typed predecessors. For any two symbolic states $\sigma, \tau \in R$, the test $\ulcorner \sigma \urcorner \subseteq \ulcorner \tau \urcorner$ is decided by checking whether $\mathsf{Empty}(\mathsf{Diff}(\sigma, \tau))$ holds. Then the termination test $\ulcorner T_{i+1} \urcorner \subseteq \ulcorner T_i \urcorner$ denotes the test $\{\ulcorner \sigma \urcorner \mid \sigma \in T_{i+1}\} \subseteq \{\ulcorner \sigma \urcorner \mid \sigma \in T_i\}$, which is decided as follows: for each $\sigma \in T_{i+1}$, check that there exists $\tau \in T_i$ such that both $\ulcorner \sigma \urcorner \subseteq \ulcorner \tau \urcorner$ and $\ulcorner \tau \urcorner \subseteq \ulcorner \sigma \urcorner$ [10].

If the outer **for** loop of the symbolic semi-algorithm ProbReach terminates, then we call the procedure ExtendEdges on the graph (T, E). Intuitively, for a

particular edge $(\sigma, a, \tau) \in E$, the procedure constructs edges with the transition type a and target symbolic state τ for all subset symbolic states of σ in T. Finally, observe that the set T is closed under typed predecessor and intersection operations. However, in a practical implementation of ProbReach, symbolic states encoding empty sets of states, and their associated edges, do not need to be added to the sets T and E respectively.

Remark 1 (termination of ProbReach*).* Termination of ProbReach is reliant on the termination of the outer **for** loop, because, if this terminates, T and E are finite, and hence the procedure ExtendEdges will also terminate. Observe that the inner **for** loop of the algorithm will not terminate if the set $\mathcal{T}ra$ is not finite. Now let $\preceq$ be a binary relation on the state space S of $\mathbb{P}$ such that $s \preceq t$ implies, for all $a \in \mathcal{T}ra$ and $s' \in \delta_a(s)$, there exists $t' \in \delta_a(t)$ such that $s' \preceq t'$. We call such a relation a *typed simulation*. Let $\approx$ be an equivalence relation on the state space S such that $s \approx t$ if there exists typed simulations $\preceq, \preceq'$ such that $s \preceq t$ and $t \preceq' s$. We call a relation such as $\approx$ a *typed mutual simulation*, and say $\approx$ has *finite index* if there are finitely many equivalence classes of $\approx$.

The arguments of [10] are adapted to show that ProbReach will terminate for any symbolic probabilistic system for which there exists a typed mutual simulation $\approx$ with finite index, given that the target set F is a set of equivalence classes of $\approx$. That is, we show that for all $\sigma \in T$, the set $\ulcorner\sigma\urcorner$ is a union of equivalence classes of $\approx$. This is achieved by proving by induction on $i \in \mathbb{N}$ that, for all $s, t \in S$ such that $s \preceq t$ for some typed simulation $\preceq$, if $\sigma \in T_i$ and $s \in \ulcorner\sigma\urcorner$, then $t \in \ulcorner\sigma\urcorner$. Probabilistic timed automata and probabilistic rectangular automata with two continuous variables exhibit such a relation, as indicated by [3] and [9] respectively.

If the semi-algorithm ProbReach terminates, the graph (T, E) is such that each symbolic state $\sigma \in T$ encodes a set of states of the symbolic probabilistic system $\mathbb{P}$, all of which can reach the target set F with positive probability. The following lemma asserts that the states encoded by the source of an edge in E are encoded by the appropriately typed predecessor of the edge's target symbolic state.

Lemma 2. *Let $\mathbb{P} = (S, Steps, R, \ulcorner\cdot\urcorner, \mathcal{T}ra, \mathsf{D})$ be a symbolic probabilistic system and let (T, E) be the graph constructed using the semi-algorithm* ProbReach*. For any transition type $a \in \mathcal{T}ra$, if $(\sigma, a, \tau) \in E$, then $\ulcorner\sigma\urcorner \subseteq \ulcorner\mathsf{pre}_a(\tau)\urcorner$.*

Next, we construct a finite-state probabilistic system, the states of which are the symbolic states generated by ProbReach, and the transitions of which are induced by the set of edges E and the finite set of distribution templates D. That is, we lift the identification of state transitions encoded in E to probabilistic transitions. We achieve this by grouping edges which have the *same* source symbolic state and which correspond to *different* transition types. Then a probabilistic transition of $\mathbb{Q}$ is derived from a distribution template by using the association between target symbolic states and the transition types of the edges in the identified group. Formally, we define a *sub-probabilistic system* $\mathbb{Q} = (T, Steps_{\mathbb{Q}})$, where $Steps_{\mathbb{Q}} : T \to 2^{\mathsf{SubDist}(T)}$ is the sub-probabilistic transition relation $Steps_{\mathbb{Q}}$

constructed as follows. For any symbolic state $\sigma \in T$, let $\pi \in \mathit{Steps}_{\mathbb{Q}}(\sigma)$ if and only if there exists a subset of edges $E_\pi \subseteq E$ and a distribution template $\nu \in \mathsf{D}$ such that:

1. if $(\sigma', a, \tau') \in E_\pi$, then $\sigma' = \sigma$;
2. if $(\sigma, a, \tau), (\sigma, a', \tau') \in E_\pi$ are distinct edges, then $a \neq a'$;
3. the set E_π is maximal;
4. for all symbolic states $\tau \in T$:

$$\pi(\tau) = \sum_{a \in \mathcal{T}ra \wedge (\sigma, a, \tau) \in E_\pi} \nu(a).$$

For any symbolic state $\sigma \in T$, any $\pi \in \mathit{Steps}_{\mathbb{Q}}(\sigma)$ may be a *sub-distribution*, as it is not necessarily the case that all of the transition types assigned positive probability by the distribution template associated with π are featured in the edges in E_π: some transition types may lead to states which cannot reach the target F. Note that the finiteness of the set D of distribution templates is required for the construction of the sub-probabilistic system $\mathbb{Q}$ to be feasible.

We now state the formal correctness of our algorithm (the proof can be found in [16]); that is, for any state $s \in S$ and symbolic state $\sigma \in T$ such that $s \in \ulcorner \sigma \urcorner$, the maximal reachability probability of $\mathbb{P}$ reaching the set $\ulcorner F \urcorner$ of states from the state s equals that of $\mathbb{Q}$ reaching the set F from σ.

Theorem 1. *If $\mathbb{Q} = (T, \mathit{Steps}_{\mathbb{Q}})$ is the sub-probabilistic system constructed using the algorithm* ProbReach*, with input given by the symbolic probabilistic system $\mathbb{P} = (S, \mathit{Steps}_{\mathbb{P}}, R, \ulcorner \cdot \urcorner, \mathcal{T}ra, \mathsf{D})$ and target set $F \subseteq R$, then for any state $s \in S$:*

$$\mathit{ProbReach}(s, \ulcorner F \urcorner) = \max_{\sigma \in T \wedge s \in \ulcorner \sigma \urcorner} \mathit{ProbReach}(\sigma, F) \ .$$

Recall from Section 2.1 that the maximal reachability probability for finite probabilistic systems can be computed using established methods [6].

We now describe a method which removes information from $\mathbb{Q}$ which is redundant to the computation of the maximal reachability probability.

Remark 2 (redundant conjunction operations). The purpose of the conjunction operation And in the algorithm ProbReach is to generate symbolic states for which multiple transition types are available. However, taking the conjunction of predecessors of transition types which are never *both* assigned positive probability by any distribution template does not add information concerning the probabilistic branching of the symbolic probabilistic system to $\mathbb{Q}$, and hence does not affect in the computation of the maximal reachability probability. To avoid taking such redundant conjunctions of state sets, we can replace the line marked $(*)$ in the semi-algorithm ProbReach with the following:

for all $\nu \in \mathsf{D}$ such that $\nu(a) > 0$ **do**
$T_{i+1} := \{\mathsf{And}(\mathsf{pre}_a(\sigma), \sigma') \mid (\sigma', b, \tau) \in \mathsf{relevant}(a, \nu, E)\} \cup T_{i+1}$
$E := \{(\mathsf{And}(\mathsf{pre}_a(\sigma), \sigma'), c, \tau) \mid (\sigma', b, \tau) \in \mathsf{relevant}(a, \nu, E) \ \wedge c \in \{a, b\}\} \cup E$
end for all

where $(\sigma, b, \tau) \in \mathsf{relevant}(a, \nu, E)$ if and only if $b \neq a$, $\nu(b) > 0$ and $(\sigma, b, \tau) \in E$.

Example 2 (continued). Say that we want to find the maximal probability of the probabilistic timed automaton of Figure 1 reaching the location SR, corresponding to correct receipt of a message, within 4 time units of the data arriving at the sender. Given that the target set F equals $\{(\mathrm{SR}, y < 4)\}$, application of ProbReach on the symbolic probabilistic system of this automaton results in the construction of the sub-probabilistic system in Figure 3. As suggested above, we do not consider symbolic states corresponding to empty sets of states. By classical probabilistic reachability analysis on this system, the maximal probability of reaching SR within 4 time units of the data arriving at the sender is 0.995.

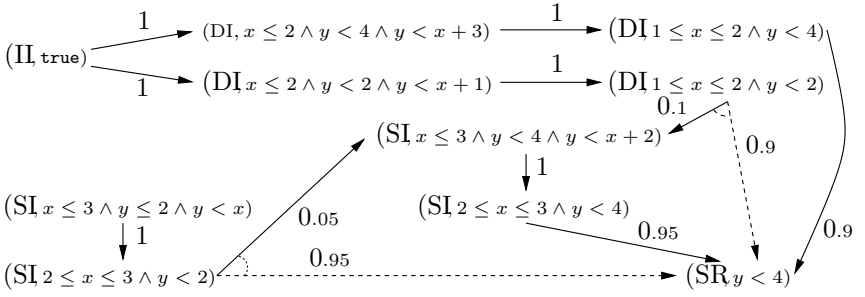


Fig. 3. The probabilistic system generated by ProbReach for the PTA in Figure 1.

The symbolic states and the solid edges are generated by the main loop of the algorithm ProbReach, while the dashed lines are added by the procedure ExtendEdges. For example, there is a solid edge corresponding to a particular transition type from the symbolic state $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 4)$ to the symbolic state $(\mathrm{SR}, y < 4)$. Then, as $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 2)$ is a subset of $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 4)$, the procedure ExtendEdges adds an extra edge from $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 2)$ to $(\mathrm{SR}, y < 4)$ of the same transition type. On inspection of Figure 1, and by the definition of the translation method for probabilistic timed automata to symbolic probabilistic systems, there exists a distribution template which assigns probability 0.9 and 0.1 to the transition types of the edges from $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 2)$ to $(\mathrm{SR}, y < 4)$, and to $(\mathrm{SI}, x \leq 3 \wedge y < 4 \wedge y < x + 2)$, respectively. Therefore, the distribution associated with the symbolic state $(\mathrm{DI}, 1 \leq x \leq 2 \wedge y < 2)$ shown in Figure 3 is constructed.

4 Conclusions

Recall that the state space exploration algorithm presented in Section 3 iterates predecessor and intersection operations; unlike a partition refinement algorithm, it does not perform *difference* operations. Our motivation is that state sets of many infinite-state systems, including timed and hybrid automata, are described by constraints. If difference operations are used when intersecting state sets, then constraints representing the states within the intersection, *and* the negation of these constraints, are represented, rather than just the former.

Note that the algorithm could be applied only to the portion of the state space which is reachable from initial states, thereby avoiding analysis of unreachable states. Furthermore, the practical implementation of our approach can be tailored to the model in question. For probabilistic timed automata, state sets and transitions resulting from time transitions do not need to be represented; instead, typed predecessors are redefined to reflect both time passage and edge transitions. Observe that the state space exploration technique presented here will only generate *convex* zones; non-convex zones are notoriously expensive in terms of space.

Our method extends to enable the verification of symbolic probabilistic systems against the existential fragments of probabilistic temporal logics such as PCTL [6,5], though at a cost of adding union and difference operations in order to cater for disjunction and negation. However, to enable the verification of full PCTL a solution to the *minimum* reachability probability problem is required.

Finally, we conjecture that the methods presented in this paper have significance for the verification of probabilistic hybrid and parameterised systems.

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS'96*, pages 313–321. IEEE Computer Society Press, 1996.
2. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.
5. C. Baier and M. Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
6. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. FSTTCS'95*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.
7. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Approximating labeled Markov processes. In *Proc. LICS 2000*, pages 95–106. IEEE Computer Society Press, 2000.
8. B. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach*. John Wiley and Sons, 1998.
9. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS'95*, pages 453–462. IEEE Computer Society Press, 1995.
10. T. A. Henzinger, R. Majumdar, and J.-F. Raskin. A classification of symbolic transition systems, 2001. Preliminary version appeared in *Proc. STACS 2000*, volume 1770 of LNCS, pages 13–34, Springer, 2000.
11. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

12. P. Iyer and M. Narasimha. Probabilistic lossy channel systems. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*, pages 667–681. Springer, 1997.
13. J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 2nd edition, 1976.
14. M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Verifying quantitative properties of continuous probabilistic timed automata. In *Proc. CONCUR 2000*, volume 1877 of *LNCS*, pages 123–137. Springer, 2000.
15. M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 2001. Special issue on ARTS'99. To appear.
16. M. Z. Kwiatkowska, G. Norman, and J. Sproston. Symbolic computation of maximal probabilistic reachability. Technical Report CSR-01-5, School of Computer Science, University of Birmingham, 2001.
17. P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.
18. J. Sproston. Decidable model checking of probabilistic hybrid automata. In *Proc. FTRTFT 2000*, volume 1926 of *LNCS*, pages 31–45. Springer, 2000.
19. J. Sproston. *Model Checking of Probabilistic Timed and Hybrid Systems*. PhD thesis, University of Birmingham, 2001.
20. M. I. A. Stoelinga and F. Vaandrager. Root contention in IEEE1394. In *Proc. ARTS'99*, volume 1601 of *LNCS*, pages 53–74. Springer, 1999.
21. M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. FOCS'85*, pages 327–338. IEEE Computer Society Press, 1985.
22. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. POPL'86*, pages 184–193. ACM, 1986.

Randomized Non-sequential Processes
(Preliminary Version)

Hagen Völzer[1,2⋆]

[1] Software Verification Research Centre, The University of Queensland, Australia
[2] Humboldt-Universität zu Berlin, Germany

Abstract. A non-sequential, i.e. "true concurrency", semantics for randomized distributed algorithms is suggested. It is based on Petri nets and their branching processes. We introduce *randomized Petri nets* and their semantics, *probabilistic branching processes*. As a main result, we show that each probabilistic branching process defines a unique canonical probability space. Finally, we show that the non-sequential semantics differs from the classical sequential semantics, modelling a new adversary, called the *distributed adversary*.

1 Introduction

A *randomized distributed algorithm* is a distributed algorithm where agents may flip coins during the execution of their programs. Randomized algorithms gained more and more attention in the last twenty years since they often solve problems simpler and more efficient than ordinary distributed algorithms. Some randomized algorithms solve problems which are known to be unsolvable by ordinary algorithms. Examples for such problems are symmetry breaking [10,12], choice coordination [20], and consensus [2]. For a survey of randomized algorithms, see [8].

To model randomized distributed algorithms, a formalism for modelling distributed algorithms has to be equipped with a coin flip construct. Existing formalisms include *probabilistic I/O automata* [24], *probabilistic programs* [19], *probabilistic concurrent processes* [1], a formalism suggested by Rao [21] which extends UNITY [6], and *probabilistic predicate transformers* [15] and its related models. Each of these models employs a sequential semantics where concurrency is expressed by nondeterminism, i.e. a run represents a total order of events.

A randomized distributed algorithm has three important features which must be accomodated by a model: (1) There is randomized choice. (2) There is nondeterministic choice. (3) There is (usually) no assumption on timing or synchrony. To our knowledge, there is no model based on Petri nets with all three features. We suggest a Petri net based model for randomized distributed algorithms in this paper which we call *randomized Petri nets*. This model provides a basis for

⋆ Postal address: Hagen Voelzer, SVRC, The University of Queensland, Qld 4072, Australia; e-mail: voelzer@svrc.uq.edu.au, Phone: +61 7 3365 1647; Fax: +61 7 3365 1533. This work was supported by DFG: Project "Konsensalgorithmen".

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 184–201, 2001.
© Springer-Verlag Berlin Heidelberg 2001

incorporation of randomized algorithms into existing Petri net based techniques for distributed algorithms (eg [22,23]).

Furthermore, we present a non-sequential semantics for randomized Petri nets where a run represents a partial order, viz the causal order, of events, thus providing a basis for investigation of the applicability of partial order verification methods [17] to randomized algorithms. Our semantics is directly based on branching processes of Petri nets [16,7] which means that we do not use a notion of global time or global state to construct the semantics. This absence of explicit global states results in the non-sequential semantics being weaker than the classical sequential semantics of randomized distributed algorithms: There are algorithms that terminate with probability 1 in the non-sequential semantics but not in the sequential semantics. As we will show, that is because, in our non-sequential semantics, a nondeterministic choice never depends on the outcome of a coin flip that happens concurrently which can be the case in the sequential semantics.

We proceed as follows. After recalling some preliminaries, we introduce randomized Petri nets in Sect. 3. The classical sequential semantics for randomized distributed algorithms is defined for randomized Petri nets in Sect. 4. In Sect. 5, we introduce the non-sequential semantics with the central notion of a *probabilistic branching process*. As a main result, we show that each probabilistic branching process defines a unique canonical probability space. Sect. 6 defines the probabilistic validity of temporal-logical formulas. Sect. 7 provides some examples which directly lead to a comparison of the sequential and the non-sequential semantics in Sect. 8, where we explain why the new semantics models a new adversary. A few remarks conclude the paper.

2 Preliminaries

This section collects some preliminaries which include Petri nets and their branching processes and a few notions from probability theory. For a motivation of these concepts we refer to the literature. In particular, we refer to [16] and [7] for branching processes. The reader who is familiar with these concepts may skip the corresponding paragraphs. The only definition deviating from the literature is the definition of a conflict on page 186.

We denote the set of natural numbers by $\mathbb{N}$ and the closed interval of real numbers between 0 and 1 by $[0,1]$. Let A be a set. The set of all subsets of A is denoted by 2^A. A *multiset* over A is a mapping $M : A \to \mathbb{N}$. We write $M[x]$ instead of $M(x)$ for the multiplicity of an element x in M. A multiset M is *finite* if $\sum_{x\in A} M[x]$ is finite. Inclusion and addition of multisets are defined elementwise, i.e. $M_1 \leq M_2$ if $\forall x \in A : M_1[x] \leq M_2[x]$ and $(M_1 + M_2)[x] = M_1[x] + M_2[x]$. If we have $M_1 \leq M_2$ then the difference $M_2 - M_1$ is also defined elementwise. A set $A' \subseteq A$ will be treated as a multiset over A by identifying it with its characteristic function.

Petri nets. A *Petri net* (or *net* for short) $N = (P, T, F)$ consists of two disjoint non-empty, countable sets P and T and a binary relation $F \subseteq (P \times T) \cup (T \times P)$.

Elements of P, T, and F are called *places*, *transitions*, and *arcs* of the net respectively. We graphically represent a place by a circle, a transition by a square, and an arc by an arrow between the corresponding elements. We write $x \in N$ for $x \in P \cup T$ where x is also called an *element* of N. For each element x of N, we define the *preset* of x by ${}^\bullet x = \{y \in N \mid (y,x) \in F\}$ and the *postset* of x by $x^\bullet = \{y \in N \mid (x,y) \in F\}$. The set of *minimal elements* of N and the set of *maximal elements* of N is defined by ${}^\circ N = \{x \in N \mid {}^\bullet x = \varnothing\}$ and $N^\circ = \{x \in N \mid x^\bullet = \varnothing\}$, respectively. For each element x of N we define the set of *predecessors* of x by $\downarrow x = \{y \in N \mid yF^+x\}$ where F^+ denotes the transitive closure of F. We restrict our attention to nets in which for each transition t, the preset ${}^\bullet t$ and the postset $t^\bullet$ are non-empty and finite[1]. Therefore, we have ${}^\circ N, N^\circ \subseteq P$.

A *marking* M of a net is a finite multiset over P. A marking is graphically represented by black tokens in the places of the net. A marking M is *safe*, if $\forall p \in P : M[p] \leq 1$. A transition t is *enabled* in a given marking M if ${}^\bullet t \leq M$. If t is enabled in a marking M_1 then t may occur, resulting in the *follower marking* $M_2 = (M_1 - {}^\bullet t) + t^\bullet$. This is denoted $M_1 \xrightarrow{t} M_2$.

A pair $\Sigma = (N, M^0)$ of a net N and a safe marking M^0 of N is called a *net system*. The marking M^0 is called *initial marking* of Σ. A set $C \subseteq T$ of transitions of a net such that $|C| > 1$ is called a *conflict* if $\bigcap_{t \in C} {}^\bullet t \neq \varnothing$. A conflict C is *maximal* if there is no conflict that contains C. A conflict C is called *free choice* if $\forall t \in C : |{}^\bullet t| = 1$.

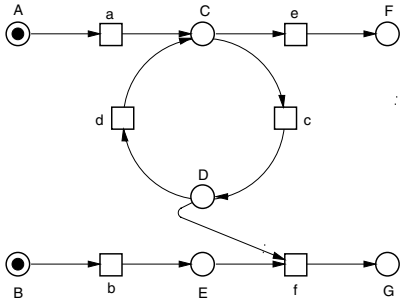


Fig. 1. A net system Σ_1

Fig. 1 shows a net with the safe marking $\{A, B\}$ –we write AB in the following for short. The sets $\{e, c\}$ and $\{f, d\}$ are maximal conflicts of Σ_1 where $\{e, c\}$ is free choice and $\{f, d\}$ is not free choice.

A *computation tree* ϑ of $\Sigma = (N, M^0)$ is a (possibly infinite) labelled rooted tree where each node is labelled with a marking of Σ and each edge is labelled with a transition of Σ such that (i) the root is labelled with M^0 and (ii) if edge (v_1, v_2) is labelled with transition t and node v_i is labelled with marking M_i for $i = 1, 2$ then we have $M_1 \xrightarrow{t} M_2$. There is a natural prefix order on the set of all computation trees of Σ and a maximal computation tree with respect to this prefix order which is unique up to isomorphism. Fig. 2 shows the maximal computation tree of Σ_1 . An *interleaved run* (*interleaving* for short) is a non-branching computation tree of Σ, i.e. a path in a computation tree of Σ starting in the root. An interleaving τ is *maximal* if there is no interleaving of

[1] This is a usual technical assumption to avoid some anomalies in the non-sequential semantics. See [4] and [7] for further explanation.

which τ is a prefix, i.e. if it is either infinite or its final marking does not enable any transition.

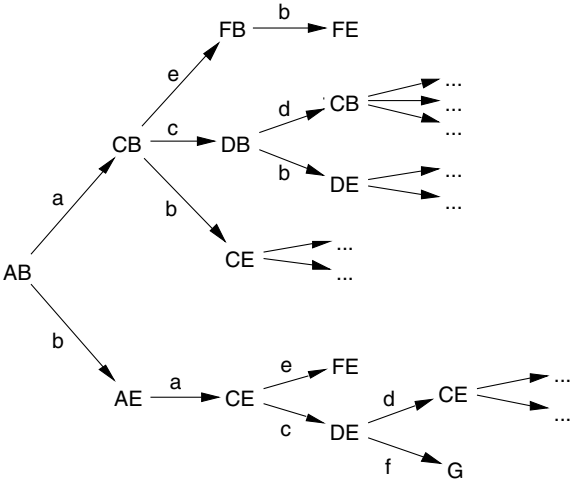


Fig. 2. The maximal computation tree of Σ_1

A set E of interleaved runs is called *interleaving property*. An interleaving property E such that $\rho \in E$ if and only if ρ satisfies some given formula Φ of a given linear-time temporal logic is called *temporal-logical interleaving property*.

Branching processes of Petri nets. Let N be a net. N is *acylic* if for each element x of N, we have $x \notin \downarrow x$ and N is *predecessor-finite* if for each element x of N, the set $\downarrow x$ is finite. Let $K = (B, E, \lessdot)$ be an acyclic, predecessor-finite net. A place $b \in B$ is called *condition* and a transition $e \in E$ is called *event*. Since K is acyclic, the transitive closure of $\lessdot$, denoted $\lessdot^+$, is a partial order, which we call *causal order*. We write $<$ instead of $\lessdot^+$ in the following. If we have $x_1 < x_2$ or $x_2 < x_1$ then we say x_1 and x_2 are *causally dependent*. If we have $x_1 < x_2$ or $x_1 = x_2$ then we write $x_1 \leq x_2$. Two elements are *in (extended) conflict*, denoted $x_1 \# x_2$, if there is a conflict $\{e_1, e_2\}$ such that $e_i \leq x_i$ for $i = 1, 2$. Two different elements are *concurrent*, denoted x_1 co x_2, if they are neither causally dependent nor in conflict.

A predecessor-finite, acyclic net K is called *occurrence net*, if (i) $^\circ K$ is finite, (ii) for each condition b of K we have $|{}^\bullet b| \leq 1$, and (iii) there is no event e of K such that $e \# e$. Let $\Sigma = (N, M^0)$ be a net system with $N = (P, T, F)$ and let $K = (B, E, \lessdot)$ be an occurrence net. Let $l : B \cup E \to P \cup T$ be a mapping such that $l(B) \subseteq P$ and $l(E) \subseteq T$. The pair $\pi = (K, l)$ is a *branching process* of Σ if (i) $l(^\circ K) = M^0$ and (ii) for each event e of K we have $l({}^\bullet e) = {}^\bullet l(e)$ and $l(e^\bullet) = l(e)^\bullet$.

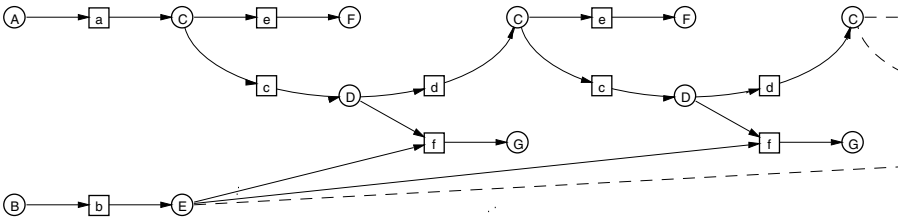


Fig. 3. The maximal branching process π of Σ_1

There is also a natural prefix order $\sqsubseteq$ on the set of branching processes of a net system Σ and a maximal branching process with respect to this prefix order which is unique up to isomorphism[2]. Fig. 3 shows the maximal branching process π of Σ_1 . A branching process of Σ without conflicts is called *non-sequential run*[3] (or *run* for short) of Σ. If a run ρ is a prefix of a branching process π then ρ is called a *run of* π. By a *maximal run* of π we mean a run of π which is maximal wrt to the prefix order. We denote the set of all runs of π, the set of all finite runs of π, and the set of all maximal runs of π by $\mathfrak{R}(\pi)$, $\mathfrak{R}_{\text{fin}}(\pi)$, and $\mathfrak{R}_{\max}(\pi)$, respectively. Note that, in contrast to interleaved runs, a proper prefix of a non-sequential run is not necessarily finite.

A *(non-sequential) temporal property* is a set of non-sequential runs. A temporal property E such that $\rho \in E$ if and only if ρ satisfies some given formula Φ of a given linear-time temporal logic is called *temporal-logical property.*

Probability. Let Ω be a non-empty set. A family of subsets $\mathcal{A} \subseteq 2^\Omega$ with $\Omega \in \mathcal{A}$ which is closed under complementation and finite union is called a *field* on Ω. If $\mathcal{A}$ is also closed under countable union then $\mathcal{A}$ is called a *σ-field* on Ω. The elements of $\mathcal{A}$ are called *measurable sets.* Let $\mathcal{A}$ be a σ-field on Ω. A mapping $P : \mathcal{A} \to [0,1]$ is a *probability measure* on $\mathcal{A}$ if $P(\Omega) = 1$ and for each countable pairwise disjoint family $\mathcal{F} \subseteq 2^\Omega$ we have:

$$P(\bigcup_{A \in \mathcal{F}} A) = \sum_{A \in \mathcal{F}} P(A). \tag{1}$$

The triple $(\Omega, \mathcal{A}, P)$ is called *probability space* if $\mathcal{A}$ is a σ-field on Ω and P is a probability measure on $\mathcal{A}$; σ-fields are closed under intersection. Therefore, for a given family $\mathcal{E} \subseteq 2^\Omega$, the smallest σ-field containing $\mathcal{E}$, denoted by $\sigma(\mathcal{E})$, is well-defined. Then, $\mathcal{E}$ is called the *generator* of $\sigma(\mathcal{E})$.

3 Randomized Petri Nets

This section introduces *randomized Petri nets*, our model for randomized distributed algorithms. We model a coin flip with two outcomes *head* and *tail* by a

[2] Engelfriet shows that the prefix order constitutes a complete lattice on the isomorphism classes of all branching processes of a net system [7].

[3] also called a *process* in the literature.

(a) Simple coin flip

(b) General coin flip

Fig. 4. Modelling coin flips

free choice conflict[4] as in Fig. 4(a). Every outcome has a probability of $\frac{1}{2}$. A transition modelling one outcome of a coin flip is called *probabilistic*. A probabilistic transition is graphically distinguished by the symbol %. A general coin flip with n outcomes is depicted in Fig. 4(b). The transitions $t_1, \ldots, t_n$ constitute a free choice conflict. Every t_i is equipped with a probability p_i, depicted at the arc leading to t_i, such that $\sum_{i=1}^{n} p_i = 1$.

A *randomized net* consists of a net where some transitions are distinguished as probabilistic and a mapping assigning a probability to each probabilistic transition.

Definition 1 (Randomized net). *Let $N = (P, T, F)$ be a net and let $T^{flip} \subseteq T$ be a set of distinguished transitions, called* probabilistic transitions*. A conflict C of N such that $C \subseteq T^{flip}$ is* probabilistic*. Furthermore, let $\mu : T^{flip} \to [0, 1]$ be a mapping such that $0 < \mu(t) < 1$ for all $t \in T^{flip}$. Then, the triple $\dot{N} = (N, T^{flip}, \mu)$ is called* randomized net *if*

(i) each probabilistic conflict is finite and free choice, and
(ii) for each maximal probabilistic conflict C we have

$$\sum_{t \in C} \mu(t) = 1 \tag{2}$$

The mapping μ is called local coin measure *of $\dot{N}$. A pair $\dot{\Sigma} = (\dot{N}, M^0)$ of a randomized net $\dot{N}$ and a safe marking M^0 of $\dot{N}$ is called* randomized net system.

Not every conflict of a randomized net system is probabilistic. Non-probabilistic conflicts are solved nondeterministically. Randomized net systems need nondeterminism to model nondeterminism of randomized distributed algorithms: There, we have usually no knowledge about the order of causally

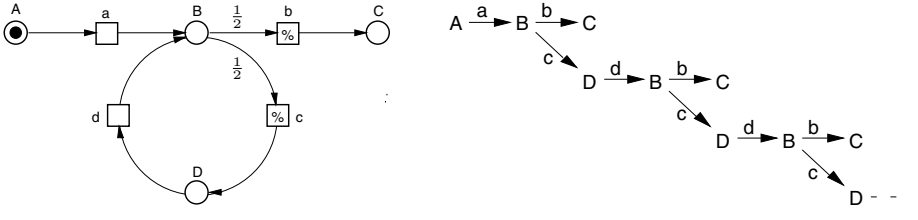
[4] Guided by our intuition of coin flips, we assign probabilities to free choice conflicts only. An extended free choice conflict can be refined to a free choice conflict by a simple, well-known construction (eg cf. [3]). This way, we may interprete also an extended free choice conflict as a coin flip.

independent events and the behaviour of the environment. Moreover, nondeterminism also models freedom of implementation. Note that there may be a non-probabilistic transition in conflict with a probabilistic transition. Such a transition would model the nondeterministic removal of the coin before it is flipped.

4 Probabilistic Computation Trees

This section presents the traditional sequential semantics of randomized distributed algorithms which we easily carry over from probabilistic programs [19] or probabilistic concurrent processes [1] to randomized net systems.

We begin with a randomized net system without concurrency and without nondeterminism. Fig. 5(a) shows such a system Σ_2. We expect the proposition "eventually C is marked", denoted $\Diamond C$, to be valid in Σ_2 with probability 1. The meaning of such a proposition is traditionally, eg in probabilistic transition systems, defined as follows.



(a) Σ_2 (b) ϑ

Fig. 5. A randomized net system and its unique maximal probabilistic computation tree

Fig. 5(b) shows the maximal computation tree ϑ of Σ_2. Since Σ_2 has neither concurrency nor nondeterminism, every branching in ϑ represents a coin flip, i.e. the local coin measure assigns a conditional probability to every branch such that the sum of the probabilities at every branching is 1. A computation tree where every branching represents a coin flip is *probabilistic*. A probability space $(\Omega, \mathcal{A}, P)$ is assigned to every probabilistic computation tree ϑ as follows. As a preparation, we assign a probability $p(\tau)$ to every finite interleaving $\tau = M_0 \stackrel{t_1}{\to} \ldots \stackrel{t_n}{\to} M_n$ by $p(\tau) = \prod_{i=1,\ldots,n} \mu(t_i)$ where, for $t_i \notin T^{\text{flip}}$, we set $\mu(t_i) = 1$. For Σ_2, we have $p(A \stackrel{a}{\to} B \stackrel{c}{\to} D \stackrel{d}{\to} B \stackrel{c}{\to} D) = \frac{1}{4}$.

Let $\Omega = \{\tau \mid \tau \text{ is a maximal interleaving of } \vartheta\}$. In the probability space on Ω to be constructed, the probability $p(\tau)$ is assigned to the set $K(\tau) = \{\tau' \in \Omega \mid \tau \text{ is a prefix of } \tau'\}$. Such a set $K(\tau)$ is called a *cone*. The set of all cones

$\mathcal{E} = \{K(\tau) \mid \tau$ is a finite prefix of $\vartheta\}$ constitutes the generator of the σ-field $\mathcal{A}$, i.e. $\mathcal{A} = \sigma(\mathcal{E})$. There is a unique probability space $(\Omega, \mathcal{A}, P)$ such that

$$P(K(\tau)) = p(\tau) \tag{3}$$

holds true for each finite interleaving τ. Each temporal-logical interleaving property E is measurable in this probability space. Then, we define E to be *valid with probability* p if $P(E) = p$. Then, $\Diamond\, C$ is actually valid with probability 1 in Σ_2.

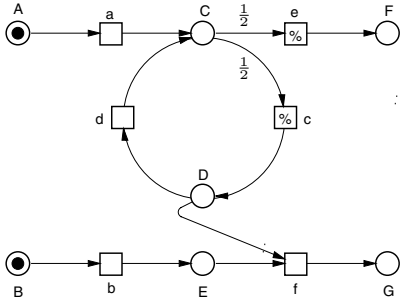


Fig. 6. Σ_3

Fig. 6 shows a randomized net system with concurrency as well as with nondeterminism. Therefore, the maximal computation tree ϑ of Σ_3 is not probabilistic. However we can choose maximal probabilistic computation trees from the prefixes of ϑ (Σ_3 has infinitely many maximal probabilistic computation trees). The choice of a maximal probabilistic computation tree and therewith of a probability space $(\Omega_\vartheta, \mathcal{A}_\vartheta, P_\vartheta)$ is nondeterministic. Therefore, a temporal-logical interleaving property E is defined to be *valid with at least probability* p if for each maximal probabilistic computation tree ϑ of $\dot{\Sigma}$ we have: $P_\vartheta(E) \geq p$. In Σ_3, $\Diamond\, F$ is valid with at least probability $\frac{1}{2}$ and $\Diamond(F \vee G)$ with at least probability 1, i.e. Σ_3 terminates with probability 1.

5 Probabilistic Branching Processes

In analogy to probabilistic computation trees, we introduce *probabilistic branching processes* in this section. A *probabilistic branching process* of a randomized net system $\dot{\Sigma}$ is a branching process of $\dot{\Sigma}$ such that all conflicts in it are probabilistic, i.e. there is no nondeterminism in a probabilistic branching process. As a main result, we show that there is a unique canonical probability space for each probabilistic branching process.

Definition 2 (Probabilistic branching process). *Let* $\dot{\Sigma} = ((N, T^{flip}, \mu), M^0)$ *be a randomized net system and* $\pi = (K, l)$ *be a branching process of* $\dot{\Sigma}$ *with events* E. *Let* $E^{flip} = \{e \in E \mid l(e) \in T^{flip}\}$ *denote the set of all* probabilistic events *of* π. *We carry over* μ *to probabilistic events:* $\mu : E^{flip} \to [0, 1]$ *by* $\mu(e) = \mu(l(e))$. *We call a probabilistic conflict* $C \subseteq E^{flip}$ *of* π complete *if it satisfies (2), i.e. if it contains all possible outcomes of the coin flip. Then,* π *is called* randomized branching process *of* $\dot{\Sigma}$ *if all maximal probabilistic conflicts*

of π are complete[5]*. A randomized branching process π of $\dot{\Sigma}$ is a* probabilistic branching process *of $\dot{\Sigma}$ if each conflict of π is probabilistic, i.e. π does not contain nondeterministic conflicts.*

Since every coin flip has only finitely many outcomes, every probabilistic branching process is finitely branching. Fig. 7 shows a finite, maximal probabilistic branching process of Σ_3 .

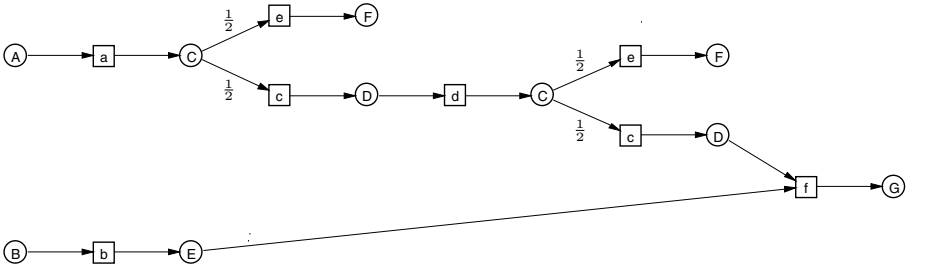


Fig. 7. A finite, maximal probabilistic branching process of Σ_3

We define a probability space for each probabilistic branching process of a randomized net system $\dot{\Sigma}$ which is derived from the local coin measure of $\dot{\Sigma}$. To do this, we define for a probabilistic branching process π the probability $p(\alpha)$ that a finite run α of π is realized in π. As in the sequential semantics, we assume the stochastic independence of all coin flips and thus define $p(\alpha)$ to be the product of the probabilities of all probabilistic events occurring in α: Let E^{flip} be the set of probabilistic events of α and μ be defined on E^{flip} as in Def. 2. We set $p(\alpha) = 1$ if $E^{\text{flip}} = \varnothing$ and

$$p(\alpha) = \prod_{e \in E^{\text{flip}}} \mu(e) \tag{4}$$

otherwise.

Theorem 1 (Probability space of a probabilistic branching process). *Let $\dot{\Sigma}$ be a randomized net system and π be a probabilistic branching process of $\dot{\Sigma}$. Let $\Omega = \mathfrak{R}_{\max}(\pi)$ and for every finite run α of π, let $K(\alpha) = \{\rho \in \mathfrak{R}_{\max}(\pi) \mid \alpha \sqsubseteq \rho\}$ be the set of maximal runs of π that have α as a prefix. Furthermore, let $\mathcal{E} = \{K(\alpha) \mid \alpha \in \mathfrak{R}_{\text{fin}}(\pi)\}$. Then, there exists a unique probability space $(\Omega, \mathcal{A}, P) = (\Omega_\pi, \mathcal{A}_\pi, P_\pi)$ such that $\mathcal{A} = \sigma(\mathcal{E})$ and for all finite runs α of π we have:*

$$P(K(\alpha)) = p(\alpha) \tag{5}$$

[5] Then, the triple $(K, E^{\text{flip}}, \mu)$ is a randomized (occurrence) net.

The proof is done in [26] and is omitted here[6]. The theorem cannot be derived from the corresponding theorem in the sequential semantics because a probabilistic branching process has more structure than its sequential counterpart, the probabilistic computation tree. The proof is technical and four pages long but requires only basic measure theory. It mainly exploits the lattice properties of branching processes. The proof rests upon the fact that all conflicts in a probabilistic branching process are free choice and finite.

We give now a proof sketch. A set $K(\alpha)$ for a finite run α of π is called a *cone*. We have to construct a probability measure P over $\sigma(\mathcal{E})$, which is already defined on $\mathcal{E}$, i.e. on all cones, by (5). For that we have to show that P can be extended to all complements of cones as well as to unions of cones and cone complements. The key step in the proof is the identification of a field on Ω, that contains $\mathcal{E}$ and that generates $\sigma(\mathcal{E})$. The field $\mathcal{M}$ that we are looking for contains exactly all finite unions of pairwise disjoint cones, i.e.

$$\mathcal{M} = \{ \bigcup_{A \in \mathcal{F}} A \mid \mathcal{F} \subseteq \mathcal{E} \text{ is finite and pairwise disjoint} \} \tag{6}$$

The mapping P can now be extended to the field $\mathcal{M}$ by (1). To show that this extension is well-defined requires the biggest effort in the proof. Further extension of P from $\mathcal{M}$ to the σ-field $\sigma(\mathcal{M}) = \sigma(\mathcal{E})$ is an application of the extension theorem, a standard theorem of measure theory. The uniqueness of the probability space follows from its construction.

6 Probabilistic Validity of Temporal Properties

In this section, we define what it means that a temporal property is valid with at least probability p in a given randomized net system. To do so, we have to restrict our attention to *measurable temporal properties*. A temporal property E is *measurable* in a randomized net system $\dot{\Sigma}$ if for every probabilistic branching process π of $\dot{\Sigma}$, E is measurable in the probability space associated with π.

Proposition 1. *Every temporal-logical property is measurable in each randomized net system.*

As in the sequential semantics, Proposition 1 is easily shown by induction over the structure of temporal-logical formulas. The proof is given in [26]. We define now the probabilistic validity of measurable temporal properties.

Definition 3 (Probabilistic validity). *Let $\dot{\Sigma}$ be a randomized net system and E be a temporal property that is measurable in $\dot{\Sigma}$. We say that E* is valid with at least probability *p if for every maximal probabilistic branching process π of $\dot{\Sigma}$, we have:*

$$P_\pi(E) \geq p \tag{7}$$

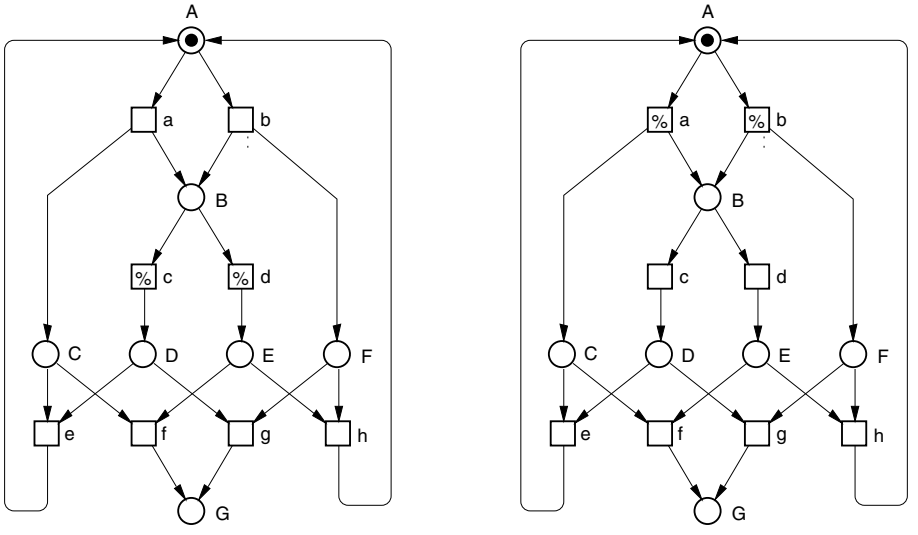
where P_π is the probability measure for π defined in Theorem 1.

[6] The proof will also be available in an extended version of this paper.

In Def. 3, we take maximality of non-sequential runs[7] as the basic liveness assumption of our semantics. To change to another notion of admissible runs, call a probabilistic branching process admissible if all its runs are admissible[8] and exchange "maximal" in Def. 3 by "admissible".

7 Examples

In the following, we omit the graphical representation of concrete probabilities because they do not play any role anymore. In each of the randomized net systems Σ_4 und Σ_5, depicted in Fig. 8, two free choice conflicts are solved, one after the other. Dependent on the solution of those conflicts, exactly one of the four transitions e, f, g, and h is then enabled.



(a) Σ_4 (b) Σ_5

Fig. 8. Two randomized net systems

In Σ_4, the conflict between c and d is probabilistic. Therefore, whether c or d occurs does not depend on whether a or b occurred before; Σ_4 terminates with probability 1, i.e. $\Diamond\, G$ is valid with probability 1 in Σ_4.

In Σ_5, not the conflict between c and d, but the conflict between a and b is probabilistic. The solution of the nondeterministic conflict between c and d may

[7] also called *progress*.

[8] or alternatively, if the set of all admissible runs has probability 1; the difference between these notions is discussed in [9].

depend on whether a or b occurred before: There is a maximal probabilistic branching process π of Σ_5 such that each occurrence of a is followed by an occurrence of c and each occurrence of b is followed by an occurrence of d. In this π, no run is finite, and therefore, Σ_5 does not terminate with probability 1.

Σ_6 und Σ_7 in Fig. 9 are similar to the randomized net systems in Fig. 8. Again, two free choice conflicts are solved but this time not sequentially but concurrently to each other. In Σ_6, both conflicts are probabilistic; Σ_6 terminates with probability 1. This is a prominent example for randomization in distributed algorithms: Both conflicts will eventually be solved *coordinated* to each other– without synchronization. Such iterated concurrent coin flips occur in many randomized distributed algorithms in the literature.

In Σ_7, only one of the two free choice conflicts is probabilistic; Σ_7 terminates with probability 1, but only in the non-sequential semantics. In contrast to Σ_6, Σ_7 does not terminate with probability 1 in the sequential semantics, because there is a probabilistic computation tree of Σ_7 where every maximal interleaving is infinite. This discrepancy between the sequential and the non-sequential semantics is further explained in the next section.

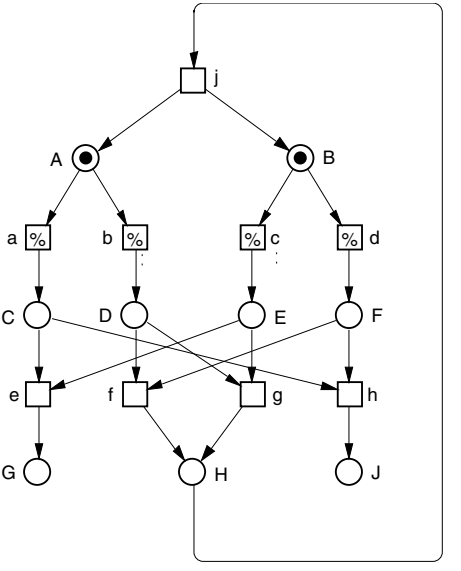
8 Non-sequential vs. Sequential Semantics

In this section, we explain the difference between the sequential semantics and the non-sequential semantics of randomized net systems by help of an example. Firm results, which will be provided in an extended version of this paper[9] require a notion of correspondence between interleaving properties and non-sequential temporal properties.

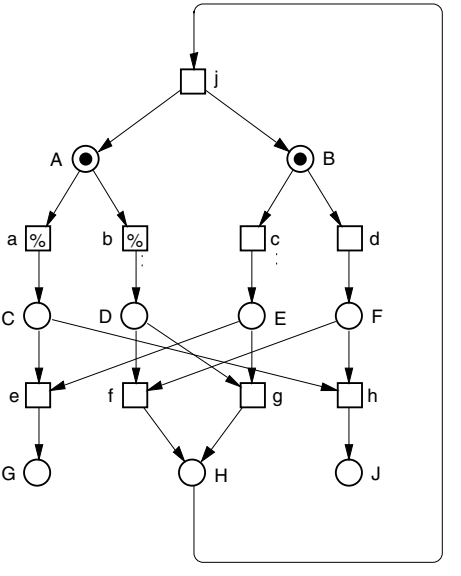
Sometimes, it is useful to adjust a semantics for randomized distributed algorithms. Differences between differently adjusted semantics are usually described by help of the notion of an *adversary* (sometimes also called *scheduler*). By that we imagine that we have two players, a coin flipper and an adversary, who interact to determine a run of a given randomized algorithm: For every finite interleaving $\tau = M_0, \ldots, M_n$, the adversary chooses either a non-probabilistic transition enabled in M_n–which then occurs–or a maximal probabilistic conflict– which is then resolved by the coin flipper by flipping a coin. Each behaviour of the adversary generates exactly one probabilistic computation tree.

An adjustment of the sequential semantics restricts the behaviour of an adversary. Usually, two kinds of restrictions are distinguished (cf.[24]): *execution-based adversaries* and *adversaries with partial on-line information*. For an execution-based adversary, we restrict the interleavings that are generated by the adver-

[9] Those firm results were obtained after the initial submission of this paper. It turns out that the non-sequential semantics is strictly weaker than the standard interleaving semantics for those interleaving properties that disregard non-causal ordering of events, so-called *equivalence robust* interleaving properties. An interleaving property E is *equivalence robust* if for all non-sequential runs ρ and each pair of sequentializations τ_1, τ_2 of ρ, we have: $\tau_1 \in E \Rightarrow \tau_2 \in E$. Those interleaving properties that we are usually interested in are equivalence robust.



(a) Σ_6



(b) Σ_7

Fig. 9. Coordination of concurrent decisions through randomization

sary. For example, we may postulate that the adversary is fair with respect to the solution of nondeterministic conflicts. For an adversary with partial on-line information, we restrict the knowledge the adversary is able to base its decisions upon. For example, we may postulate that the adversary does not know the entire history but only the current state. (If additionally, the adversary does not know the outcome of prior coin flips then the randomized net system in Fig. 8(b) terminates with probability 1 under this adversary.)

The following example suggests that the adversary associated with the non-sequential semantics can be viewed as an adversary with partial on-line information with respect to the adversary associated with the sequential semantics. This adversary, we call it the *distributed adversary*, has in contrast to the sequential adversary, no knowledge about non-causal order of events. In particular, the distributed adversary cannot depend the solution of a nondeterministic conflict on the outcome of coin flips that happen concurrently to that conflict, i.e. the distributed adversary can depend the solution of a nondeterministic conflict only on the causal history of that conflict. To see this, we consider Fig. 10.

Fig. 10 shows the randomized net system Σ_8 , its two probabilistic branching processes π_1 and π_2 and its six probabilistic computation trees ϑ_1 to ϑ_6. From each probabilistic branching process, we can derive a set of probabilistic computation trees by sequentialization. From π_1, we derive ϑ_1 and ϑ_2 and from π_2, we derive ϑ_3 and ϑ_4. Σ_8 shows that not every probabilistic computation tree can be derived from a probabilistic branching process: ϑ_5 and ϑ_6 can neither be assigned to π_1 nor to π_2. The computation trees ϑ_5 and ϑ_6 represent a behaviour of the sequential adversary where decisions of the adversary depend on the outcome of a concurrent coin flip. A similar behaviour of the sequential adversary prevents the termination of the randomized net system in Fig. 9(b) when c always occurs concurrently to b and d always occurs concurrently to a.

The characteristic property of the probability spaces of π_1, π_2 and of ϑ_1 to ϑ_4 which distinguishes them from the probability spaces of ϑ_5 and ϑ_6 is the following. Let α be a finite run of a probabilistic branching process π and let e be an event of π that is enabled at the end of α, i.e. α can be extended to some finite run β of π by appending e. Then, the occurrence of a concurrent event e' does not influence the conditional probability of e. More precisely, if e' is another event that is enabled in α such that e and e' are concurrent and we denote the resulting run when e' is appended to α by α' then we have

$$P(e|\alpha) = P(e|\alpha') \tag{8}$$

where $P(e|\alpha)$ denotes the conditional probability that e occurs provided that α is realized. Equation (8) holds in the probability space of each probabilistic branching process but not necessarily in the probability space of a probabilistic computation tree. That means that the order of concurrent events influences probability in the sequential semantics.

Note that it is easy to simulate the sequential semantics in the non-sequential semantics by sequentializing the randomized net system, i.e. by adding an initially marked place p to the net and connecting each transition t of the net with

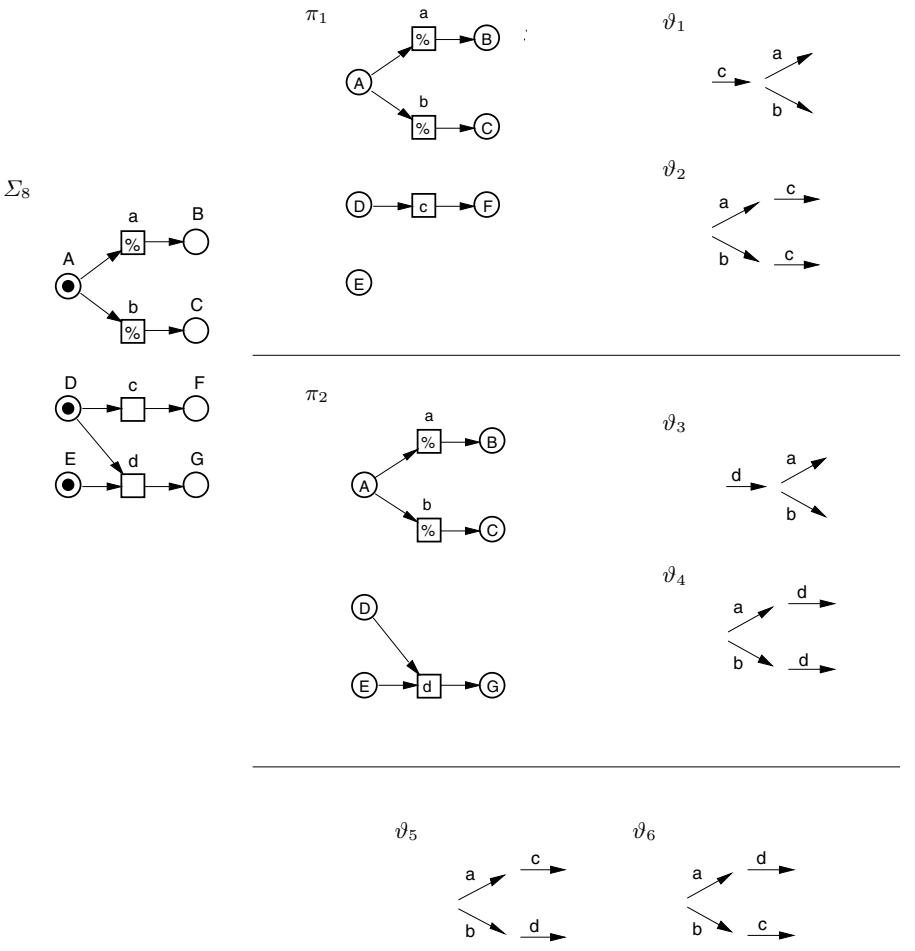


Fig. 10. Non-sequential vs. sequential semantics

p by a loop, i.e. adding (p, t) and (t, p) to F. That means that the non-sequential semantics discriminates more systems than the sequential semantics does. This is of course already the case for non-probabilistic systems.

9 Conclusion

Motivated by the application domain of randomized distributed algorithms, we restricted probabilistic conflicts to be free choice conflicts. As already noted, we could easily include probabilistic extended free choice conflicts[10] as well. However, if we want to go beyond extended free choice then we have to deal with

[10] A conflict C is *extended free choice* if for all $t_1, t_2 \in C$, we have ${}^\bullet t_1 = {}^\bullet t_2$.

confusion (cf. [25]) where the enabling of a conflict depends on the order of concurrent events. We would then inevitably need a notion of time, i.e. we have to order concurrent events, and we have to give up property (8) to construct a probability space. Then we would come closer to models motivated by performance analysis such as stochastic Petri nets [14] where we also find models combining causality and probability (eg [5]).

As the distributed adversary[11] is weaker than the general sequential adversary, it might admit more efficient distributed algorithms to solve a given synchronization- or coordination problem. This conjecture will be subject of future work. To assume a distributed adversary rather than a sequential adversary may result in a more faithful model in many cases. However, we must not introduce new causality when implementing an algorithm which was proven correct with respect to the non-sequential semantics–unless we take some measures such as encryption to hide the knowledge to the adversary that is provided by the new causality.

The adoption of verification techniques developed for the sequential semantics such as *extreme fairness* [18], *α-fairness* [13], fairness for labels [1], and *computable fairness* [11] to the non-sequential semantics is straight-forward because these techniques do not depend on specific properties of interleavings. However, these adoptions will reflect the discrepancy between the two semantics.

As suggested in this paper, the new approach raises further questions towards further research such as

- Is the non-sequential semantics robust with respect to refinement of actions as the non-sequential semantics of non-probabilistic systems is?
- Can we adopt partial order verification methods to randomized Petri nets?

Finally, since the non-sequential semantics strictly separates nondeterminism, randomization, and concurrency, we hope to better understand the subtle interaction of these three components in randomized distributed algorithms which is often blamed for the difficulty to verify these algorithms. This hope is already substantiated as we were able to prove, with the new semantics, two new impossibility results with respect to the power of randomized distributed algorithms in [26] where one of these results was discovered during the development of the semantics. These results will be subject of a forthcoming paper.

Acknowledgement. We thank Stefan Haar for substantial comments on earlier versions of this material.

References

1. Christel Baier and Marta Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66:71–79, 1998.

[11] Actually, we get a class of distributed adversaries since we can further restrict the knowledge of the distributed adversary.

2. Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the* 2^{nd} *Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983.
3. Eike Best. Structure theory of Petri nets: the free choice hiatus. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 167–205. Springer, 1987.
4. Eike Best and César Fernández. *Nonsequential Processes*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
5. Ed Brinksma, Joost-Pieter Katoen, Rom Langerak, and Diego Latella. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):552–565, 1995.
6. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
7. Joost Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.
8. Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On randomization in sequential and distributed algorithms. *ACM Computing Surveys*, 26(1):7–86, March 1994.
9. Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, July 1983.
10. Alon Itai and Michael Rodeh. Symmetry breaking in distributive networks. In *Proc. 22nd Annual Symposium on Foundations of Computer Science*, pages 150–158. IEEE, 1981.
11. Manfred Jaeger. Fairness, computable fairness, and randomness. In *Proc. 2nd PROBMIV Int. Workshop on Probabilistic Methods in Verification*. Technical Report CSR-99-8 School of Computer Science, University of Birmingham, 1999.
12. Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the* 4^{th} *Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1985.
13. Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In *Proc. Workshop on Logics of Programs*, volume 193 of *LNCS*, 1985.
14. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. Wiley, 1995.
15. Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
16. Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13:85–108, 1981.
17. Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann (eds). *Partial Order Methods in Verification, DIMACS Workshop, Proceedings*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1996.
18. Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proc. 15th Annual Symposium on Theory of Computing (STOC)*, pages 278–290. ACM, 1983.
19. Amir Pnueli and Lenore D. Zuck. Probabilistic verification. *Information and Computation*, 103:1–29, 1993.
20. Michael O. Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.
21. Josyula Rao. Reasoning about probabilistic algorithms. In *Proceedings of the* 9^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 247–264. ACM, 1990.

22. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998.
23. Wolfgang Reisig, Ekkart Kindler, Tobias Vesper, Hagen Völzer, and Rolf Walter. Distributed algorithms for networks of agents. In *Lectures on Petri Nets II: Applications*, volume 1492 of *LNCS*, pages 331–385. Springer, 1998.
24. Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. Technical report mit/lcs/tr-676, MIT, Laboratory for Computer Science, June 1995.
25. Einar Smith. On the border of causality: contact and confusion. *Theoretical Computer Science*, 153:245–270, 1996.
26. Hagen Völzer. *Fairneß, Randomisierung und Konspiration in Verteilten Algorithmen*. PhD thesis, Humboldt-Universität zu Berlin, Institut für Informatik, February 2001. in German, available via
http://dochost.rz.hu-berlin.de/dissertationen/voelzer-hagen-2000-12-08.

Liveness and Fairness in Process-Algebraic Verification

Antti Puhakka and Antti Valmari

Tampere University of Technology, Software Systems Laboratory,
PO Box 553, FIN-33101 Tampere, FINLAND,
{anpu,ava}@cs.tut.fi

Abstract. Although liveness and fairness have been used for a long time in classical model checking, with process-algebraic methods they have seen far less use. One reason for this is that most well-known process-algebraic theories such as CSP and CCS have limited capability for handling liveness properties. In this article we discuss the problems and possibilities of liveness and fairness in process algebra. We show that most well-known semantic equivalences do not preserve enough fairness-related information and that fairness properties are difficult to combine with the bottom-up compositionality of process algebra. However, we also establish positive results for a useful subset of fairness properties. We develop a method that does not assume new fairness-related constructs or rules for processes, but uses the standard LTS model. We demonstrate the method by removing livelocks from a communication protocol.

1 Introduction

In the verification of concurrent systems it is often important to show that the system eventually performs some desired task. Such properties are called *liveness* properties [1,15]. For proving liveness properties some *fairness assumptions* [2, 11,16,17] usually have to be added to the system, meaning that the system is not allowed to continuously favour some choices at the expense of others.

Within classical model checking [6,24] liveness and fairness have been used in one form or another for quite some time. However, in the context of *process-algebraic* methods [19,25] they have seen relatively little use. One reason for this is that with most well-known process-algebraic semantic models it is difficult to express liveness properties. For example, the weak bisimilarity of *CCS* [19] ignores divergences (livelocks) completely, and the *CSP* model [25] does not preserve any information on a system after it has performed a divergence trace. One problem seems to be combining liveness and fairness with bottom-up type of *compositionality*. That kind of compositionality is an important advantage offered by process-algebraic approaches for attacking *the state-explosion problem* (see e.g. [27]).

A variant of CSP called *CFFD* (*Chaos-Free Failures Divergences*) [29] has been previously developed which preserves information even after the execution of a divergence trace, and is therefore suitable for expressing liveness properties.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 202–217, 2001.
© Springer-Verlag Berlin Heidelberg 2001

CFFD covers the properties expressible in linear-time temporal logic [18,21] without the nextstate-operator (where the state-based logic is interpreted in an action-based setting), as well as deadlocks [14]. It is also well-suited for typical process-algebraic verification methods [26].

However, it is not clear how fairness assumptions should be used within the process-algebraic approach. [20] and [7] describe methods where certain types of fairness assumptions can be added to processes in a CCS-like setting. In the former, process expressions can be augmented with the ω-regular set of infinite sequences of actions it is allowed to execute. In the latter, states of a labelled transition system (LTS) can be marked as Büchi states, and only those infinite executions are considered where some Büchi state is visited infinitely often.

Approaches based on some finite representation of fairness assumptions have two levels: the ordinary operational and the additional fairness/liveness. This creates the problem that the two levels can be in conflict with each other: the fairness level rules out an infinite execution, while the operational level prevents the process from stopping or choosing other actions. Furthermore, this can happen even when the process is a composition of "healthy" subprocesses that are able to execute within their allowed sequences (see Section 3).

In this paper we use a CSP-like process algebra with LTSs as models of processes. LTSs are a simple, well-understood and widely accepted formalism for describing the behaviour of concurrent processes. Therefore, we take the view here that ordinary LTSs should be sufficient for describing process behaviour, whether that behaviour is "fair" or not. An LTS that has been formed from a finite LTS by adding a fairness assumption is typically infinite. However, this is often true only of an intermediate system, and when the LTS is placed in a larger context the result is finite. The (partial) solution we will develop later allows us to avoid the construction of the intermediate infinite system.

It should also be noted that our approach and the above-mentioned approaches based on finite representations are not mutually exclusive. In the future we may be able to use, say, Büchi-type fair LTSs as a finite notation for well-defined infinite ordinary LTSs. Then we could prove how the finite representations behave with respect to parallel composition etc., by using this connection.

Other works dealing with fairness and process algebra include [2,4,5,8,9,10, 12,13]. Most of these deal with *global* fairness assumptions, intended to capture the idea that all processes get execution time. All these and [7,20] are discussed in more detail in a longer version of this paper [23].

After reviewing the basic definitions in Section 2, we will in Section 3 consider a (hypothetical) operator that is used to add fairness assumptions to systems. We describe the properties we believe such an operator should have in order to be meaningful in a compositional context. We then show that most well-known semantic models are partly incompatible with these requirements, and also that the requirements impose limitations on the allowed fairness properties and process contexts. However, in Section 4 we are able to establish positive results for a small but useful subclass of fairness constraints. The approach is based on ordinary LTSs and can be carried through on the denotational level within

the CFFD-semantic model. In Section 5 we demonstrate the approach on the alternating bit protocol example, and in Section 6 we present some conclusions.

2 Background

The behaviour of a process consists of executing *actions*. There are two kinds of actions: *visible* and *invisible*. Invisible actions are denoted with a special symbol τ. The behaviour of a process is represented as a *labelled transition system*. It is a directed graph whose edges are labelled with action names, with one state distinguished as the initial state.

Definition 1. *A* labelled transition system*, abbreviated* LTS*, is a four-tuple* $(S, \Sigma, \Delta, \hat{s})$*, where*

- S *is the set of* states*,*
- Σ*, the* alphabet*, is the set of the* visible actions *of the process; we assume that* $\tau \notin \Sigma$*,*
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ *is the set of* transitions*, and*
- $\hat{s} \in S$ *is the* initial state*.*

We use $s -a\!\!\rightarrow s'$ as an abbreviation for $(s, a, s') \in \Delta$, and this is extended in the obvious way to $s -\sigma\!\!\rightarrow s'$ and $s -\xi\!\!\rightarrow$, where σ is a finite and ξ a finite or infinite sequence of actions. Let $restr(\sigma, A)$ denote the result of removal of all actions from σ that are not in A. We write $s =\rho\!\!\Rightarrow s'$ iff there is σ such that $s -\sigma\!\!\rightarrow s'$ and $\rho = restr(\sigma, \Sigma)$. $s =\rho\!\!\Rightarrow$ is defined similarly.

Let A^* denote the set of finite and A^ω that of infinite sequences of elements of a set A. The empty sequence is denoted with ε. We need the following semantic sets extracted from an LTS. A *trace* of an LTS is the sequence of visible actions generated by any finite execution that starts in the initial state. An infinite execution that starts in the initial state generates either an *infinite trace* or a *divergence trace*, depending on whether the number of visible actions in the execution is infinite. The *stable failures* describe the ability of the LTS to refuse actions after executing a particular trace.

Definition 2. *Let* $L = (S, \Sigma, \Delta, \hat{s})$ *be an LTS.*

- $Tr(L) = \{\, \sigma \in \Sigma^* \mid \hat{s} =\sigma\!\!\Rightarrow \,\}$ *is the set of the* traces *of* L*.*
- $Inftr(L) = \{\, \xi \in \Sigma^\omega \mid \hat{s} =\xi\!\!\Rightarrow \,\}$ *is the set of the* infinite traces *of* L*.*
- $Divtr(L) = \{\, \sigma \in \Sigma^* \mid \exists s : \hat{s} =\sigma\!\!\Rightarrow s \wedge s -\tau^\omega\!\!\rightarrow \,\}$*, where* τ^ω *denotes the infinite sequence of* τ*-actions, is the set of the* divergence traces *of* L*.*
- $Sfail(L) = \{\, (\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : \hat{s} =\sigma\!\!\Rightarrow s \wedge \forall a \in A \cup \{\tau\} : \neg(s -a\!\!\rightarrow) \,\}$ *is the set of the* stable failures *of* L*.*

The *parallel composition operator* defined below forces precisely those component processes to participate in the execution of a visible action that have that action in their alphabets. The invisible action is always executed by exactly one component process at a time. We first define the product of LTSs as the LTS

that satisfies the above description and has the Cartesian product of component state sets as its set of states, and then define parallel composition by picking the part of the product that is reachable from the initial state of the product.

Definition 3. *Let* $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$ *and* $L_2 = (S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ *be LTSs. Their* product *is the LTS* $(S', \Sigma, \Delta', \hat{s})$ *such that the following hold:*

- $S' = S_1 \times S_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $((s_1, s_2), a, (s_1', s_2')) \in \Delta'$ *if and only if either*
  - $[\, a \in (\Sigma_1 \cup \{\tau\}) - \Sigma_2$ *and* $(s_1, a, s_1') \in \Delta_1$ *and* $s_2' = s_2\,]$, *or*
  - $[\, a \in (\Sigma_2 \cup \{\tau\}) - \Sigma_1$ *and* $(s_2, a, s_2') \in \Delta_2$ *and* $s_1' = s_1\,]$, *or*
  - $[\, a \in \Sigma_1 \cap \Sigma_2$ *and* $(s_1, a, s_1') \in \Delta_1$ *and* $(s_2, a, s_2') \in \Delta_2\,]$.
- $\hat{s} = (\hat{s}_1, \hat{s}_2)$

The parallel composition $L_1 || L_2$ *is the LTS* $(S, \Sigma, \Delta, \hat{s})$ *such that*

- $S = \{\, s \in S' \mid \exists \sigma \in \Sigma^* : \hat{s} =\sigma\Rightarrow s \,\}$
- $\Delta = \Delta' \cap (S \times (\Sigma \cup \{\tau\}) \times S)$

The *hiding* operator converts given visible actions into τ-actions and removes them from the alphabet.

Definition 4. *Let* $L = (S, \Sigma, \Delta, \hat{s})$ *be an LTS and* A *any set of action names. Then* **hide** A **in** L *is the LTS* $(S, \Sigma', \Delta', \hat{s})$ *such that the following hold:*

- $\Sigma' = \Sigma - A$
- $(s, a, s') \in \Delta'$ *if and only if*
  $a = \tau \wedge \exists b \in A : (s, b, s') \in \Delta$, *or* $a \notin A \wedge (s, a, s') \in \Delta$.

An important property of an equivalence is that when a component process in a system is replaced by an equivalent process, the system will remain equivalent to the original one. This is formally captured by the *congruence* property.

Definition 5. *An equivalence "*$\simeq$*" is a* congruence *with respect to a process operator* $op(L_1, \ldots, L_n)$ *iff* $L_1 \simeq L_1' \wedge \cdots \wedge L_n \simeq L_n'$ *implies* $op(L_1, \ldots, L_n) \simeq op(L_1', \ldots, L_n')$.

We now define the CFFD semantic model and equivalence, which will be our main equivalence notion in this article.

Definition 6. *Let* L *and* L' *be LTSs with the same alphabet.*

- *The* CFFD model *of* L *is the 3-tuple* $(Sfail(L), Divtr(L), Inftr(L))$
- $L \simeq_{\mathsf{CFFD}} L' \iff$
  $[\, Sfail(L) = Sfail(L') \wedge Divtr(L) = Divtr(L') \wedge Inftr(L) = Inftr(L')\,]$

The traces are not included in the model because they can be determined from *Sfail* and *Divtr* (see e.g. [29]). It should be noted that when certain process-algebraic operators are used, a component called *stability* must be included in the CFFD-model. This one bit of information tells whether or not there are τ-transitions from the initial state of the LTS. However, with hiding and parallel composition this component is not needed, so we will not use it here.

CFFD-equivalence is a congruence with respect to parallel composition and hiding, as shown in [29], for example.

3 LTSs, Temporal Logic, and Fairness Operators

The desired properties of reactive and concurrent systems are often expressed by using *linear temporal logic* [18,21]. We next present a straightforward adaptation of the logic to our process-algebraic framework.

Definition 7. *A* formula *is generated by the grammar* $\psi ::= true \mid a \mid En(a) \mid \neg\psi \mid \psi \vee \psi \mid \psi \,\mathcal{U}\, \psi$, *where* a *is an action. We also use the following derived formulae:* $\psi \wedge \phi \equiv \neg(\neg\psi \vee \neg\phi)$, $\psi \Rightarrow \phi \equiv \neg\psi \vee \phi$, $\Diamond\psi \equiv true \,\mathcal{U}\, \psi$ *and* $\Box\psi \equiv \neg\Diamond\neg\psi$.

Definition 8. *Let* $L = (S, \Sigma, \Delta, \hat{s})$ *be an LTS. The set of the* infinite executions *of* L *is* $infex(L) = \left\{ s_0 a_1 s_1 a_2 \ldots \;\middle|\; \hat{s} = s_0 \wedge \forall i \geq 1 : s_{i-1} -a_i\rightarrow s_i \right\}$.

Below we will use the following notation: if $\eta = s_0 a_1 s_1 a_2 \ldots$ is an infinite execution, then $acts(\eta) = a_1 a_2 \ldots$ and $\eta^i = s_i a_{i+1} s_{i+1} a_{i+2} \ldots$.

Definition 9. *Let* $L = (S, \Sigma, \Delta, \hat{s})$ *be an LTS and* $\eta = s_0 a_1 s_1 a_2 \ldots$ *an infinite execution of* L. *Then*

- $L, \eta \models true$
- $L, \eta \models a$ *iff* $a_1 = a$
- $L, \eta \models En(a)$ *iff* $s_0 -a\rightarrow$ *(i.e., iff* $\exists s \in S : (s_0, a, s) \in \Delta$*)*
- $L, \eta \models \neg\psi$ *iff not* $L, \eta \models \psi$
- $L, \eta \models \psi \vee \phi$ *iff* $L, \eta \models \psi$ *or* $L, \eta \models \phi$
- $L, \eta \models \psi \,\mathcal{U}\, \phi$ *iff* $\exists j \geq 0 : L, \eta^j \models \phi \wedge \forall k : 0 \leq k < j : L, \eta^k \models \psi$

The properties of reactive systems are usually divided into *safety* and *liveness* properties [1,15]. Safety properties express requirements of the form "nothing bad must ever happen". The violation of a safety property can always be detected in a finite execution. Liveness properties express requirements of the form "something good must eventually happen", and the violation of a liveness property can only be detected in an infinite execution.

Fairness properties are liveness properties which state that certain actions are not infinitely overtaken by other actions in the choices the system makes. Two well-known classes of fairness properties are *weak fairness* and *strong fairness*. Weak fairness with respect to action a means that if a is from some point on continuously enabled, then it must be eventually executed. This can be expressed by the formula $\Diamond\Box En(a) \Rightarrow \Box\Diamond a$. Strong fairness means that if the action is from some point on enabled infinitely often, then it must be eventually executed. This can be expressed by $\Box\Diamond En(a) \Rightarrow \Box\Diamond a$.

It is customary in the verification of liveness properties to assume that the system satisfies some fairness constraint. A fairness constraint is a fairness property that is assumed, rather than proved, of the system. It formalises the idea that the underlying system is fair towards processes or choices.

Let us assume that there is some fairness constraint ϕ that we would like to express within our process-algebraic framework. We would like to have a

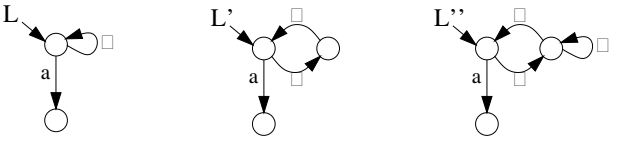


Fig. 1. The processes L, L' and L''

corresponding "fairness operator" Φ_ϕ, that given an LTS L produces a new LTS L' whose finite behaviour (safety properties) is like that of L, but whose infinite executions fulfill the given fairness constraint. More precisely, we want all traces ($Tr(L)$) and stable failures ($Sfail(L)$) to stay the same (so that deadlocks are not affected), and exactly those infinite traces ($Inftr(L)$) and divergence traces ($Divtr(L)$) to remain that are created by some infinite execution in compliance with ϕ. These requirements are stated formally in the following.

Definition 10. *An operator Φ_ϕ is a* fairness operator *for the formula ϕ iff for every LTS $L = (S, \Sigma, \Delta, \hat{s})$ each of the following holds:*

- $Tr(\Phi_\phi L) = Tr(L)$
- $Sfail(\Phi_\phi L) = Sfail(L)$
- $Divtr(\Phi_\phi L) = \left\{ \sigma \in \Sigma^* \;\middle|\; \exists \eta \in infex(L) : L, \eta \models \phi \wedge restr(acts(\eta), \Sigma) = \sigma \right\}$
- $Inftr(\Phi_\phi L) = \left\{ \xi \in \Sigma^\omega \;\middle|\; \exists \eta \in infex(L) : L, \eta \models \phi \wedge restr(acts(\eta), \Sigma) = \xi \right\}$

It is important to notice that this does not yet *define* a fairness operator, we have just stated desired properties of a (hypothetical) operator. An obvious requirement is also that any equivalence we use should be a congruence with respect to the fairness operator.

However, it turns out that the above properties are not easy to achieve. Consider the three LTSs L, L' and L'' in Figure 1. These are all CFFD-equivalent; the same holds for most process-algebraic semantic models. If we apply weak fairness towards a, i.e. $\phi(a) \equiv \Diamond\Box En(a) \Rightarrow \Box\Diamond a$, the divergence in the initial state of L disappears. However, a is not continuously enabled in L', so the fairness assumption does not remove the divergence there. Thus, $\varepsilon \notin Divtr(\Phi_{\phi(a)} L)$, but $\varepsilon \in Divtr(\Phi_{\phi(a)} L')$, and the results are not equivalent. We can try using strong fairness instead, because this forces execution of a even in L'. However, by comparing L and L'' we can similarly show that strong fairness leads to CFFD-different results. The conclusion from this counter-example is thus the following:

Proposition 1. *"$\simeq_{\mathsf{CFFD}}$" cannot be a congruence with respect to a fairness operator $\Phi_{\phi(a)}$ for the formula $\phi(a) \equiv \Diamond\Box En(a) \Rightarrow \Box\Diamond a$ or $\Box\Diamond En(a) \Rightarrow \Box\Diamond a$.*

This can also be formulated in terms of the CSP [25] and other similar equivalences. For instance, $L \simeq_{\mathsf{CSP}} L'' \simeq_{\mathsf{CSP}} \Phi_{\phi(a)} L'' \simeq_{\mathsf{CSP}} CHAOS \not\simeq_{\mathsf{CSP}} \Phi_{\phi(a)} L$.

Clearly the reason why most models are not congruences with respect to the fairness operator is because they do not preserve enough information on the enabledness of actions during infinite executions. A notable exception is

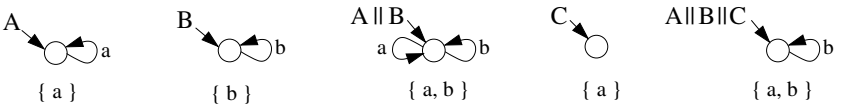


Fig. 2. A, B, C and their parallel compositions, with alphabets shown

the strong bisimilarity of [19], but as is well known, it treats invisible events no differently from visible events, and thus does not allow us to abstract them away.

Furthermore, we would like to make one more "soundness" requirement for the hypothetical fairness operator. This is because the fairness operator would typically be applied to some process L (e.g. a communication channel) which can be placed in a larger context $C[\cdot]$ (e.g. a protocol system). The property of the underlying system expressed by the fairness constraint should remain the same in the larger context. Therefore, within some reasonable limits, it should make no difference whether the same fairness constraint is assumed of L or of the composition $C[L]$.[1] Thus, the fairness operator should ideally have the following property of *context-independence*, which essentially means (limited) commutativity with parallel composition and hiding.

Definition 11. *Let ϕ be a formula expressed in terms of the actions A. A fairness operator Φ_ϕ for ϕ is* context-independent *with respect to equivalence "$\simeq$", iff "$\simeq$" is a congruence with respect to Φ_ϕ, and for any LTSs L and L'*

- $(\Phi_\phi L) \,||\, L' \simeq \Phi_\phi(L \,||\, L')$ *[Note that "$||$" is a commutative operator.]*
- *if $B \cap A = \emptyset$, then* $\mathbf{hide}\ B\ \mathbf{in}\ \Phi_\phi L \ \simeq\ \Phi_\phi \mathbf{hide}\ B\ \mathbf{in}\ L$

However, as readers familiar with process algebra may already expect, the property of context-independence cannot be fulfilled by an equivalence that preserves liveness properties, even if it is as detailed as strong bisimulation. This is essentially because other processes can interfere with the actions we use to declare fairness.

As an example, if we want that process A in Figure 2 always gets a chance to eventually execute a in the combination $A||B$, we can declare either weak or strong fairness towards a. The resulting process has no executions ending in an infinite sequence of b's, so there are none even when this process is combined with C. Consequently, there are no divergences after we hide b from the result: $\mathit{Divtr}(\mathbf{hide}\ b\ \mathbf{in}\ (\Phi_{\phi(a)}(A \,||\, B) \,||\, C)) = \emptyset$. However, if we combine $A \,||\, B$ with C before adding the fairness constraint, the result is able to execute b^ω because a is not enabled in the combination $A \,||\, B \,||\, C$. This turns into a divergence when we hide b: $\varepsilon \in \mathit{Divtr}(\mathbf{hide}\ b\ \mathbf{in}\ \Phi_{\phi(a)}(A \,||\, B \,||\, C))$. This shows that if an equivalence distinguishes whether there are infinite τ-executions in a process or not, the fairness operator $\Phi_{\phi(a)}$ cannot be context-independent with respect to it.

Proposition 2. *Let "$\simeq$" be an equivalence such that $L \simeq L'$ implies $\mathit{Divtr}(L) = \emptyset \Leftrightarrow \mathit{Divtr}(L') = \emptyset$, and let $\phi(a) \equiv \Diamond\Box En(a) \Rightarrow \Box\Diamond a$ or $\Box\Diamond En(a) \Rightarrow \Box\Diamond a$. Then, no fairness operator $\Phi_{\phi(a)}$ is context-independent with respect to "$\simeq$".*

[1] Practical application of this principle is demonstrated in Section 5.

The same example (easily translated into CCS) can be used to illustrate a point mentioned in Section 1. Namely, if we used "infinitary restriction" as in [20] and, say, restricted $A \,||\, B$ to the set $(b^*a)^\omega$, this would not cause any problems. However, the result of combining this process with C would be equivalent to $A \,||\, B \,||\, C$ restricted to the empty set of sequences, which is obviously an "impossible" situation.

In the next section we will, however, propose a partial solution that does not suffer from any of the above-mentioned problems. Furthermore, it is compatible with the CFFD-semantics, and therefore allows us to use existing CFFD-based verification tools. We achieve this by restricting our scope in two ways. Firstly, we consider only a specific set of fairness constraints. Secondly, we restrict the set of processes for which we apply the constraints.

4 LTSs as Fairness Operators

Similarly as in [20], we will no longer consider fairness assumptions based on the enabledness of actions. Instead, we will consider fairness assumptions of the following form, which seem to occur frequently in proofs involving fairness:

$$\Box\Diamond a_1 \vee \cdots \vee \Box\Diamond a_m \Rightarrow \Box\Diamond b_1 \vee \cdots \vee \Box\Diamond b_n$$

Here, $m > 0$, $n \geq 0$, and the interpretation is "if one of $a_1, \ldots, a_m$ occurs infinitely many times then one of $b_1, \ldots, b_n$ also has to occur infinitely many times". In case $n = 0$ the formula reduces to $\neg(\Box\Diamond a_1 \vee \cdots \vee \Box\Diamond a_m)$, with the interpretation "$a_1, \ldots, a_m$ cannot occur infinitely many times". From now on we will denote this formula by $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$.

Below we define an LTS which has all possible traces but only those infinite traces that model the fairness property. Furthermore, it has no divergences and the only actions it is allowed to refuse are the a_i-actions after executing one of these actions. We then define the fairness operator by using such an LTS and parallel composition. The idea of a "fairness LTS" was used in an application-specific way in [22] to ensure that neither side of a bidirectional communication protocol is starved. There, the result was obtained by using a finite upper and lower approximation of the LTS representing the property, together with some manual reasoning. Here, we intend to provide a general theory that allows using the above class of fairness properties in all applicable systems.

Definition 12. *$L = (S, \Sigma, \Delta, \hat{s})$ is a* fairness LTS *for $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$ if and only if $\Sigma = \{a_1, \ldots, a_m, b_1, \ldots, b_n\}$, and*

- $Tr(L) = \Sigma^*$
- $Divtr(L) = \emptyset$
- $Sfail(L) \subseteq \Sigma^* \times \{\emptyset\} \;\cup\; \Sigma^*\{a_1, \ldots, a_m\} \times 2^{\{a_1, \ldots, a_m\}}$
- $Inftr(L) = \Sigma^\omega - \Sigma^*\{a_1, \ldots, a_m\}^\omega$

It is easy to see that the LTS L_ψ defined below has the desired properties. It is also easy to see that the operator definition given later would produce equivalent results in terms of CFFD if we used any other LTS with these properties.

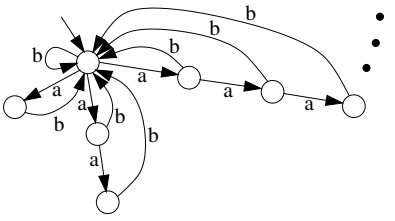


Fig. 3. A fairness LTS for actions a and b

Definition 13. *For a formula* $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$*, let* L_ψ *be the LTS* $(S, \Sigma, \Delta, \hat{s})$*, where*

- $S = \{s_0^0\} \cup \bigcup_{i=1}^{\infty} S^i$*, where* $S^i = \{s_1^i, s_2^i, \ldots, s_i^i\}$ *and* $s_j^{\,i} \neq s_{j'}^{\,i'}$ *unless* $i = i'$ *and* $j = j'$
- $\Sigma = \{a_1, \ldots, a_m, b_1, \ldots, b_n\}$
- $\Delta = \left\{ (s, b_k, s_0^0) \;\middle|\; s \in S \wedge 1 \leq k \leq n \right\} \cup \bigcup_{i=1}^{\infty} \Delta^i$*, where* $\Delta^i = \left\{ (s_0^0, a_k, s_1^i) \;\middle|\; 1 \leq k \leq m \right\} \cup \left\{ (s_j^i, a_k, s_{j+1}^i) \;\middle|\; 1 \leq j < i \wedge 1 \leq k \leq m \right\}$
- $\hat{s} = s_0^0$

Proposition 3. L_ψ *is a fairness LTS for* $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$*.*

Figure 3 illustrates L_ψ corresponding to fairness constraint $\psi(a; b)$. It has a-branches of length 1, 2, 3, ... from the initial state, and from every state (including the initial state) it is possible to return to the initial state with b. In case $m > 1$ and/or $n > 1$, the a and b-actions are repeated for every a_i, b_j, respectively. In case $n = 0$, we get the fairness LTS by removing all the b-actions.

The fairness LTS has to be infinitely branching, because there does not exist a finitely branching divergence-free LTS with all finite traces but with only the allowed infinite traces. However, this does not cause any problems for us on the denotational level, because CFFD-equivalence is a congruence even among infinitely branching LTSs. We would also like to emphasise that the fairness LTS is a theoretical tool by which we can add a fairness assumption to a system in a consistent way without having to create a complicated special theory for this purpose. When applying fairness to verification we will not actually construct the infinite fairness LTS, as will be explained in the next section.

As indicated above, before we define the actual fairness operator we have to restrict the set of LTSs to which it can be applied. Intuitively, we could require that in these LTSs the actions a_i always start at unstable states. Such a state has already the nondeterminstic choice of taking a τ-transition instead of the a_i-transition, and stable failures are not affected by refusing a_i. However, it turns out that a related but weaker requirement suffices:

Definition 14. *LTS* $L = (S, \Sigma, \Delta, \hat{s})$ *is* compatible *with* $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$ *iff* $\{a_1, \ldots, a_m, b_1, \ldots, b_n\} \subseteq \Sigma$ *and* $\forall (\sigma, A) \in Sfail(L) : (\sigma, A \cup \{a_1, \ldots, a_m\}) \in Sfail(L)$. *The set of LTSs compatible with* ψ *is denoted* $COMP_\psi$.

Note that the given condition can be determined from the CFFD-model of an LTS, so its validity is preserved when a system is replaced by a CFFD-equivalent one.

The fairness operator Ψ_ψ can now be defined simply as follows.

Definition 15. *For a formula* $\psi(a_1, \ldots, a_m, b_1, \ldots, b_n)$, $\Psi_\psi(L) = L \,||\, L_\psi$.

The following result shows that Ψ_ψ really is a fairness operator in the sense of Definition 10. We omit the proof which is based on Proposition 3, Definition 12, Definition 14 and the properties of "||".

Proposition 4. *When used on LTSs from* $COMP_\psi$, Ψ_ψ *is a fairness operator for* $\psi(a_1, \ldots, a_m; b_1, \ldots, b_n)$.

As our approach is "one-level", there obviously cannot be a similar conflict between the operational and fairness levels that is possible with "two-level" approaches, as discussed in Section 1. In our approach, if a process were unable to execute within the allowed infinite execution sequences, this would present itself as new deadlocks. Therefore, the above result is especially important, because it shows that this will never happen. Furthermore, unlike in some other approaches, the fact that subprocesses can stop executing (e.g. when blocked by others) does not cause any problems in our approach.

The congruence property follows directly from the congruence property of "$\simeq_{\mathsf{CFFD}}$" with respect to "||", and context-independence from the commutativity and associativity of "||" and from a commutativity property of "||" and "**hide**".

Proposition 5. *"*$\simeq_{\mathsf{CFFD}}$*" is a congruence with respect to* Ψ_ψ.

Proposition 6. Ψ_ψ *is context-independent with respect to "*$\simeq_{\mathsf{CFFD}}$*".*

It is also easy to see that compatibility is preserved by our fairness operators and that our fairness operators commute among themselves.

5 Verification Example

Although the above approach is consistent for the given class of fairness properties, it involves infinite LTSs. This does not mean, however, that the solution is only a theoretical one. Often the infinite LTS produced by the operator is only an intermediate subsystem, and the complete system, where the subsystem is composed with other processes and actions are hidden, is finite. We can take advantage of this by using the context-independence property.

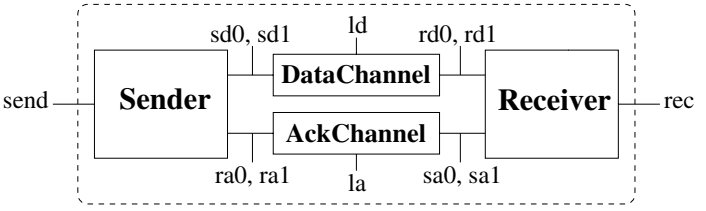


Fig. 4. The alternating bit protocol

When we have constructed a finite system P and noted that it contains divergences, we may find it reasonable to make certain fairness assumptions about some components of the system. Including the assumptions as fairness operators creates a modified system P'. By the context-independence property we can move the fairness operators and the hiding of the actions involved from the subcomponents to the topmost level of compositional system construction. There we can check the effect of the fairness operators/LTSs on the system without actually having to construct the fairness LTSs. The theory automatically guarantees that the finite behaviour (traces and stable failures) of P' is the same as in P. The first thing we check is that the infinite traces of the system are unaffected (it can be shown that the new system P' will be finite precisely when this is the case). We then check that all divergences are caused by infinite executions that are removed by the fairness operators. If this is true, we have shown that P' is the same as P but without the divergences.

We illustrate this approach by applying it to the well-known alternating bit protocol [3]. This protocol is intended for sending messages over channels that can lose messages, but cannot reorder them. There are two one-way channels, one for the data from the sender to the receiver and another for acknowledgements from the receiver to the sender, as depicted in Figure 4. Acknowledgements are needed because messages can be lost. If the acknowledgement for a message is not received in time, the protocol attempts retransmission. In order not to confuse new messages with retransmissions, each message and acknowledgement contains a sequence number, which is either 0 or 1.

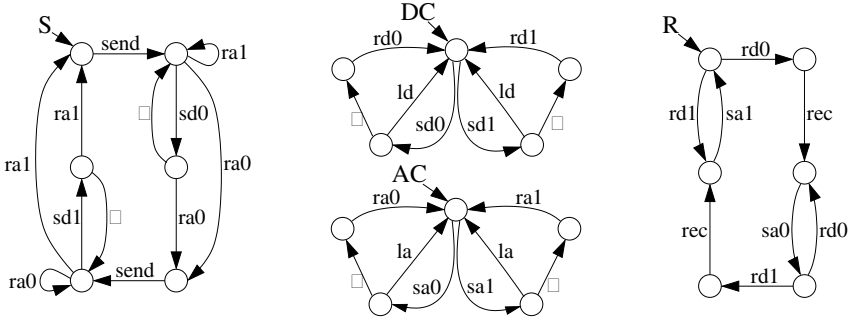


Fig. 5. The LTSs of the components of the alternating bit protocol

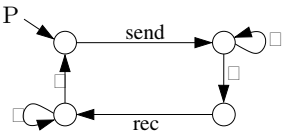


Fig. 6. The reduced global behaviour of the alternating bit protocol

Our LTS definitions of the sender (S), data channel (DC), acknowledgement channel (AC) and receiver (R) are shown in Figure 5. For simplicity we do not model the data content of messages, as it does not directly affect the behaviour of the protocol. The sender gets a sending request from the user, and then sends a data message to the data channel with the appropriate bit value. If a correct acknowledgement is not received in time, the sender makes a timeout by executing the invisible τ-transition and then sends again. The sender contains some extra transitions for consuming unexpected acknowledgements. The data channel gets a message from the sender and then chooses either to lose it (ld) or pass it through (τ) and give it to the receiver. The acknowledgement channel works similarly. When the receiver gets a data message with a new bit value, it declares it with rec and sends an acknowledgement. For repeated messages it only sends an acknowledgement.

We hide the internal actions $I = \{sd0, sd1, ld, rd0, rd1, sa0, sa1, la, ra0, ra1\}$ from the total system and leave only the external actions $\{send, rec\}$ visible. Then the complete system $P = \mathbf{hide}\ I\ \mathbf{in}\ (S \,||\, DC \,||\, AC \,||\, R)$ is reduced with a CFFD-preserving reduction algorithm [28]. The result is shown in Figure 6. We note that the behaviour is otherwise acceptable, but there are two τ-loops, i.e. divergences, in the system. We cannot know with certainty that after entering one of these loops the system ever executes any more visible actions.

The channels can lose all messages that are given to them. We can therefore guess that the divergences are caused by an infinite sequence of retransmissions and losses of messages in the channels. If we assume that the channels cannot lose an infinite number of messages, the divergences should then disappear. We will now formally verify that this is the case by using fairness operators. Let $P' = \mathbf{hide}\ I\ \mathbf{in}\ (S \,||\, (\Psi_{\psi(ld)} DC) \,||\, (\Psi_{\psi(la)} AC) \,||\, R)$, where $\psi(ld) \equiv \neg\Box\Diamond ld$ and $\psi(la) \equiv \neg\Box\Diamond la$. We are allowed to use these operators because the states where ld and la start are unstable, and thus DC and AC are in $COMP_{\psi(ld)}$ and $COMP_{\psi(la)}$, respectively. By the context-independence property this is equivalent to:

$$\mathbf{hide}\ \{ld, la\}\ \mathbf{in}\ \Psi_{\psi(ld)}\Psi_{\psi(la)}(\ \mathbf{hide}\ I'\ \mathbf{in}\ (\ S \,||\, DC \,||\, AC \,||\, R\ )),$$

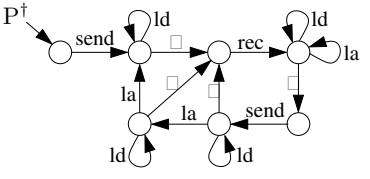


Fig. 7. Like Figure 6, but now also ld and la are visible

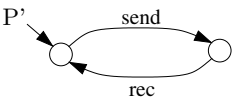


Fig. 8. The behaviour with the fairness assumption

where $I' = I - \{ld, la\}$ (this shows the usefulness of context-independence). The inside process $P^\dagger = \mathbf{hide}\ I'\ \mathbf{in}\ (\, S \,||\, DC \,||\, AC \,||\, R\,)$ after CFFD-reduction is shown in Figure 7.

We can now determine the CFFD-model of P'. By Proposition 4, $\mathit{Sfail}(P') = \mathit{Sfail}(P)$. As for *Inftr*, the only infinite trace of P (Figure 6) is $(\mathit{send}\ \mathit{rec})^\omega$, and from Figure 7 we see that this infinite trace is possible without any *ld*- or *la*-actions. Therefore, it does not violate $\psi(ld)$ or $\psi(la)$, so $\mathit{Inftr}(P') = \mathit{Inftr}(P)$. As for *Divtr*, there are no divergences in Figure 7, so any divergences would have to emerge in hiding from infinite traces ending in $\{ld, la\}^\omega$. However, the fairness operators remove all such infinite traces, so $\mathit{Divtr}(P') = \emptyset$. We can conclude that the behaviour of P' is as shown in Figure 8, and this is clearly acceptable.

However, we can obtain an even better result. We made the assumption that the channels can only lose a finite number of messages, during their entire operation. We will next try the weaker assumption that the channels can only lose a finite number of messages *between* passing messages through, i.e., that the channels cannot from some point on lose all messages. Therefore, let $P'' = \mathbf{hide}\ A\ \mathbf{in}(\, S \,||\, \Psi_{\psi(ld;rd0,rd1)} DC \,||\, \Psi_{\psi(la;ra0,ra1)} AC \,||\, R\,)$. As before, $\psi(ld; rd0, rd1)$ means $\Box\Diamond ld \Rightarrow \Box\Diamond rd0 \vee \Box\Diamond rd1$, and so on. By following the same arguments as above, $\mathit{Sfail}(P'') = \mathit{Sfail}(P)$ and $\mathit{Inftr}(P'') = \mathit{Inftr}(P)$.

Because even infinite *ld*- and *la*-executions are now possible, for *Divtr* we need a larger diagram where also *rd*- and *ra*-actions are visible. We do not show the diagram (with 34 states) here, but it is straightforward to check from it that all infinite traces that could turn into divergences violate either $\psi(ld; rd0, rd1)$ or $\psi(la; ra0, ra1)$, so $\mathit{Divtr}(P'') = \emptyset$. Thus, even with the weaker assumption, the behaviour of the protocol is the LTS in Figure 8.

We would like to emphasise that although the part of the proof done by investigating Figure 7 and the corresponding 34-state LTS was itself a model checking task, this does not mean that our approach reduces to classical model checking. Firstly, in classical model checking, the complete state space would have to be used instead of Figure 7. In the example the complete state space contains 268 states. Figure 7 can be constructed with any CFFD-preserving reduced LTS construction method that avoids the construction of the full state space, of which there are many. Secondly, the final result of our analysis was not the answer "yes, the formula holds", but the LTS in Figure 8. This LTS not only allows us to verify all properties preserved by the equivalence, but it can also be placed as a component in further compositional analysis.

6 Conclusions

In this article we have been studying the use of fairness and liveness in process algebra. We proposed intuitively reasonable requirements for a hypothetical operator that would implement fairness constraints. However, we showed that there are incompatibilities between these requirements and most well-known semantic models. One problem is that the models do not sufficiently preserve information on the enabledness of actions in the infinite executions of the system. We also demonstrated that the compositionality of process algebra imposes limitations on the use of fairness. This is because outside processes can interfere with the actions which we use to define a fairness constraint.

However, we presented an approach where a particular class of fairness constraints can be used within the ordinary LTS model and our existing compositional semantics. We showed that the approach is compatible with all the stated requirements, and we used it to remove livelocks from an example protocol. To avoid the above inconsistency with compositionality there are two possible approaches: we can restrict the set of target processes for the fairness operator, and/or we can restrict the set of contexts within which the property of context-independence is guaranteed. The approach we took here was the former.

Obvious topics for further research are enhancing the automated support for the approach and extending the results to a larger set of fairness properties. It is also important to note that our approach does not rule out finite representations such as Büchi automata or ω-regular languages, used in [7,20]. These could be used as finite representations for certain well-defined infinite ordinary LTSs, so that the behavioural properties of these finite representations would be proved, rather than defined, by us.

Acknowledgements. The work of A. Puhakka was funded by the Academy of Finland, project "Unifying Action-Based and State-Based Verification Techniques", and the TISE Graduate School.

References

1. Alpern, B. & Schneider, F. B.: "Defining Liveness". Information Processing Letters, Vol. 21 (No. 4), 1985, North-Holland.
2. Apt, K., Francez, N. & Katz, S.: "Appraising fairness in languages for distributed programming". Distributed Computing (1988) Vol. 2 (No. 4), Springer-Verlag, pp. 226–241.
3. Bartlett, K. A., Scantlebury, R. A. & Wilkinson, P. T.: "A Note on Reliable Full-Duplex Transmission Over Half-Duplex Links", *Communications of the ACM* 12 (5) 1969, pp. 260–261.
4. Bergstra, J. A., Klop, J. W.: "Verification of an Alternating Bit Protocol by Means of Process Algebra". *Mathematical Methods of Specification and Synthesis of Software Systems '85*, Lecture Notes in Computer Science 215, Springer-Verlag 1985, pp. 9–23.

5. Brinksma, E., Rensink, A. & Vogler, W.: "Fair Testing". *CONCUR'95, Sixth International Conference on Concurrency Theory*, Lecture Notes in Computer Science 962, Springer-Verlag 1995, pp. 313–327.
6. Clarke, E. M. & Emerson, E. A.: "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic". *Proc. Logic in Programs: Workshop, 1981*, Lecture Notes in Computer Science 131, Springer-Verlag 1981, pp. 52–71.
7. Cleaveland, R. & Lüttgen, G.: "A Semantic Theory for Heterogeneous System Design". *Proc. Foundations of Software Technology and Theoretical Computer Science 2000*. Lecture Notes in Computer Science 1974, Springer-Verlag 2000, pp. 312–324.
8. Costa, G. & Stirling, C.: "A Fair Calculus of Communicating Systems". Acta Informatica 21 (1984), Springer-Verlag, pp. 417–441.
9. Costa, G. & Stirling, C.: "Weak and Strong Fairness in CCS". *Proc. Mathematical Foundations of Computer Science 1984*, Lecture Notes in Computer Science 176, Springer-Verlag 1984, pp. 245–254.
10. Davies, J. & Schneider, S.: "Using CSP to Verify a Timed Protocol over a Fair Medium". *Proc. CONCUR'92, Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 355–369.
11. Francez, N.: "Fairness". Springer-Verlag 1986, 295 p.
12. Hennessy, M.: "Axiomatising Finite Delay Operators". Acta Informatica 21 (1) 1984, Springer-Verlag, pp. 61–88.
13. Hennessy, M.: "An Algebraic Theory of Fair Asynchronous Communicating processes". Theoretical Computer Science 49 (1987), North-Holland, pp. 121–143.
14. Kaivola, R. & Valmari, A.: "The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic". *Proc. CONCUR '92, Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.
15. Lamport, L.: "Proving the Correctness of Multiprocess Programs". IEEE Transactions on Software Engineering, Vol. SE-3 (No. 2) 1977, IEEE Computer Society, pp. 125–143.
16. Lamport, L.: "The Temporal Logic of Actions". ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16 (No. 3) 1994, pp. 872–923.
17. Lehmann, D., Pnueli, A. & Stavi, J.: "Impartiality, Justice and Fairness: The Ethics of Concurrent Termination". *Proc. Eighth International Colloquium on Automata, Languages and Programming 1981*, Lecture Notes in Computer Science 115, Springer-Verlag 1981, pp. 264–277.
18. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification*. Springer-Verlag 1992, 427 p.
19. Milner, R.: *Communication and Concurrency*. Prentice-Hall 1989, 260 p.
20. Parrow, J.: *Fairness Properties in Process Algebra with Applications in Communication Protocol Verification*. Ph.D. thesis, Uppsala University, Department of Computer Systems, 1985, 176 p.
21. Pnueli, A.: "A Temporal Logic of Concurrent Programs". Theoretical Computer Science, 13, 1981, pp. 45–60.
22. Puhakka, A. & Valmari, A.: "Livelocks, Fairness and Protocol Verification". *Proc. World Computer Conference 2000, Conference on Software: Theory and Practice*, International Federation for Information Processing (IFIP), pp. 471–479.
23. Puhakka, A. & Valmari, A.: "Liveness and Fairness in Process-Algebraic Verification". To be published as Tampere University of Technology, Software Systems Laboratory Report 24, ISBN 952-15-0650-4.

24. Queille, J. P. & Sifakis, J.: "Specification and Verification of Concurrent Systems in CESAR". *Proc. Fifth International Symposium on Programming*, 1981.
25. Roscoe, A. W.: *The Theory and Practice of Concurrency*. Prentice-Hall 1998, 565 p.
26. Valmari, A.: "Failure-based Equivalences Are Faster Than Many Believe". *Proc. Structures in Concurrency Theory 1995*, Springer-Verlag "Workshops in Computing" series, 1995, pp. 326–340.
27. Valmari, A.: "Compositionality in State Space Verification Methods". *Proc. Application and Theory of Petri Nets 1996*, Lecture Notes in Computer Science 1091, Springer-Verlag 1996, pp. 29–56.
28. Valmari, A., Kemppainen, J., Clegg, M. & Levanto, M.: "Putting Advanced Reachability Analysis Techniques Together: the 'ARA' Tool". *Proc. Formal Methods Europe '93*, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 597–616.
29. Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.

Bounded Reachability Checking with Process Semantics*

Keijo Heljanko

Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
Keijo.Heljanko@hut.fi

Abstract. Bounded model checking has been recently introduced as an efficient verification method for reactive systems. In this work we apply bounded model checking to asynchronous systems. More specifically, we translate the bounded reachability problem for 1-safe Petri nets into constrained Boolean circuit satisfiability. We consider three semantics: process, step, and interleaving semantics. We show that process semantics has often the best performance for bounded reachability checking.

1 Introduction

Bounded model checking [3] has been proposed as a verification method for reactive systems. The main idea is to look for counterexamples which are shorter than some fixed length for a given property. If a counterexample can be found, then the property does not hold for the system. If no counterexample can be found using this bound, usually the result is inconclusive. The decision procedure most often used in bounded model checking is propositional satisfiability. Given the transition relation of the reactive system to be model checked, the property, and the bound n, the transition relation and property are "unrolled" n times to obtain a propositional formula which is satisfiable iff there is a counterexample with bound n. The implementation ideas are quite similar to procedures used in AI planning [11,15].

In this work we apply bounded model checking to asynchronous systems. More specifically, we translate the bounded reachability problem for 1-safe Petri nets into constrained Boolean circuit satisfiability. This work can be seen as a continuation of the work done in [9]. There we discuss using the step and interleaving semantics for bounded reachability, while the formalism into which the problem is translated being logic programs with stable model semantics. The main contribution of this paper is that we show that the so called process semantics of Petri nets [1,2] can be used to improve the efficiency of bounded model checking. Namely, also the process semantics can be efficiently encoded into constrained Boolean circuits.

* The financial support of the Academy of Finland (Projects 43963 and 47754), and Tekniikan Edistämissäätiö foundation are gratefully acknowledged.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 218–232, 2001.
© Springer-Verlag Berlin Heidelberg 2001

As an additional contribution we report on an implementation called punroll, which uses the BCSat constrained Boolean circuit satisfiability checker to check whether the generated constrained circuit is satisfiable, thus solving the bounded reachability problem.

The structure of the rest of the paper is the following. First we introduce Petri nets and the three different semantics in Sect. 2. Then we shortly introduce constrained Boolean circuits in Sect. 3, and in Sect. 4 show how the bounded reachability problem can be translated into them. After that we discuss our implementation and experiments in Sect. 5, and finish with conclusions in Sect. 6.

2 Petri Nets

We will now introduce Petri nets. A *net* is a triple (P, T, F), where P and T are disjoint sets of *places* and *transitions*, respectively, and F is a function $(P \times T) \cup (T \times P) \to \{0, 1\}$. Places and transitions are generically called *nodes*. If $F(x, y) = 1$ then we say that there is an *arc* from x to y. The *preset* of a node x, denoted by ${}^\bullet x$, is the set $\{y \in P \cup T \mid F(y, x) = 1\}$. The *postset* of x, denoted by $x^\bullet$, is the set $\{y \in P \cup T \mid F(x, y) = 1\}$. In this paper we consider only finite nets in which every transition has a nonempty preset *and* a nonempty postset. A *marking* of a net (P, T, F) is a mapping $P \to \mathbb{N}$ (where $\mathbb{N}$ denotes the natural numbers including 0). We identify a marking M with the multiset containing $M(p)$ copies of p for every $p \in P$. For instance, if $P = \{p_1, p_2\}$ and $M(p_1) = 1$, $M(p_2) = 2$, we write $M = \{p_1, p_2, p_2\}$. A 4-tuple $\Sigma = (P, T, F, M_0)$ is a *net system* if (P, T, F) is a net and M_0 is a marking of (P, T, F) (called the *initial marking* of Σ). We will use as a running example the net system in Fig. 1.

2.1 Step Semantics

To save some space, we define the behavior of a net system through step semantics. The (usual) interleaving semantics will then be defined later based on this more general concept.

A step is a non-empty set of transitions $S \subseteq T$. [1] We denote a step by $[S\rangle$. A marking M *enables* a step S if for all $p \in P$ it holds that $M(p) \geq \sum_{t \in S} F(p, t)$. If the step S is enabled at M, then it can *fire* or *occur*, and its occurrence *leads to* a new marking M' defined as $M'(p) = M(p) + \sum_{t \in S}(F(t, p) - F(p, t))$ for every place $p \in P$. We denote this firing of a step by $M[S\rangle M'$.

A (possibly empty) sequence of steps $\sigma = [S_0\rangle[S_1\rangle \cdots [S_{n-1}\rangle$ is a *step execution* of the net system $\Sigma = (P, T, F, M_0)$ if there exist markings M_1, M_2, ... , M_n such that $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{n-1}[S_{n-1}\rangle M_n$. The marking reached by the occurrence of σ is M_n. A marking M is a *reachable marking* if there exists a step execution σ such M is reached by the occurrence of σ. A marking M is *reachable with bound n* if there exists a step execution σ consisting of (exactly) n steps

[1] We only consider a class of nets where the transitions cannot be self-concurrent. Therefore a set suffices and multisets, i.e., bags are not needed.

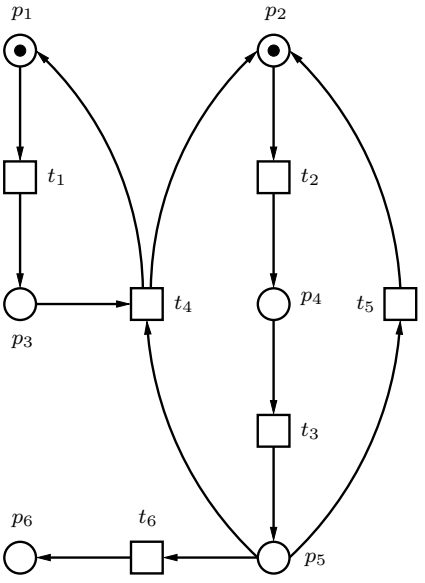


Fig. 1. Running Example

such M is reached by the occurrence σ. Correspondingly we say that a marking M is *reachable within bound* n if there exists an integer $0 \leq i \leq n$ such that M is reachable with bound i.

In our running example the step $[t_1, t_2\rangle$ is enabled in the initial marking and thus $\{p_1, p_2\}[t_1, t_2\rangle\{p_3, p_4\}$. The marking $\{p_3, p_6\}$ is reachable with bound 3, as $\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$ is a step execution.

A marking M of a net is *n-safe* if $M(p) \leq n$ for every place p. A net system Σ is n-safe if all its reachable markings are n-safe. In this work we restrict ourselves to net systems which are 1-safe. They are quite an interesting class, as e.g., net systems arising from synchronization of state machines are 1-safe. Note that for 1-safe net systems all reachable markings are reachable within bound $n = (2^{|P|} - 1)$. Thus the set "marking reachable within bound n" can be seen as a lower approximation of the set of reachable markings which improves as the bound n increases. See discussion in [3] on how to check whether a bound is sufficient for completeness. Quite often a much smaller bound than the one discussed above suffices for completeness. For a general discussion of the computational complexity of verification problems for 1-safe Petri nets, see e.g., [6].

2.2 Interleaving Semantics

An *interleaving execution* is a step execution $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{n-1}[S_{n-1}\rangle M_n$ such that for all $0 \leq i \leq n-1$ it holds that $|S_i| = 1$. A marking is *reachable in the interleaving semantics* if there exists an interleaving execution σ such that

M is reached by the occurrence of σ. The bounded versions of reachability are defined similarly to the step case.

Again in our example the marking $\{p_3, p_6\}$ is reachable in the interleaving semantics with a bound 4, as $\{p_1, p_2\}[t_1\rangle\{p_2, p_3\}[t_2\rangle\{p_3, p_4\}[t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$ is an interleaving execution. Notice however, that the marking $\{p_3, p_6\}$ is *not* reachable in the interleaving semantics with bound 3.

It is well known, see e.g., [1] that for the net class used here the set of reachable markings in the step and interleaving semantics coincide. However, in bounded model checking using step semantics might be useful, as in many cases markings can be reached with a smaller bound than in the interleaving semantics.

2.3 Process Semantics

However, there is a problem with steps. Namely, there can be several step executions which intuitively represent the same "concurrent behavior". These can in bounded model checking introduce search space which can adversely effect the running time of the solver used. To avoid this we will use a well known semantics from the literature called the process semantics, see [1,2].

We will now recall from the literature a construction which constructs a process from a finite step execution. The following is a modified version (simpler because of 1-safeness) of the Construction 4.9 in [1].

For this definition we need some additional notation. For a net $N = (P, T, F)$ the function $Max(N) = \{x \in P \mid x^\bullet = \emptyset\}$. Let L be a finite set. A *labelled net* is a 4-tuple (P, T, F, l), where (P, T, F) is a net and $l\colon P \cup T \to L$ is a labelling.

Definition 1. *(Derivation of process from step execution.) Let $\Sigma = (P, T, F, M_0)$ be a net system and let $\sigma = [S_0\rangle[S_1\rangle \cdots [S_{n-1}\rangle$ be a sequence of steps such that $M_0[S_0\rangle M_1[S_1\rangle \cdots [S_{n-1}\rangle M_n$ is a step execution of Σ. We associate with σ a labelled net $\Pi(\sigma)$ by creating a sequence of labelled nets $N_i = (B_i, E_i, G_i, l_i)$ with labelling $l_i : B_i \cup E_i \to P \cup T$ by induction on i, where $0 \leq i \leq n$.*

($i = 0$): $E_0 = \emptyset$, $G_0 = \emptyset$, and B_0 contains for each $p \in P$ such that $M_0(p) = 1$ a place b with $l_0(b) = p$.
($i = i + 1$): Suppose that N_i has been constructed.
First we require that everything in N_i is also in N_{i+1}. For all $x, y \in B_i \cup E_i$: $x \in B_i \Rightarrow x \in B_{i+1}$, $x \in E_i \Rightarrow x \in E_{i+1}$, $(x, y) \in G_i \Rightarrow (x, y) \in G_{i+1}$ and $l_{i+1}(x) = l_i(x)$.
Then for each $t \in S_i$ do the following:

- *for each $p \in {}^\bullet t$ find the place $b(p) \in Max(N_i)$ such that $l_i(b(p)) = p$,*
- *add a new transition e to E_{i+1} with $l_{i+1}(e) = t$ and add $(b(p), e)$ to G_{i+1} for all $p \in {}^\bullet t$,*
- *for each $p \in t^\bullet$ add a new place $b'(p)$ to B_{i+1} with $l_{i+i}(b'(p)) = p$ and $(e, b'(p)) \in G_{i+1}$.*

Finally take $\Pi(\sigma) = N_n = (B_n, E_n, G_n, l_n)$.

The construction above is fully deterministic (as this version is for 1-safe nets only) and thus the result is unique up to isomorphism. This fact is well known, see e.g., the discussion of a similar definition, Def. 3 in [12]. For simplicity, from now on we will identify all isomorphic processes as being equivalent.

Consider now our running example in Figure 1. It has a step execution $\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$. Now given $\sigma = [t_2\rangle[t_1, t_3\rangle[t_6\rangle$ we can construct the process $\Pi(\sigma)$ given in Figure 2, where the labelling l of nodes is given in parenthesis.

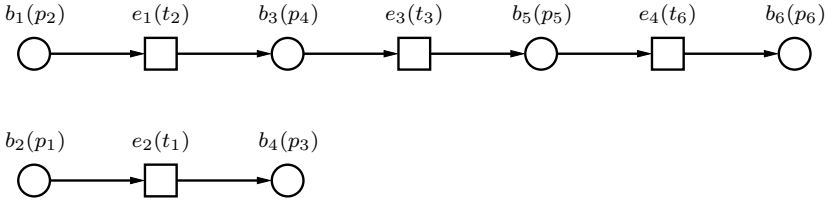


Fig. 2. A process $\pi = (B, E, G, l)$

It is easy to see that for example also the sequences of steps $\sigma' = [t_1, t_2\rangle[t_3\rangle[t_6\rangle$, $\sigma'' = [t_2\rangle[t_3\rangle[t_1, t_6\rangle$, and $\sigma''' = [t_1\rangle[t_2\rangle[t_3\rangle[t_6\rangle$ will yield the same process, i.e., $\Pi(\sigma') = \Pi(\sigma'') = \Pi(\sigma''') = \Pi(\sigma)$. All of these step executions "solve the arising conflicts" in the same way and lead to the same final marking of the process π, i.e., $l(Max(\pi)) = \{p_3, p_6\}$. Thus if we are only interested in the final marking it should intuitively be sufficient to only generate one of them. We will now show how this can be done in bounded reachability checking.

We present an algorithm which given a process π gives a sequence of steps $FNF(\pi)$ (for *Foata normal form* of π) which together with Σ fully characterizes the process π. The Algorithm 1 computes the Foata normal form of a process. It is the algorithm presented on page 47 of [16] (with small notational changes). We define some notation for the algorithm. Given a set of transitions $C \subseteq E$ of the process $\pi = (B, E, G, l)$, let G^* be the transitive closure of the flow relation G, and define $MinE(C) = \{e \in C \mid \text{for all } e' \in (C \setminus \{e\}) \text{ it holds that } (e', e) \notin G^*\}$.

Assume that we are given a Foata normal form $FNF(\pi) = [S_0\rangle[S_1\rangle \cdots [S_{n-1}\rangle$ for a process π of a 1-safe net system Σ. It is easy to prove that there are markings $M_1, M_2, \ldots, M_n$ such that in the initial state M_0 of Σ the step execution $M_0[S_0\rangle M_1[S_1\rangle M_2 \cdots M_{n-1}[S_{n-1}\rangle M_n$ can occur.

This normal form is actually the Foata normal form from the theory of Mazurkiewicz traces, see e.g., [5]. It is only (quite trivially) adapted to processes of 1-safe net systems. To our knowledge it was first applied to processes of 1-safe net systems in the verification algorithm setting in [7]. (The fact that the technique used is a Foata normal form is discussed in more detail in an extended version [8], as well as in [16].)

Algorithm 1 *The Foata normal form of a process*

input: A process $\pi = (B, E, G, l)$ of a 1-safe net.
output: Foata normal form of π: A sequence of steps $FNF = [S_0\rangle[S_1\rangle \cdots [S_{n-1}\rangle$.

```
1  begin
2    C := E;
3    FNF := ε;
4    while C ≠ ∅ do
5      S := l(MinE(C));
6      FNF := FNF · [S⟩;
7      C := C \ MinE(C);
8    endwhile
9    return FNF;
10 end
```

When run on the process π of Figure 2, we will get the result $FNF(\pi) = [t_1, t_2\rangle[t_3\rangle[t_6\rangle$. This intuitively corresponds to a step execution which is "greedy", i.e., it always fires transitions at the earliest possible time moment, while still respecting the structure of the process π. Thus the step execution in Foata normal form is always among the shortest which yield the process π.

The Algorithm 1 gives an easy way of generating a Foata normal form of a process. We will in our implementation use a different definition, which is equivalent but more suitable for the implementation techniques we use. (We have not found this version in the literature. However, it is just a simple adaptation of the version for traces, see e.g., [5].)

Definition 2. *The sequence of steps $\sigma = [S_0\rangle[S_1\rangle \cdots [S_{n-1}\rangle$ is a step execution of a 1-safe net system Σ in Foata normal form if:*

(a) $\sigma = \epsilon$ (i.e., σ is the empty step sequence), or
(b) There are markings $M_1, M_2, \ldots, M_n$ such that in the initial state M_0 of Σ the step execution $M_0[S_0\rangle M_1[S_1\rangle M_2 \cdots M_{n-1}[S_{n-1}\rangle M_n$ can occur, and:
- *For each $1 \leq i \leq n-1$ and for each $t \in S_i$ there exists a transition t' in S_{i-1} such that $t'^{\bullet} \cap {}^{\bullet}t \neq \emptyset$. (Each transition t in step i with $i \geq 1$ has some transition t' in step $i-1$ which generates some part of its preset.)*

Now there is a bijection between processes and step executions in Foata normal form. Given a step execution σ one can construct the corresponding process $\pi = \Pi(\sigma)$, and given the process π we can construct the step execution $\sigma' = FNF(\pi)$ and in fact $\sigma' = \sigma$ iff σ was in Foata normal form (according to Def. 2). Thus they both describe the same concurrent behavior. It is therefore only a matter of taste whether one talks about processes or step executions in Foata normal form. We have chosen to talk about processes and process semantics, as that is the terminology most often used in Petri net literature [1, 2]. Our actual implementation is, however, based on the definition of the Foata normal form for step executions, namely Def. 2.

We thus define the process semantics as follows. A marking M is a *reachable in the process semantics* if there exists a step execution σ in Foata normal

form, such that M is reached by the occurrence of σ. The bounded versions of reachability are again defined similarly to the step case.

To rephrase our discussion, here is the (not surprising) main result used in bounded model checking with process semantics.

Theorem 1. *Let Σ be a 1-safe net system. A marking M is reachable within bound n in Σ iff in the process semantics M is reachable within bound n in Σ.* [2]

3 Boolean Circuits

This section is largely based on the presentation of [10]. A *Boolean circuit* is an directed acyclic graph where the nodes are called *gates*. The gates with no outgoing edges are *output gates* and *input gates* are those gates which do not have incoming edges nor an associated Boolean function. Each non-input gate has a Boolean function associated with it and it "calculates" the output value from the values of its children. Boolean circuits can be expressed with *Boolean expression systems*. Given a finite set $\mathcal{V}$ of Boolean variables, a Boolean equation system $\mathcal{S}$ over $\mathcal{V}$ is a set of equations of the form $v = f(v_1, \ldots, v_k)$, where $v, v_1, \ldots, v_k \in \mathcal{V}$ and f is an arbitrary Boolean function. Boolean circuits can now be seen as Boolean equation systems with the following two properties. (i) Each variable has at most one equation. (ii) The equations are not recursive. (In the sense that the variable dependency graph [10] is acyclic.)

A *truth valuation* for $\mathcal{S}$ is a function $\tau : \mathcal{V} \to \{\text{true}, \text{false}\}$. A valuation is *consistent* if $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ for each equation in $\mathcal{S}$. The *constrained satisfiability problem* for Boolean circuits is the following: given that variables $c^+ \subseteq \mathcal{V}$ must be true and variables in $c^- \subseteq \mathcal{V}$ must be false, is there a consistent valuation that respects these constraints? We call such a truth assignment a *satisfying truth assignment*. The constrained Boolean circuit satisfiability problem is obviously an **NP**-complete problem under the plausible assumption that each Boolean function in the system can be evaluated in polynomial time.

In the rest of this paper we use Boolean circuits where the following Boolean functions are used as gates:

- $\top$ is always true.
- $\bot$ is always false.
- $\mathsf{not}(v)$ = true iff v is not true.
- $\mathsf{or}(v_1, \ldots, v_k)$ = true iff at least one of v_i, $1 \leq i \leq k$ is true.
- $\mathsf{and}(v_1, \ldots, v_k)$ = true iff all of v_i, $1 \leq i \leq k$ are true.
- $\mathsf{card}_L^U(v_1, \ldots, v_k)$ = true iff for the cardinality c of the set of variables v_i which are true it holds that $L \leq c \leq U$. (Where L and U are fixed constants $0 \leq L \leq U$.)

The function $\mathsf{card}_L^U(v_1, \ldots, v_k)$ is actually a family of functions. We use in this work only the special form $\mathsf{card}_0^1(v_1, \ldots, v_k)$, which is true if less than two of the variables in the set $\{v_1, \ldots, v_k\}$ are true. We will show that this function is quite useful for compactly encoding which transitions can not be fired concurrently.

[2] Note the use of *within* instead of *with*. A marking may be reachable with a bound n and only reachable with bound i in the process semantics, where $i < n$.

4 Translating Bounded Reachability into Boolean Circuits

We will now present how to translate the bounded reachability problem for 1-safe nets into constrained satisfiability problem for Boolean circuits. The Figures 3-5 give parts of the translation for our running example of Figure 1. We suggest the reader to consult them while reading the definition of the translation. Consider a 1-safe net system $\Sigma = (P, T, F, M_0)$ and a fixed bound n. We first construct (in (a)-(b) below) a constrained Boolean circuit which captures the possible step executions of Σ of length $\leq n$, where $n \geq 0$.

(a) To capture the initial marking, for each place $p_j \in P$ we create a gate $\mathsf{p_j}(0)$ and associate $\top$ as the function if $M_0(p_j) = 1$, and $\bot$ otherwise.

(b) For each step $0 \leq i \leq n-1$ we add the following gates:

1. For each transition $t_j \in T$ we create an input gate $\mathsf{t_j}(i)$. If this gate is true, it intuitively means that the transition t_j fires in step i.
2. For each place $p_j \in P$ we create an or gate $\mathsf{gp_j}(i+1)$ with the children $\{\mathsf{t_1}(i), \ldots, \mathsf{t_k}(i)\}$, where $\{t_1, \ldots, t_k\}$ is the preset of p_j. The gate $\mathsf{gp_j}(i+1)$ will be true if some transition in step i generates a token to the place p_j.
3. For each place $p_j \in P$ we create an or gate $\mathsf{rp_j}(i+1)$ with the children $\{\mathsf{t_1}(i), \ldots, \mathsf{t_k}(i)\}$, where $\{t_1, \ldots, t_k\}$ is the postset of p_j. The gate $\mathsf{rp_j}(i+1)$ will be true if some transition in step i removes a token from p_j.
4. For each place $p_j \in P$ we create a not gate $\mathsf{nrp_j}(i+1)$ with the child $\mathsf{rp_j}(i+1)$.
5. For each place $p_j \in P$ we create an and gate $\mathsf{fp_j}(i+1)$ with the children $\mathsf{p_j}(i)$ and $\mathsf{nrp_j}(i+1)$. The gate $\mathsf{fp_j}(i+1)$ is true when a place p_j contains a token before step i, and no transition removing tokens from it appears in step i.
6. For each place $p_j \in P$ we create an or gate $\mathsf{p_j}(i+1)$ with the children $\mathsf{gp_j}(i+1)$ and $\mathsf{fp_j}(i+1)$. The gate $\mathsf{p_j}(i+1)$ is true when after step i the place p_j contains a token. (Either a token was generated in step i or a token residing on the place p_j before step i still remains on the place p_j after the step i.)
7. For each transition $t_j \in T$ we create an and gate $\mathsf{pt_j}(i)$ with the children $\{\mathsf{p_1}(i), \ldots, \mathsf{p_k}(i)\}$, where $\{p_1, \ldots, p_k\}$ is the preset of t_j. The gate $\mathsf{pt_j}(i)$ will be true if all the preset places of transition t_j in step i contain a token.
8. For each transition $t_j \in T$ we create a not gate $\mathsf{nt_j}(i)$ with the child $\mathsf{t_j}(i)$.
9. For each transition $t_j \in T$ we create an or gate $\mathsf{tt_j}(i)$ and constrain it to be true. It has two children $\mathsf{nt_j}(i)$ and $\mathsf{pt_j}(i)$. The constrained gate $\mathsf{tt_j}(i)$ ensures that either the transition t_j is not fired in step i or all of its preset tokens are available.
10. For each place $p_j \in P$ such that $|p^\bullet| \geq 2$ we create a card_0^1 gate $\mathsf{ncp_j}(i)$ and constrain it to true. It has children $\{\mathsf{t_1}(i), \ldots, \mathsf{t_k}(i)\}$, where $\{t_1, \ldots, t_k\}$ is the postset of p_j. The constrained gate $\mathsf{ncp_j}(i)$ ensures

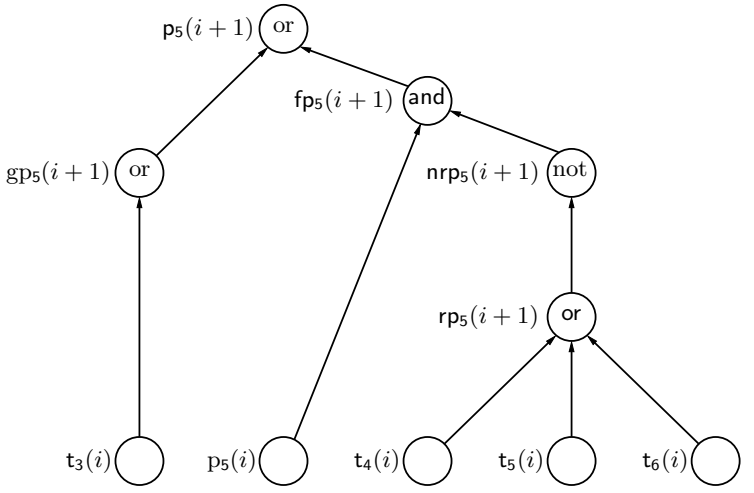


Fig. 3. Example: Translation for the place p_5

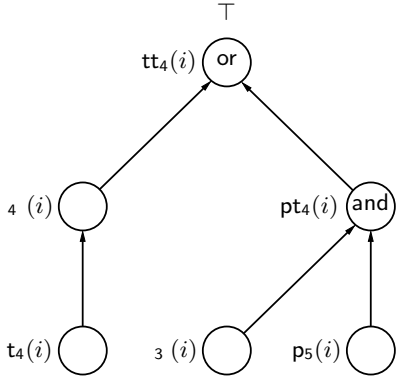


Fig. 4. Example: Translation for the transition t_4

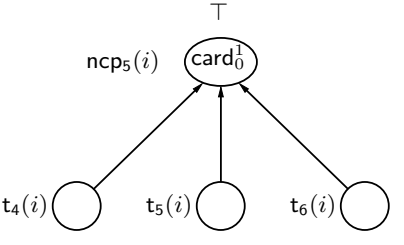


Fig. 5. Example: Translation of the conflicts with respect to place p_5

that at most one of the transitions which have the place p_j in preset can appear in step i. We say that this set of transitions is *in conflict* with respect to the place p_j.

The translation (a)-(b) as given above allows for "idle steps" in which no transition occurs. Thus the program encodes all the step executions of length n or less. We have chosen to remove the possibility of idling steps in our implementation.[3] Thus we always add the following gates to the system:

(c) For each step $0 \leq i \leq n-1$ add an or gate $\mathsf{ni}(i)$ (for non-idle) and constrain it to true. It has the children $\{\mathsf{t_1}(i), \ldots, \mathsf{t_k}(i)\}$, where $\{t_1, \ldots, t_k\} = T$. Thus the gate $\mathsf{ni}(i)$ will be true if at least one transition fires in step i.

We denote by $SC(\Sigma, n)$ (for step circuit) the translation given by (a)-(c). Given a valuation τ of the circuit $SC(\Sigma, n)$, we can obtain the corresponding sequence of markings and steps $M_0, [S_0\rangle, M_1, [S_1\rangle, \ldots, M_{n-1}, [S_{n-1}\rangle, M_n$ by having transition $t_j \in S_i$ iff $\mathsf{t_j}(i)$ is true, and $p_j \in M_i$ iff $\mathsf{p_j}(i)$ is true. Because gates of form $\mathsf{t_j}(i)$ are the only input gates, the mapping from sequences of steps to consistent truth valuations is in fact a bijection.

Lemma 1. *The constrained Boolean circuit $SC(\Sigma, n)$ has a satisfying truth assignment τ iff $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{n-1}[S_{n-1}\rangle M_n$ is a step execution of Σ, where $M_0, [S_0\rangle, M_1, [S_1\rangle, \ldots, M_{n-1}, [S_{n-1}\rangle, M_n$ is the sequence of markings and steps corresponding to τ.*

Thus we get our main result.

Theorem 2. *The constrained Boolean circuit $SC(\Sigma, n)$ encodes step executions of length n.*

4.1 The Interleaving Semantics

Sometimes we would also like to consider the interleaving semantics. It is easy to add a set of constrained gates to the circuit which disallow non-singleton steps.

(i) For each step $0 \leq i \leq n-1$ add an card_0^1 gate $\mathsf{nc}(i)$ (for non-concurrent) and constrain it to true. It has the children $\{\mathsf{t_1}(i), \ldots, \mathsf{t_k}(i)\}$, where $\{t_1, \ldots, t_k\} = T$. Thus the gate $\mathsf{nc}(i)$ will be true if at most one transition fires in step i.

We call the translation given by (a)-(c),(i) the interleaving circuit $IC(\Sigma, n)$.

Theorem 3. *The constrained Boolean circuit $IC(\Sigma, n)$ encodes interleaving executions of length n.*

[3] Here the semantics of the translation differs from the one presented in [9].

4.2 The Process Semantics

The translation for the process semantics is the main contribution of this paper. The main idea behind it is to modify the translation for step semantics in such a way that all step executions which are not in Foata normal form are disallowed.

If one looks at Def. 2 it is easy to see that each transition t in step S_i (not including the initial step S_0) has to have at least one transition t' in step S_{i-1} which generates at least one token to the preset of t. It is now straightforward to enforce this in a local way.

We change the preset of a transition in the following way. The part (b) of the translation is replaced by (b'), which is identical to (b) except that 7 is replaced by the 7' and 7" (see Figure 6 for an example):

(b') For each step $0 \leq i \leq n-1$ we add the following gates (1-6,8-10 omitted):

7'. For each transition $t_j \in T$ we create an or gate $\mathsf{dpt_j}(i)$ (for disjunctive preset) with the children $\{\mathsf{gp_1}(i), \ldots, \mathsf{gp_k}(i)\}$, where $\{p_1, \ldots, p_k\}$ is the preset of t_j. The gate $\mathsf{dpt_j}(i)$ will be true if a token was generated to some preset place of transition t_j in step $i-1$. (The previous step!)

7". For each transition $t_j \in T$ we create an and gate $\mathsf{pt_j}(i)$ with the children $\{\mathsf{p_1}(i), \ldots, \mathsf{p_k}(i), \mathsf{dpt_j}(i)\}$, where $\{p_1, \ldots, p_k\}$ is the preset of t_j. The gate $\mathsf{pt_j}(i)$ will be true if all the preset places of transition t_j in step i contain a token and the transition is locally in Foata normal form.

Note that the child gates of gates added by 7' already existed in the step translation as they are generated by 2. The 7" is almost identical to 7 except that the gate created in 7' has been added to the list of children. The gate generated by 7" now assures that both the preset of the transition is available *and* the transition is locally in Foata normal form. These local constraints on transition enabledness together imply that the step execution will as a whole be in Foata normal form (again according to Def. 2).

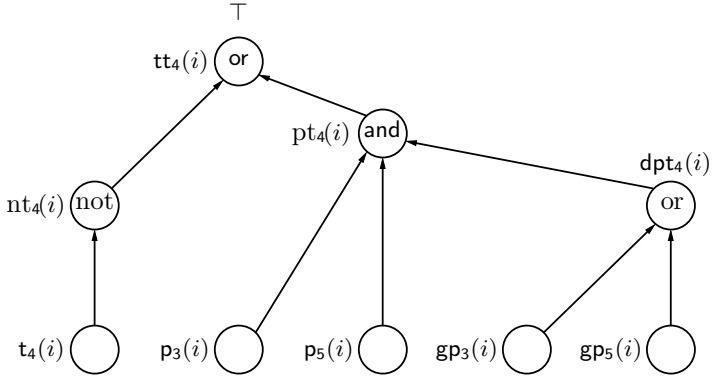


Fig. 6. Example: Process semantics translation of t_4

As in Def. 2, the initial step is special.

(p) For each place $p_j \in P$ we create a gate $\mathsf{gp_j}(0)$ and associate $\top$ with it.

We call the translation given by (a),(p),(b'),(c) the process circuit $PC(\Sigma, n)$. We say that a process π has depth n if the corresponding Foata normal form step execution $FNF(\pi)$ has length n. We have the following result.

Theorem 4. *The constrained Boolean circuit $PC(\Sigma, n)$ encodes processes of depth n.*

4.3 Checking Reachability

We have presented three translations which encode executions with bound n in different semantics. We can now add any Boolean constraint on the final marking M, as given by the syntax $f ::= p \in P \mid \neg f_1 \mid f_1 \vee f_2 \mid f_1 \wedge f_2$. Given a parse tree of the formula f, we convert it to a Boolean circuit $FC(f, n)$ of same size by replacing each atomic proposition $p \in P$ by the gate $\mathsf{p}(n)$, and all other formula types with the corresponding gates having the same children as in the parse tree. Finally the top-level gate f is constrained to true.

Theorem 5. *Let $C(\Sigma, n)$ be one of $PC(\Sigma, n)$, $SC(\Sigma, n)$, $IC(\Sigma, n)$. The constrained Boolean circuit $RC(\Sigma, f, n) = C(\Sigma, n) \cup FC(f, n)$ has a satisfying truth assignment iff there exists a marking M which satisfies f and is reachable in Σ with bound n in (process, step, interleaving) semantics.*

The size of each translation $RC(\Sigma, f, n)$ as the sum of number of gates and connections between them is linear, i.e., $\mathcal{O}((n \cdot (|P| + |T| + |F|)) + |f|)$.[4]

5 Experimental Results

We have implemented the reachability translations described in the previous section in a tool called punroll (for process unroller). We have implemented the following optimization which simplifies away places (transitions) which can never have a token (can never fire). For each step $0 \leq i \leq n-1$:

(i) For each transition $t_j \in T$: If for some place $p \in {}^\bullet t_j$ the gate $\mathsf{p}(i)$ has function $\bot$ associated with it (or alternatively in the process semantics: for all places $p \in {}^\bullet t_j$ the gate $\mathsf{gp}(i)$ has function $\bot$ associated with it), then associate gate $\mathsf{t_j}(i)$ with function $\bot$.

(ii) For each place $p_j \in T$: If for all transitions $t \in {}^\bullet p_j$ the gate $\mathsf{t}(i)$ is associated with $\bot$, then associate the gate $\mathsf{gp_j}(i+1)$ with $\bot$.

(iii) For each place $p_j \in T$: If both gates $\mathsf{p_j}(i)$ and $\mathsf{gp_j}(i+1)$ are associated with $\bot$, then associate $\mathsf{p_j}(i+1)$ with $\bot$.

[4] This bound also holds if we restrict ourselves to Boolean circuits without card_0^1 gates, because in principle each card_0^1 gate with k children can be simulated with (a simple ripple-carry adder style) circuit of size $\mathcal{O}(k)$ which contains only and and or gates.

(iv) Simplify the circuits of step i by substituting $\bot$ when associated by (i)-(iii).

The punroll tool can also add a constraint which requires that the marking reached is a deadlock, as given by the property $f = dead = \neg \bigvee_{t \in T} \bigwedge_{p \in \bullet t} p$. As a constrained satisfiability checker for Boolean circuits we use BCSat [10]. It operates internally on Boolean circuits, and also directly supports card_0^1 gates. BCSat is available from: ¡http://www.tcs.hut.fi/~tjunttil/bcsat/¿.

We use a set of deadlock checking benchmarks collected by Corbett [4]. They have been converted from communicating state machines to nets by Melzer and Römer [13]. The BYZA4_2A example is an exception to this rule, it is from [14]. The models were picked by choosing the nontrivial ones which have a deadlock.

For each model and all three semantics we incremented the used bound n until a deadlock was found. After that we stored the translation using that bound, and report the time for BCSat 0.3 to find the first satisfying truth assignment. In some cases a satisfying truth assignment could not be found within a reasonable time in which case we report the time used to prove that there are no satisfying truth assignments for the circuit with bound n.

The experimental results can be found in Fig. 7. The columns are:

- Problem: The problem name with the size of the instance in parenthesis.
- $|P|$: Number of places in the net.
- $|T|$: Number of transitions in the net.
- Pr. n: The smallest integer n such that a deadlock could be found using the process semantics / in case of $> n$ the largest integer n for which we could prove that there is no deadlock with that bound using the process semantics.
- Pr. s: The time in seconds to find the first satisfying truth assignment / to prove that there is no satisfying truth assignment. (See Pr. n above.)
- St. n and St. s: same as Pr. n and Pr. s but for the step semantics.
- Int. n and Int. s: same as Pr. n and Pr. s but for the interleaving semantics.
- States: Number of reachable states of the net system, or a lower bound $> n$.[5]

The times reported are the average of 5 runs as reported by the /usr/bin/time command on a Linux PC with an AMD Athlon 1GHz processor, 512MB RAM.

The set of experiments we used is too small to say anything conclusive about the performance of the method. There are, however, still some interesting observations to be made. In the experiments the process and step semantics often allow to use a smaller bound to find a deadlock. This partly explains their better performance when compared to the interleaving semantics. The process semantics has better performance than step semantics on e.g., BYZA4_2A, KEY(2), and MMGT(4). Several of the benchmarks (14 out of the 54 circuits used) were solved "with preprocessing" by BCSat, for example DARTES(1) in all semantics. The KEY(x) examples do not have a large number of reachable states, but seem to be still hard for bounded model checking, the results also indicate the reverse to be sometimes true, see e.g., BYZA4_2A with process semantics.

The punroll tool, the net systems, and the circuits used are available from: http://www.tcs.hut.fi/~kepa/experiments/Concur2001/.

[5] These differ from the ones reported in [9], where there unfortunately are some errors.

Problem	$\|P\|$	$\|T\|$	Pr. n	Pr. s	St. n	St. s	Int. n	Int. s	States
BYZA4_2A	579	473	8	5.6	8	179.8	>7	6.8	>2500000
DARTES(1)	331	257	32	0.1	32	1.5	32	1.5	>1500000
DP(12)	72	48	1	0.0	1	0.0	12	1.5	531440
ELEV(1)	63	99	4	0.0	4	0.0	9	0.6	163
ELEV(2)	146	299	6	0.0	6	0.2	12	12.7	1092
ELEV(3)	327	783	8	0.4	8	2.7	15	126.5	7276
ELEV(4)	736	1939	10	5.4	10	67.7	>13	560.5	48217
HART(25)	127	77	1	0.0	1	0.0	>5	0.3	>1000000
HART(50)	252	152	1	0.0	1	0.0	>5	1.3	>1000000
HART(75)	377	227	1	0.0	1	0.0	>5	3.2	>1000000
HART(100)	502	302	1	0.0	1	0.0	>5	5.9	>1000000
KEY(2)	94	92	36	22.7	>27	76.0	>27	30.1	536
KEY(3)	129	133	>30	179.0	>27	198.6	>27	47.3	4923
KEY(4)	164	174	>27	32.9	>27	221.0	>27	58.7	44819
MMGT(2)	86	114	6	0.1	6	0.2	8	1.3	816
MMGT(3)	122	172	7	0.4	7	1.0	10	40.4	7702
MMGT(4)	158	232	8	2.9	8	253.6	>11	476.0	66308
Q(1)	163	194	9	0.1	9	0.2	>17	660.5	123596

Fig. 7. Experiments

6 Conclusions

We have presented how bounded reachability checking for 1-safe Petri nets can be done using constrained Boolean circuits. For step and interleaving semantics these translations can be seen as circuit versions of the logic program translations in [9]. The process semantics translation is new and is based on the notion of Foata normal form for step executions. We have created on an implementation called punroll. We report on a set of benchmarks, where the BCSat tool is used to find whether the constrained circuit is satisfiable or not. The experiments seem to indicate that the process semantics translation is often the most competitive one.

It should be quite straightforward to also use other forms of concurrency than 1-safe net systems with process semantics. The crucial point is to be able to encode the constraints needed for a step execution to be in a Foata normal form in a local manner.

The close connection of bounded reachability checking to AI planning techniques [11,15] needs to be investigated further. It might be useful to use stochastic methods [11] in the verification setting. Also applying process semantics for AI planning needs to be investigated. (Step semantics has been used in [15].)

There are interesting topics for further research. We would like to extend the tool to handle bounded LTL model checking [3]. For interleaving semantics this is quite straightforward, but there are some subtle issues with step and process semantics which need to be solved.

Acknowledgements. The author would like to warmly thank T. A. Junttila and I. Niemelä for creating BCSat, and for fruitful discussions.

References

1. E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.
2. E. Best and C. Fernández. *Nonsequential Processes: A Petri Net View*, volume 13 of *EATCS monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
3. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer, 1999. LNCS 1579.
4. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. Technical report, Department of Information and Computer Science, University of Hawaii at Manoa, 1995.
5. V. Diekert and Y. Métivier. Partial commutation and traces. In *Handbook of formal languages, Vol. 3*, pages 457–534. Springer, Berlin, 1997.
6. J. Esparza. Decidability and complexity of Petri net problems – An introduction. In *Lectures on Petri Nets I: Basic Models*, pages 374–428. Springer-Verlag, 1998. LNCS 1491.
7. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 87–106, 1996. LNCS 1055.
8. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm, 2001. Accepted for publication in Formal Methods for System Design.
9. K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 90–96, Stanford, USA, March 2001. AAAI Press, Technical Report SS-01-01.
10. T. A. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In *Computational Logic – CL 2000; First International Conference*, pages 553–567, London, UK, 2000. LNCS 1861.
11. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press / MIT Press, 1996.
12. H. C. M. Kleijn and M. Koutny. Process semantics of P/T-nets with inhibitor arcs. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, pages 261–281, 2000. LNCS 1825.
13. S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV '97)*, pages 352–363, 1997. LNCS 1254.
14. S. Merkel. Verification of fault tolerant algorithms using PEP. Technical Report TUM-19734, SFB-Bericht Nr. 342/23/97 A, Technische Universität München, München, Germany, 1997.
15. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
16. S. Römer. *Theorie und Praxis der Netzentfaltungen als Basis für die Verifikation nebenläufiger Systeme*. PhD thesis, Technische Universität München, Fakultät für Informatik, München, Germany, 2000.

Techniques for Smaller Intermediary BDDs

Jaco Geldenhuys and Antti Valmari

Tampere University of Technology, Software Systems Laboratory
PO Box 553, FIN-33101 Tampere, FINLAND
{jaco, ava} @cs.tut.fi

Abstract. Binary decision diagrams (BDDs) have proven to be a powerful technique for combating the state explosion problem. Their application to verification is usually centered around the computation of the transitive closure of some binary relation. The closure is usually computed with a *fixed point* algorithm that expands some set until it stops growing. Unfortunately, the BDDs that arise during the computation are often much larger than the final BDD. The computation may thus fail because of lack of memory, even if the final BDD would be small. To alleviate this problem, this paper proposes four variations of the fixed point algorithm. They reduce the sizes of the intermediary BDDs by "rounding down" the sets they represent in such a way that the final BDD does not change. Consequently, more iterations may be required to compute the fixed point, but the intermediary BDDs computed during the run are smaller. The performance of the new algorithms is illustrated with a large number of experiments.

1 Introduction

The greatest obstacle of automated verification techniques based on state exploration is the *state explosion problem*: the number of states grows exponentially in the number of components. One important way of attacking this problem is to represent sets of states as *binary decision diagrams* (*BDD*s) [5] instead of representing each state explicitly.

The use of BDDs relies on representing the individual states of the system under analysis as bit vectors of some fixed length N. There are altogether 2^N such vectors, but usually only a small subset of them represents states that the system can reach. We will say that the N-bit vectors represent 2^N unique *syntactically possible states*, and the reachable states are a subset of the syntactically possible states. Any subset of the 2^N different N-bit vectors can be represented with a BDD. The benefit of BDDs comes from the fact that a small BDD can often represent a huge set of states, and the BDD operations that are needed in verification are cheap as long as the BDDs are small. As a consequence, BDDs make it possible to efficiently manipulate many state spaces with an astronomical number of states.

Instead of the set of reachable states, BDDs can be used for representing other sets of states that are interesting for verification. For instance, a BDD can

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 233–247, 2001.
© Springer-Verlag Berlin Heidelberg 2001

represent the set of states from which an undesirable situation can be reached. One can then check whether any initial state of a system is in this set. *Symbolic model checking* [8] is an advanced application of this idea.

At the heart of these applications of BDDs is a fixed point algorithm that starts from a set of states and computes the closure of the set under some relation by computing a sequence of supersets until the set stops growing. The standard version of this algorithm adds new states in a breadth-first manner.

Although a small BDD can often represent a huge set of states, not all big sets of states can be represented with small BDDs. This leads to the *BDD size explosion problem*. It has been observed that breadth-first computation of the fixed point produces intermediary sets of states that often fail to have small BDD representations, even if the final set of states does [9,17,25,26].

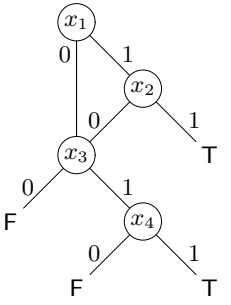
In this paper we present novel algorithms that compute different intermediary sets of states than the breadth-first method, and aim at choosing those sets so that they have small BDD representations. This is different from most, but not all, earlier suggestions for improving the BDD computation of the fixed point. The relationship of our contribution to earlier work is discussed in Section 2.3.

It seems that BDDs are well suited to synchronous systems such as hardware circuits, but have been less successful when it comes to asynchronous systems such as Petri nets or concurrent software, as some authors have pointed out [7]. Here *synchronous* means that the system is driven by a common clock, so that each subsystem makes precisely one transition per clock pulse. (It may, however, be an idling transition.) Our new techniques are at their best with asynchronous systems. They work by "freezing" either variables or whole processes in their initial states while exploring the state space for the rest of the system. Initially, most of the system is frozen and only a few variables or a single process is active. As fixed points are reached, the restricted variables or processes are gradually unfrozen until the entire state space has been explored. Our methods require more iterations than the standard algorithm to reach the final fixed point, but the intermediary BDDs produced during the computation are smaller, leading to improved memory *and* runtime efficiency.

In Section 2 we discuss the context of our work and relate it to similar approaches. The new methods are explained in Sections 3 and 4, and experimental results are presented in Section 5. We offer our conclusions in Section 6.

2 Background and Related Work

A transition system is a tuple $M = (S, R, I)$, where S is a finite set of syntactically possible states, $R \subseteq S \times S$ is the transition relation, and $I \subseteq S$ is the set of initial states. A transition system is defined over a set of v variables $V = \{x_1, x_2, \ldots, x_v\}$. For simplicity, we assume that each x_i is a binary variable taking its values from the set $\{0, 1\}$. (When this does not hold, each variable must be mapped to a set of secondary variables that store the binary encoding of the values of the primary variable.)



(a)

```
1  function Fixed-point(I, R)
2      S' := I
3      repeat
4          S := S'
5          S' := S ∪ Image(S, R)
6      until S' = S
7      return S'
```

(b)

Fig. 1. **(a)** A BDD example: $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$
(b) The standard method for computing the fixed point

In some cases the set of variables is partitioned into components or processes such that $\pi = \{V_1, V_2, \ldots, V_n\}$ (where $V = \cup_{1 \leq i \leq n} V_i$ and $V_i \cap V_j = \emptyset$ when $i \neq j$). This induces a corresponding partition π_R over the transition relation such that $\pi_R = \{R_0, R_1, R_2, \ldots, R_n\}$, where R_i for $i > 0$ is the set of local transitions of process/component i. We have that for $i > 0$

$$R_i = \{(s, s') \mid s \text{ and } s' \text{ differ only in the values they assign to variables in } V_i\}$$

and R_0 is the set of global (or synchronizing) transitions so that $R_0 = R - (R_1 \cup R_2 \cup \cdots \cup R_n)$.

2.1 Binary Decision Diagrams

An *(ordered) binary decision diagram* or *(O)BDD* [5] is a data structure for representing a set of bit vectors of equal length or, equivalently, a Boolean formula. It is a directed acyclic graph in which every vertex has either zero or exactly two successor vertices. Vertices with no output edges are labeled by "F" or "T". Each of the remaining vertices is labeled by a variable, and its output edges are labeled by "0" and "1". Exactly one vertex, the *root*, has no incoming edge. Figure 1(a) shows a BDD that represents the set $\{0011, 0111, 1011, 1100, 1101, 1110, 1111\}$, or the formula $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. A vector $\langle x_1 x_2 x_3 x_4 \rangle$ is in the set if and only if the corresponding path from the root down through the BDD ends with "T", where the "corresponding" path is the one where the output edge from each node is selected according to the value of x_i.

The ordering in which the variables occur in a BDD is fixed. Even after fixing the ordering a set may have several BDD representations, but among them is a unique minimal one. The size of the minimal BDD (that is, the number of nodes in it) may depend crucially on the ordering, and it is difficult to know in advance whether a particular ordering would be good.

Several BDDs over the same set of variables and with the same ordering of those variables can be represented efficiently in one data structure by sharing identical sub-BDDs. This is useful in algorithms that manipulate BDDs.

At the heart of verification with BDDs is the computation of a *fixed point* according to the algorithm in Figure 1(b). It takes as inputs the set of states I and transition relation R and produces the set of states $S_R \subseteq S$ that are reachable from a state in I by zero or more transitions from R.

The algorithm is implemented by encoding the sets I, S and S' and the relation R as BDDs $\mathcal{I}$, $\mathcal{S}$, $\mathcal{S}'$, and $\mathcal{R}$, respectively. Because R is a set of pairs $(s, s') \in S \times S$, the BDD encoding $\mathcal{R}$ has two BDD variables for each variable in V, namely an *old* variable that corresponds to the domain of R (the "s" of (s, s')), and the *new* variable that corresponds to the codomain of R (the "s'" of (s, s')). The BDDs $\mathcal{I}$, $\mathcal{S}$, and $\mathcal{S}'$ all use the new variables.

The fixed point algorithm uses an *Image* operator. The forward image of a set of states S with respect to a transition relation R is the set of states that can be reached from S by precisely one transition from R:

$$\mathit{Image}(S, R) = \left\{ s' \;\middle|\; \exists s \in S : (s, s') \in R \right\}.$$

The algorithm is called "fixed point" because S keeps on growing until no more growth is possible. It computes a sequence $S_0 \subseteq S_1 \subseteq \cdots \subseteq S_n$ of sets such that $S_0 = I$ and $S_n = S_{n-1}$.

2.2 Related Work

The BDD size explosion problem has been addressed in several ways; the literature is extensive, but widely disseminated. The majority of approaches can be classified along the following broad (not necessarily disjoint) lines:

1. Modifications to the BDD structure. A modification of the internal structure of the BDD may lead to greater efficiency in memory and time. One class of structure modifications is the variations on traditional BDDs such as zero-suppressed BDDs [19], algebraic decision diagrams [2], multi-valued decision diagrams [20, 29], and parity ordered BDDs [30]. These approaches enjoy varying degrees of success, but do not solve the problem of large intermediary BDDs.

Also included in this set of techniques is the idea of variable reordering. It is well-known that the order of variables in the BDD encoding of the state space and transition relation can have a dramatic influence on the size of the BDD. Rudell and others have proposed dynamic reordering of the variables [15,28].

2. Alternative representation of the transition relation. Burch et al. partition the transition relation into a set of disjuncts (or conjuncts) [6]. This has become a standard approach to alleviate problems with the computation of the intermediary BDDs, but it leaves the problem of their excessive size unresolved. It has been developed further by Cabodi et al. [9], for instance.

3. Alternative representation of states. Narayan et al. divide the set of syntactically possible states into disjoint partitions, and then represent the intersection

of each partition with the state space of a hardware circuit as a BDD [24]. Different BDD variable orderings may be used for each part, so that the part-BDDs can be small even when no ordering can make a single-BDD representation small. A very similar approach was proposed by Cabodi et al. in [10].

Hu, York and Dill, on the other hand, store the intermediary BDDs as implicitly conjoined BDDs [18]. With each iteration of the algorithm the list of conjuncts grows larger; this makes it necessary to trim the list from time to time by combining conjuncts according to some heuristic.

4. Alternative implementation of the Image *operator.* Coudert, Berthet and Madre suggested using $Image(S_1, R)$ instead of $Image(S, R)$, where S_1 is any set between the $S' - S$ and S' of the previous iteration [14]. This may speed up the individual iteration steps, but produces the same, potentially very big, intermediary BDDs as the standard algorithm.

Other researchers have looked at *iterative squaring*: partial closures of the transition relation are computed beforehand, and used to compute the fixed point in fewer iterations [7,9]. This is possible because when the ith iterative square R^i of the relation is used, the result of $Image(S, R^i)$ is the set of all those states reachable via 1, 2, 3, ..., or 2^i transitions from a state in S. Usually, however, the size and cost of computing iterative squares preclude its use.

5. Mixed breadth- and depth-first search. The fixed point algorithm in Figure 1(b) results in a breadth-first traversal: at each iteration the set of all states reachable via one transition is added to the current set of reachable states. The set of new states is known as the *frontier*. If only a subset of the frontier set is added, the result is a mixture of breadth-first and depth-first search, known as *guided search* [27] or *partial traversal* [11]. The extreme case where only a single state is added during each iteration, and the state is selected as a neighbour of the most recently added state results in a highly inefficient version of depth-first search.

In [26] Ravi and Somenzi investigated techniques to extract a *dense* subset of the frontier. The density of a BDD is defined as the ratio of the minterms it represents to the number of nodes it uses. When the size of the frontier exceeds a preset limit, heuristics are applied to find and replace by F those vertices whose removal has a (hopefully) small effect on the set represented by the BDD.

An alternative to subsetting the frontier set after the image computation, is to restrict the transitions relation that is used in the *Image* operation. User-specified constraints or *hints* have been suggested for this purpose [4,27].

Burch et al. also suggest a modified breadth-first search that is based on computing the fixed point componentwise [7]. What we suggest in this paper can be thought of as a generalisation of that idea.

This list is by no means complete. For example, instead of computing the exact fixed point, it is possible to compute under- and over-approximations more cheaply [13,16]. Depending on where one draws the line between solving and avoiding the problem, other approaches could include the combination of symbolic verification and partial-order (that is, stubborn-set-like) reduction [1], abstraction-based approaches [21], compositional verification [22], and perhaps even using SAT-based methods to avoid the use of BDDs altogether [3].

2.3 Contribution

Of the above-mentioned classes, our method falls primarily in class 5. Like [26] our new methods are based on BDD subsetting. However, instead of heuristics based on the structure of the BDD, two of them are guided by the initial state of the system, and the other two by a division of the system into components. Their operation can thus be intuitively understood at the level of the system under analysis.

Some of the hints suggest by Ravi and Somenzi in [27] correspond almost exactly to one of our proposed methods (V_1). However, while their approach relies on the user to supply the hints, our techniques are fully automatic. There are also similarities at a conceptual level between our work and that of Miner and Ciardo [23], in that both techniques exploit *locality* of transitions. There are however significant differences, both in terms of the context (they use Petri Nets and MDDs) and in the application of these ideas.

The new methods introduced here all rely on the fact that to obtain the correct fixed point it is not necessary to compute $Image(S, R)$ precisely. It suffices that (1) at each iteration step the new states S_{new} that are added to S are a subset of $Image(S, R)$; (2) at each iteration step the algorithm makes progress either by causing S to grow, or by changing the criterion for computing S_{new}; and (3) the algorithm does not terminate before it is certain that no subset of $Image(S, R)$ introduces new states. This fact is exploited by computing intermediary sets of states that have a (hopefully much) smaller BDD representation than the sets produced by the standard algorithm, while still eventually yielding the correct final set of states. Two of our methods additionally rely on partitioning the transition relation.

Our methods thus work by "rounding down" the intermediary BDDs. As a consequence, the number of iteration steps is increased. However, in most of our experiments, this effect was more than compensated for by the fact that, due to smaller intermediary BDDs, the time spent per iteration step is reduced. Thus our methods often save *both memory and time*.

3 Freezing BDD Variables

In this section two of our new methods, V_1 and V_2, are presented. The basic idea of these approaches is to "freeze" most BDD variables to their initial values and to explore the states that involve changing only the unfrozen variables. When a fixed point is reached, more variables are unfrozen and a new fixed point is computed. This continues until all variables are unfrozen and a fixed point is reached. The V_1 and V_2 methods differ only the way that the "freezing" of variables is performed.

To explain the methods, it is necessary to first discuss the details of how the *Image* operator is implemented with BDD operations. Figure 2(a) shows how it is done. In the figure, $\mathcal{S}$ is the BDD encoding for S, and $\mathcal{R}$ is the encoding for R. $\mathcal{S}_0$ is a "shifted" version of $\mathcal{S}$. That is, it is otherwise the same as $\mathcal{S}$, but

$\mathcal{S}_0 := \text{shift}(\mathcal{S})$	$\mathcal{S}_0 := \text{shift}(\mathcal{S})$	$\mathcal{S}_0 := \text{shift}(\mathcal{S})$
$\mathcal{T}_1 := \mathcal{S}_0 \wedge \mathcal{R}$	$\mathcal{T}_1 := \mathcal{S}_0 \wedge \mathcal{R}$	$\mathcal{T}_1 := \mathcal{S}_0 \wedge \mathcal{R} \wedge \mathcal{A}_i$
$\mathcal{S}_2 := \exists s : \mathcal{T}_1$	$\mathcal{S}_2 := (\exists s : \mathcal{T}_1) \wedge \mathcal{I}_i$	$\mathcal{S}_2 := \exists s : \mathcal{T}_1$
$\mathcal{S}' := \mathcal{S} \vee \mathcal{S}_2$	$\mathcal{S}' := \mathcal{S} \vee \mathcal{S}_2$	$\mathcal{S}' := \mathcal{S} \vee \mathcal{S}_2$
(a) Standard algorithm	**(b)** V_1 technique	**(c)** V_2 technique

Fig. 2. The computation of $S' := S \cup \mathit{Image}(S, R)$

it uses the old variables. $\mathcal{T}_1$ is the subset of transitions that start at a state in S. $\mathcal{S}_2$ is the set of target states of the transitions in $\mathcal{T}_1$, and it is obtained by existentially quantifying away all the old variables. It represents $\mathit{Image}(S, R)$. $\mathcal{S}'$ is the encoding of the new, expanded set of reachable states; that is, $S \cup \mathit{Image}(S, R)$. The intermediary BDDs calculated during the computation are $\mathcal{S}_0$, $\mathcal{T}_1$, $\mathcal{S}_2$ and $\mathcal{S}'$.

We now consider two variable freezing techniques named V_1 and V_2.

V_1 Unfreezing is accomplished by computing the new set of states and then cutting away those states where the frozen BDD variables are not in their initial values. In Figure 2(b) this is done in the calculation of $\mathcal{S}_2$. The BDD $\mathcal{I}_i$ denotes the initial state with all non-frozen BDD variables existentially quantified, that is, $\mathcal{I}_i = \exists x'_{i+1} : \exists x'_{i+2} : \cdots \exists x'_n : \mathcal{I}$, where $\mathcal{I}$ represents the set of states with which the fixed point computation is started, and $V_{\mathsf{df}} = \{x'_{i+1}, x'_{i+2}, \ldots, x'_n\}$ is the set of unfrozen BDD variables. Both $\mathcal{I}_i$ and V_{df} use the new BDD variables. When a fixed point is reached, one or more frozen BDD variables are added to V_{df}, until finally it contains all BDD variables, and the algorithm terminates.

V_2 This technique is otherwise similar to V_1, but it implements the freezing of variables in a slightly different way. It uses the conjunction

$$\mathcal{A}_i \stackrel{\mathsf{def}}{=} (x_1 = x'_1) \wedge (x_2 = x'_2) \wedge \cdots \wedge (x_i = x'_i)$$

where $x_1, \ldots, x_i$ are the old and $x'_1, \ldots, x'_i$ the new versions of the frozen variables. $\mathcal{A}_i$ is conjoined with the set of transitions in the computation of $\mathcal{T}_1$ to eliminate the frozen components (see Figure 2(c)).

Why does the freezing of variables reduce the size of intermediary BDDs? Each BDD vertex has two children that may be other BDD vertices or the constants F or T. If the corresponding BDD variable of a BDD vertex is frozen, then one of the children of the vertex is F, and the BDD does not branch at that vertex. Therefore, BDDs with many frozen variables are small.

To have an intuitive image of why the method helps also when most variables are unfrozen, consider a program that consists of, say, ten parallel processes. In the ordinary fixed point algorithm, after five iterations, the set of reached states consists of those where one of the processes has taken at most five steps and the

remaining have taken no steps, plus those where one process has taken at most four steps and another one has taken one step, plus those where one has taken at most three steps and either another one has taken two steps or two other processes have taken one step each, plus ... . The corresponding BDD contains lots of dependencies between the local states of the processes: a process may have progressed far if and only if the other processes are close to their starting point. On the other hand, the final state space does not contain such dependencies.

Tight relationships between variables of the types "if and only if" and "exclusive or" tend to yield big BDDs, unless they hold between BDD variables that are very close to each other in the ordering of the BDD variables. Since all processes cannot simultaneously be close neighbours of each other in the BDD variable ordering, big intermediary BDDs seem unavoidable when processing asynchronous systems with the standard fixed point algorithm. Indeed, the measurements in [25] and Section 5, among others, confirm that the intermediary BDDs may be much bigger than the final BDD.

The freezing and stepwise unfreezing of BDD variables correspond roughly to keeping some of the processes at (or close to) their initial state, while letting the others go as far as they can without getting any signals or messages from the frozen processes. When a frozen process is finally allowed to move, its local state is not in an "if and only if" or "exclusive or" relationship with the states of the other processes, because those processes which were unfrozen earlier have already reached all the states they can reach at this stage, while the others are still frozen in their initial states. The problematic dependencies between the values of BDD variables are thus largely avoided. Unfortunately, this beautiful picture partially breaks down when a newly unfrozen process starts to interact with the earlier unfrozen processes. Even so, the method gives savings in almost every measurement of Section 5, and the savings are often significant.

4 Partially Freezing the Transition Relation

These methods make use of the partitioned transition relation. That is, the transition relation R is represented as a disjunction $R = R_0 \vee R_1 \vee \cdots \vee R_n$. In practice, R_i may, for instance, be the set of transitions that correspond to the atomic actions of the ith process of the system, or even a single atomic statement. We have investigated two partition freezing techniques: P_1 and P_2.

$\mathbf{P}_1$ This method cycles through the partitions trying each subset of transitions until a fixed point is reached. The algorithm terminates when all n partitions have been tried successively without any new states being added. This algorithm is shown in Figure 3.

$\mathbf{P}_2$ This technique is similar to P_1 except that when a partition is tried and yields new states, all transitions are retried from the start.

5 $S' := S \cup \mathit{Image}(S, R_i)$
6 **if** $S' = S$ **then** $i := i + 1$ **else** $i := 0$ **endif**

```
1  function Fixed-point-P1 (I, R_0, R_1, ..., R_{n-1})
2     i := 0; j := 0; S' := I
3     repeat
4        S := S'
5        S' := S ∪ Image(S, R_j)
6        if S' = S then i := i + 1; j := (j + 1) mod (n + 1) else i := 0 endif
7     until i > n
8     return S'
```

Fig. 3. The method P_1 for computing the fixed point

5 Experimental Results

In Table 1 measurements for a series of experiments containing both academic and also more practical examples are given. In general, time and memory consumption is very sensitive to the implementation details of the BDD tool. To have full control over the details we used a BDD implementation of our own.

The first column specifies system parameters, such as the number of components. The other columns contain measurements for the standard algorithm and each of the new techniques. Each column contains two figures: the total number of iteration steps required for the computation of the final fixed point, and the maximum number of BDD vertices required during the computation. The latter number is the largest number of vertices ever needed to simultaneously represent the set of initial states, the transition relation (or partitions), and the most recent intermediary BDD $\mathcal{S}_0$, $\mathcal{T}_1$, $\mathcal{S}_2$ or $\mathcal{S}'$.

In each row of each table, the best number of BDD vertices and the second best number of iterations are shown in boldface. The best number of iterations is always obtained with the standard method. Those cases where a new algorithm needed more BDD vertices than the standard algorithm are shown in italics.

The following examples were used:

1. Dining philosophers: The system consists of a ring of n dining philosophers. Each philosopher has two variables that model the states of his left and right forks (up or down). A philosopher first picks up his left fork, then his right, then puts down his left, and finally his right, returning to his initial state. A fork can only be picked up if the neighbour that shares the fork is not using it.

2. m-Bit counters: An array of n counters of m bits each that have been initialised to zero. During each transition one of the counters is incremented.

3. Sort: An array with n elements containing the numbers $0, \ldots, n-1$ shuffled according to a fixed formula. Each transition consists of an exchange of any two elements that are in the wrong order. The system terminates when the elements are arranged in ascending order.

4. Network of communicating processors: This model is an abstraction of a set of processors that communicate via a shared network. A processor non-deterministically issues a request via the network and increments a local counter of

Table 1. Iterations and maximum intermediary BDD size for various models

	n	Std		V_1		V_2		P_1		P_2	
Dining philosophers	1	2	**14**	4	**14**	**2**	**14**	**2**	**14**	**2**	14
	2	4	75	11	73	**8**	73	9	**67**	11	**67**
	3	7	192	18	177	**14**	177	20	**158**	27	**158**
	4	10	379	25	300	**20**	300	37	260	52	**256**
	5	13	708	32	433	**26**	433	60	385	88	**357**
	10	28	4092	67	1098	**56**	1098	265	1508	503	**952**
	20	58	19432	137	**2428**	**116**	**2428**	1125	8206	3308	2592
	50	148	133852	347	**6418**	**296**	**6418**	7305	101564	45523	11112
	100	-	-	697	**13068**	**596**	**13068**	-	-	348548	37312
	200	-	-	1397	**26368**	**1196**	**26368**	-	-	-	-
	500	-	-	3497	**66268**	**2996**	**66268**	-	-	-	-
	1000	-	-	6997	**132768**	**5996**	**132768**	-	-	-	-
n 3-bit counters	1	8	**31**	10	**31**	**8**	**31**	**8**	**31**	**8**	**31**
	2	15	176	19	89	**16**	89	17	**80**	23	**80**
	5	36	1685	46	299	**40**	299	44	**281**	110	**281**
	10	71	7172	91	**649**	**80**	**649**	89	796	395	796
	20	141	29147	181	**1349**	**160**	**1349**	179	2501	1490	2501
n 4-bit counters	1	16	**47**	18	**47**	**16**	**47**	**16**	**47**	**16**	**47**
	2	31	395	35	129	**32**	129	33	**114**	47	**114**
	5	76	4749	86	420	**80**	420	84	**387**	230	**387**
	10	151	20835	171	**900**	**160**	**900**	169	1082	835	1082
	20	301	85535	341	**1860**	**320**	**1860**	339	3372	3170	3372
n-Element sort	1	1	**2**	3	**2**	**1**	**2**	**1**	**2**	**1**	**2**
	2	2	**43**	6	**43**	3	**43**	**2**	**43**	**2**	**43**
	3	3	147	9	**146**	5	**146**	5	*156*	5	*156*
	4	5	1248	13	**1235**	**8**	**1235**	10	*1590*	11	*1590*
	5	6	2133	16	**2066**	**10**	**2066**	13	*2945*	18	*2945*
	6	8	6356	21	**4842**	**14**	4999	19	5819	29	5819
	7	10	15768	25	**8930**	**17**	9143	24	10410	43	10410
	8	13	98686	30	**43946**	**21**	46713	33	64220	64	65261
	9	14	160977	33	**61282**	**23**	**61282**	36	91540	86	91540
	10	16	663058	39	**167817**	**28**	202764	44	180142	114	183168
Network of communicat. processors	1	3	**54**	7	**54**	**4**	**54**	6	*55*	5	*55*
	2	7	722	19	663	14	663	16	**527**	23	531
	3	10	4933	37	3674	30	3674	29	2237	61	**2202**
	4	13	32795	61	20327	52	20327	**46**	10289	125	**10144**
	5	-	-	91	103937	80	103937	**67**	46803	221	**44044**
	6	-	-	127	474640	114	474640	**92**	190629	355	**182832**
Tree arbiter	2	7	224	16	224	12	224	18	202	18	**201**
	4	22	4188	45	3662	37	3662	93	2452	145	**2127**
	8	61	390601	110	244966	94	244966	470	139785	1255	**64412**
Solitaire	5×3	3	**494**	15	*496*	**9**	*496*	15	*616*	17	*616*
	3×5	3	**545**	11	**545**	6	**545**	6	*611*	7	*611*
	6×3	4	684	17	684	**9**	**683**	23	*869*	23	*869*
	3×6	4	**716**	12	**716**	7	**716**	7	*801*	9	*801*
	4×4	14	17810	32	15865	**18**	17189	59	7549	110	**7159**
	5×4	19	155893	52	143040	**35**	*169904*	99	64906	373	**55690**
	4×5	19	126707	52	112093	**33**	115495	75	56507	182	**48557**
	4×6	23	777273	65	678756	41	773689	116	383244	394	**288113**

outstanding requests. When a server acknowledges the request, the processor can remove the acknowledgment from the network and decrement its counter. Each network slot stores the address (number) of the processor that posted the request, an empty/occupied flag, and a request/acknowledge flag.

5. Tree arbiter: The tree arbiter is a full binary tree of $2n - 1$ cells. The n leaves of the tree can asynchronously request a shared resource. The other cells in the tree arbitrate the requests. When an arbiter cell sees that one or both of

its children requests the resource, the cell (unless it is the root cell) requests the resource from its parent, grants the resource to its requesting children, and then releases the resource to its parent. If both children request the resource, it is granted to the favourite child, and, after it releases it, to the other child. After a child has been granted the resource, the other child becomes the favourite. Each cell has a request flag (indicating that the cell is requesting the resource), a grant flag (indicating that the cell has been granted the resource), and a favourite flag (indicating which child the cell will favour next).

6. Solitaire: The solitaire game is played on a rectangular board with $x \times y$ holes. A peg may jump over a neighbouring peg into an empty hole, after which the neighbour is removed. Initially, all holes contain pegs, except for the hole closest to the center. The game ends when no further jumps are possible.

In most of our measurements, the best algorithm is either V_2 or P_2. The results obtained with V_1 are close to those of V_2, which is not surprising, because the two are very similar. Even the second least successful algorithm needs fewer BDD vertices than the standard algorithm in most of the experiments, and often the advantage is significant. The results were best with the counter and dining philosophers examples. In them, V_1 and V_2 reduced the maximum BDD size to linear. Although V_1 and V_2 compute almost the same thing, they yield somewhat different results. This points out that algorithms and measurements like the ones in this paper are sensitive to seemingly small differences in computation order. The figures in this paper are thus indicative rather than the final truth.

The performance of BDDs is sensitive also to the ordering of the BDD variables. We did not experiment much with different orderings, because the first ordering we tried always proved better than our later attempts.

Our methods introduce yet another ordering: that in which BDD variables are unfrozen or components of a partitioned transition relation applied. In the few cases where we tried to modify the "unfreeze" ordering, no ordering was clearly superior over another, with the exception of the dining philosophers system. Reversing the unfreeze ordering of dining philosophers significantly increases the number of iterations while keeping the number of BDD vertices the same.

Figures 4 and 5 depict the growth of the intermediary BDD sizes for the dining philosophers example with 10 philosophers, and the tree arbiter example with 4 leaves cells. The V_1 and V_2 methods are shown on the left, and the P_1 and P_2 methods on the right. The data for the standard method is the same in the left- and right-hand graphs, but the scales are different. It is clear that the size of the intermediary BDDs are drastically reduced in most cases, but it is equally clear that the number of iterations are drastically increased.

Table 2 shows the results of a further experiment to measure the cost of the increased number of iterations. Here we compare the V_1 method to the standard algorithm for computing the fixed point. The set of reachable states of the dining philosophers model was computed with both algorithms for various values of n. The number of vertices of the resulting BDD, the number of iterations needed to reach the fixed point, the number of BDD vertex records created, the computation time in seconds and the memory consumption in Megabytes

Table 2. Standard algorithm versus V_1 for n dining philosophers model

		standard alg.				V_1			
n	BDD vertices	iterations	created	time	memory	iterations	created	time	memory
5	35	13	1702	1.4	1.4	31	1833	0.6	1.4
10	85	28	9437	1.3	1.8	66	9548	1.2	1.8
20	185	58	40117	4.5	3.3	136	35808	3.5	3.3
30	285	88	50608	72.8	3.9	206	36119	6.4	3.4
40	385	118	92613	288	6.0	276	40480	11.6	3.6
50	485	–				346	44016	17.3	3.6
100	985					696	45241	71.7	3.7
200	1985					1396	50675	347	3.9
500	4985					3496	79738	4790	5.1
700	6985					4896	111694	11700	6.1
f	$10n - 15$	$3n - 2$				$7n - 4$			

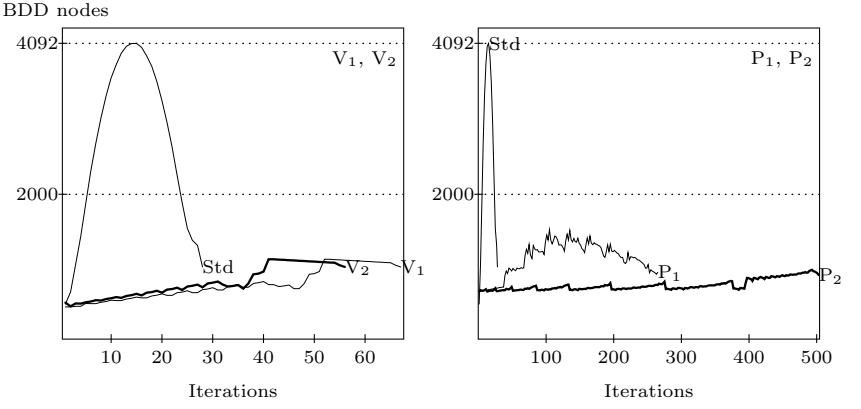


Fig. 4. Growth of intermediary BDDs for the 10 dining philosophers model

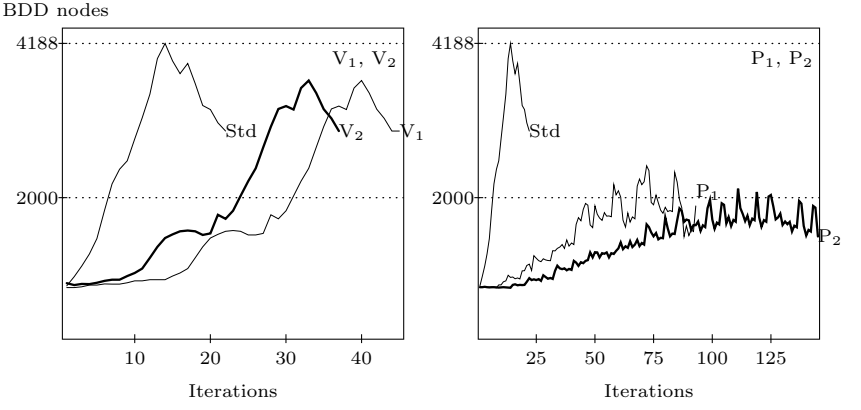


Fig. 5. Growth of intermediary BDDs for the tree arbiter with 4 leave cells

(10^6, not 2^{20}) were recorded. For some columns a formula that matches all the experimental figures in the column is given in the bottom row. This experiment was run on a Pentium 100 MHz processor and 16 Mbytes of memory.

The number of BDD vertex records created during the computation includes the vertices needed to represent the initial marking and the transition relation, and it also includes garbaged vertices. It is thus very much affected by the details of the triggering of garbage collection. The size of the final BDD, given in the second column, does not include any of these.

Although the time and memory consumption figures are only indicative, the superiority of V_1 over the standard algorithm in the dining philosophers system is beyond doubt.

6 Conclusions

We suggested four modifications to the standard fixed point algorithm that uses BDDs for computing the set of states reachable from a given set of states by repeated applications of a given binary relation.

BDDs have been reported to not work very well with asynchronous systems, such as communication protocols or Petri nets. We discussed a possible intuitive reason for this, and pointed out that our new methods attack that reason. And not entirely without success: in the biggest experiments where we were able to complete the standard algorithm, the best maximum BDD size with our new methods was 4.8% (philosophers), 11%, 4.6%, 2.2% (counters), 25% (sort), 31% (network), 16% (arbiter), and 37% (solitaire) of the maximum BDD size with the standard method. In certain academic examples, like the dining philosophers, our best method worked like a dream, but, as is usual with enhanced verification methods, the results with more realistic examples were less spectacular.

Because V_1 and V_2 operate at the level of the BDD variables, they can be used without having any other information than the set of initial states and transition relation. They can thus be implemented within a BDD verification library and hidden totally from the user.

Several matters bear further investigation: the order in which variables and processes are unfrozen, and the number of variables unfrozen in each step. On the other hand, unfreezing BDD variables one process at a time is a natural heuristic for reducing the number of iterations (and we used it in Section 5). In this way extra information from the user can be useful, although it is not mandatory. The algorithms P_1 and P_2 require that the transition relation has been partitioned by the user or someone/something else, and there has been some work done on the automatic decomposition of (synchronous) systems [12].

Because of its good performance and ease of use we believe that V_2 would be a valuable addition to BDD-based verification tools for asynchronous systems.

Acknowledgements. The work of J. Geldenhuys was funded by the Academy of Finland, project UVER.

References

1. R. Alur, R.K. Brayton, T. Henzinger, S. Qadeer, & S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV'97: Proc. 9th Intl. Conf. Computer-Aided Verification*, LNCS #1254, pp. 340–351. Springer-Verlag, Jun 1997.
2. R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, & F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 188–191, Nov 1993.
3. A. Biere, A. Cimatti, E.M. Clarke, & Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #1579, pp. 193–207. Springer-Verlag, Mar 1999.
4. R. Bloem, I.-H. Moon, K. Ravi, & F. Somenzi. Approximations for fixpoint computations in symbolic model checking. In *Proc. World Multiconference on Systemics, Cybernetics and Informatics*, Vol. VIII, Part II, pp. 701–706, 2000.
5. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, Aug 1986.
6. J.R. Burch, E.M. Clarke, & D.E. Long. Symbolic model checking with partitioned transition relations. In *Proc. IFIP Intl. Conf. Very Large Scale Integration*, pp. 49–58, Aug 1991.
7. J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, & D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, Apr 1994.
8. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, & L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, Jun 1992.
9. G. Cabodi, P. Camurati, L. Lavagno, & S. Quer. Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In *Proc. 34th ACM/IEEE Conf. Design Automation*, pp. 728–733, Jun 1997.
10. G. Cabodi, P. Camurati, & S. Quer. Improved reachability analysis of large finite state machines. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 354–360, Nov 1996.
11. G. Cabodi, P. Camurati, & S. Quer. Improving symbolic traversals by means of activity profiles. In *Proc. 36th ACM/IEEE Conf. Design Automation*, pp. 306–311, Jun 1999.
12. H. Cho, G.D. Hachtel, E. Macii, B. Plessier, & F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1451–1464, Dec 1996.
13. H. Cho, G.D. Hachtel, E. Macii, B. Plessier, & F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1465–1478, Dec 1996.
14. O. Coudert, C. Berthet, & J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. Intl. Workshop on Automatic Verification Methods for Finite State Systems*, LNCS #407, pp. 365–373. Springer-Verlag, Jun 1989.
15. M. Fujita, Y. Matsunaga, & T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proc. European Design Automation Conference*, pp. 50–54, 1991.

16. S.G. Govindaraju, D.L. Dill, A.J. Hu, & M.A. Horowitz. Approximate reachability with BDDs using overlapping projections. In *Proc. 35th ACM/IEEE Conf. Design Automation*, pp. 451–456, Jun 1998.
17. A.J. Hu. *Efficient Techniques for Formal Verification Using Binary Decision Diagrams*. PhD thesis, Stanford Univ., 1995.
18. A.J. Hu, G. York, & D.L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proc. 31th ACM/IEEE Conf. Design Automation*, pp. 276–282, Jun 1994.
19. S.-i. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Conf. Design Automation*, pp. 272–277, Jun 1993.
20. T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, Electronics Research Laboratory, Univ. of California at Berkeley, 1995.
21. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Princeton University Press, 1994.
22. D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon Univ., 1993.
23. A. Miner & G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. 20th Intl. Conf. Application and Theory of Petri Nets*, LNCS #1639, pp. 6–25. Springer-Verlag, Jun 1999.
24. A. Narayan, A.J. Isles, J. Jain, R.K. Brayton, & A.L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 388–393, Nov 1997.
25. E. Pastor, O. Roig, J. Cortadella, & R. Badia. Petri net analysis using boolean manipulation. In *Proc. 15th Intl. Conf. Application and Theory of Petri Nets*, LNCS #815, pp. 416–435. Springer-Verlag, 1994.
26. K. Ravi & F. Somenzi. High-density reachability analysis. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 154–158, Nov 1995.
27. K. Ravi & F. Somenzi. Hints to accelerate symbolic traversal. In *Proc. 10th IFIP WG 10.5 Intl. Conf. Correct Hardware Design and Verification Methods*, LNCS #1703, pp. 250–264. Springer-Verlag, Sept 1999.
28. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 42–47, Nov 1993.
29. A. Srinivasan, T. Kam, S. Malik, & R.K. Brayton. Algorithms for discrete function manipulation. In *Proc. ACM/IEEE Intl. Conf. Computer-Aided Design*, pp. 92–97, Nov 1990.
30. S. Waack. On the descriptive and algorithmic power of parity ordered binary decision diagrams. In *Proc. 14th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS #1200, pp. 201–212. Springer-Verlag, Feb 1997.

An Algebraic Characterization of Data and Timed Languages

Patricia Bouyer[1⋆], Antoine Petit[1⋆⋆], and Denis Thérien[2⋆⋆⋆]

[1] LSV, CNRS UMR 8643, ENS de Cachan
61 Av. du Président Wilson
94235 Cachan Cedex, France
{bouyer, petit}@lsv.ens-cachan.fr
[2] School of Computer Science, McGill University
3480 University
Montréal, QC, Canada, H3A 2A7
denis@cs.mcgill.ca

Abstract. Algebra offers an elegant and powerful approach to understand regular languages and finite automata. Such framework has been notoriously lacking for timed languages and timed automata. We introduce the notion of monoid recognizability for data languages, which include timed languages as special case, in a way that respects the spirit of the classical situation. We study closure properties and hierarchies in this model, and prove that emptiness is decidable under natural hypotheses. Our class of recognizable languages properly includes many families of deterministic timed languages that have been proposed until now, and the same holds for non-deterministic versions.

1 Introduction

The class of regular languages can be characterized in various ways: finite automata, rational expressions, monadic second order logic, extended temporal logics, finite monoids... [RS97]. All these characterizations constitute not only one of the cornerstones of theoretical computer science but also form the fundamental basis for much more practical research on verification (see *e.g.* [CGP99]). Among all these equivalences, the simplest is undoubtedly the *purely algebraic* one claiming that a word language is regular if and only if it is monoid recognizable *i.e.* it is the inverse image by a morphism of some subset of a finite monoid.

In the framework of timed languages, very useful to specify and verify real-time systems, the situation is far from being so satisfactory. The original class of timed automata, proposed by Alur and Dill [AD94] has a decidable emptiness problem, but is not closed under complement. Several logical characterizations

⋆ Research partly supported by the French project RNRT "Calife"
⋆⋆ Research partly supported by the French-India project CEPIPRA n°2102 – 1
⋆⋆⋆ Research supported by NSERC, FCAR, and the von Humboldt Foundation

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 248–261, 2001.
© Springer-Verlag Berlin Heidelberg 2001

[Wil94,HRS98] or even Kleene-like theorems [ACM97,Asa98,BP99,BP01] have been proposed for the whole class of timed automata but no purely algebraic one. Interesting subclasses of timed automata, closed under complement, have been proposed and often logically characterized. For instance, (recursive) event clocks automata [AFH94] are closed under complement and can be characterized in a nice logical way [HRS98]. But once again, even if a related notion of counter-free timed languages has been defined, no algebraic characterization exists.

For the first time, at least to our knowledge, we propose in this paper a purely algebraic characterization for timed languages. In fact, we deal with a more general framework than timed languages, the so-called *data languages*. We consider a finite alphabet of actions Σ and a set of data $\mathcal{D}$ (this set of data could be some time domain but also anything else). A data word is thus a sequence of pairs (a, d) where $a \in \Sigma$ and $d \in \mathcal{D}$.

We propose to use a finite fixed number of registers to store the data. When a new letter (a, d) is read, the data is kept or not depending only on the letter a and on the value of the current computation in the finite monoid M. Then the new value of the computation is calculated from the previous value, the current letter a and some *finite and bounded* information from the registers. Hence, the precise values of the registers are never used by the monoid. Only a finite bounded amount of information is needed to decide whether a data word is in the language or not.

We obtain in this way, for any set of actions Σ and set of data $\mathcal{D}$, a class of so-called "monoid recognizable" data languages. This class is closed under boolean operations. As first result, which shows the interest of our approach, the choice of the monoid is fundamental. More precisely, we prove that, like in the formal language case, if two monoids are such that none of them divides the other, then the corresponding class of data languages are incomparable. We then study the exact power of the number of registers.

We next define a notion of deterministic data automata and, as one of our two main theorems, we prove that a data language is monoid recognizable if and only if it is accepted by some data automaton. Note that the translation from monoid to automaton and vice versa is simple and very close to what happens in formal language theory, which emphasizes once more the elegance of the proposed approach.

We then focus on the problem of deciding emptiness of languages recognized by data automata, or equivalently, monoid recognizable. We propose a simple and nice condition related to the registers and the data domain under which emptiness is decidable. More precisely, under this condition, we propose our second main result: an algorithm to transform a data automaton $\mathcal{A}$ into a finite automaton recognizing the classical formal language of those words of Σ^* that can be obtained from a data word accepted by $\mathcal{A}$ by erasing the data. The idea of this construction is similar to the region automaton construction of Alur and Dill [AD94].

If the set of data $\mathcal{D}$ is a time domain, our recognizable data languages contain all the timed languages recognized by deterministic timed automata [AD94] or their deterministic extensions [DZ98,CG00]. But our class also contains a lot of timed languages which cannot be recognized by any timed automata (even non-deterministic ones).

We also briefly study two possible extensions of our model. First, we consider non-deterministic data automata (or equivalently a non-deterministic notion of monoid recognizability). Then we get a larger class of data languages, still closed under union and intersection but not anymore by complementation. On the contrary, this new class is closed by concatenation and iteration. Once again, emptiness can be decided, by an algorithm similar to the one used in the deterministic case. Second, we show that if we extend the power of the registers and allow computations to be performed on them, then what monoid is used to recognize the language becomes essentially irrelevant.

This paper contains only sketches of proofs. The complete proofs are available in the technical report [BPT01].

2 Basic Definitions

If Z is any set, Z^* denotes the set of finite sequences of elements in Z. We consider throughout this paper a finite alphabet Σ and an unrestricted set of data $\mathcal{D}$. Among the elements of $\mathcal{D}$, we distinguish a special initial value, denoted by $\perp$. A *data word* over Σ and $\mathcal{D}$ is a finite sequence $(a_1, d_1) \dots (a_p, d_p)$ of $(\Sigma \times \mathcal{D})^*$. A data language is a set of data words. If $k \geq 1$ is the number of registers, a *register update*, or simply an *update*, $\overline{up}$, is defined by a subset of $\{1, \dots, k\}$. This update induces a function up from $\mathcal{D}^k \times \mathcal{D}$ into $\mathcal{D}^k$ mapping $((d_i)_{i=1..k}, d)$ to $((d'_i)_{i=1..k})$ where $d'_i = d$ if $i \in \overline{up}$ and $d'_i = d_i$ otherwise. In the sequel, we will simply write $((d'_i)_{i=1..k}) = up((d_i)_{i=1..k}, d)$. If $\sim$ is an equivalence defined on $\mathcal{D}^k$ and if $\theta \in \mathcal{D}^k$, we denote $\overline{\theta}$ the class of θ modulo $\sim$.

3 Monoid Recognizability

A first naive attempt to define a notion of monoid recognizability for data languages could simply consider a morphism from the free monoid $(\Sigma \times \mathcal{D})^*$ to some finite monoid M. But the corresponding class of languages would be very restricted. Since the image of $\Sigma \times \mathcal{D}$ would be finite, the simple language $\{(a, d)(a, d') \mid d \neq d'\}$ would not be monoid recognizable as soon as $\mathcal{D}$ is infinite. Similar unsuccessful attempts have been studied in details [ABB80] in the framework of formal languages over an infinite alphabet. We thus need to have a finite mechanism to take into account the data. But, in order to maintain the relevance of the monoid, this mechanism should be very simple and in particular unable to perform any computation. We propose to use, as mechanism, a finite fixed number of registers to store the data. These registers are used through a notion of updates as defined in the previous section. Note that such updates

decide to store or not a data independently of this data or of the values of the registers.

Definition 1. *Let L be a data language over Σ and $\mathcal{D}$, let M be a finite monoid. We say that M* recognizes *L if there exists a subset P of M, an integer k, for each pair $(m,a) \in M \times \Sigma$ an update $up_{m,a}$ over k registers, an equivalence of finite index $\sim$ on $\mathcal{D}^k$, and a morphism $\varphi : (\Sigma \times \mathcal{D}^k/\!\sim)^* \longrightarrow M$ such that: a data word $(a_1,d_1)\dots(a_n,d_n)$ is in L if and only if the sequences $(\theta_i)_{i=0\dots n}$ and $(m_i)_{i=0\dots n}$ defined by:*

$$\begin{cases} \theta_0 = \bot^k \\ \theta_{i+1} = up_{m_i,a_{i+1}}(\theta_i, d_{i+1}) \end{cases} \quad \text{and} \quad \begin{cases} m_0 = 1_M \\ m_{i+1} = m_i \ \varphi(a_{i+1}, \overline{\theta_{i+1}}) \end{cases}$$

satisfy $m_n \in P$.

When a new letter (a_{i+1}, d_{i+1}) is read, the data is kept or not (as determined by $up_{m_i,a_{i+1}}$) depending thus only on the letter a_{i+1} and on the value m_i of the current computation in the finite monoid M. Then the new value m_{i+1} of the computation is calculated from the previous value m_i, the current letter a_{i+1} and some *finite and bounded* information, $\overline{\theta_{i+1}}$, from the registers. Hence, the precise values of the registers are never used by the monoid. Only a finite bounded amount of information is needed to decide whether a data word is in the language or not. Note that, the sequences $(\theta_i)_{i=0\dots n}$ and $(m_i)_{i=0\dots n}$ associated with a given data word are unique. A data language is said to be *monoid recognizable* if there exists some finite monoid recognizing it.

Note that the definition proposed in [KF94], also based on registers, is much more restrictive than ours. In particular their languages have the following property, if $(a_1,d_1)\dots(a_n,d_n)$ is in the language then for any bijection ν from $\mathcal{D}$ into $\mathcal{D}$, $(a_1,\nu(d_1))\dots(a_n,\nu(d_n))$ has also to be in the language. We do not require at all such a property.

Example 1. The data language $L = \{(a,d)(a,d') \mid d \neq \bot, d \neq d'\}$ over $\{a\}$ and $\mathcal{D}$ is recognized by the finite monoid $M = \{1, y, y^2, 0\}$ with $y^3 = 0$ and $0x = x0 = 0$ for any $x \in M$. We use two registers and we define two classes over $\mathcal{D}^2$: $\overline{\theta_{\neq}} = \{(d,d') \mid d \neq d'\}$ and $\overline{\theta_{=}} = \mathcal{D}^2 \setminus \overline{\theta_{\neq}}$. We define a morphism $\varphi : (\{a\} \times \{\overline{\theta_{\neq}}, \overline{\theta_{=}}\})^* \longrightarrow M$ by $\varphi(a, \overline{\theta_{\neq}}) = y$, $\varphi(a, \overline{\theta_{=}}) = 0$. Our updates are $up_{1,a} = \{1\}$ and if $z \in M \setminus \{1\}$, $up_{z,a} = \{2\}$. With these definitions, L is accepted by M (with $P = \{y^2\}$). As an example of computation, consider the data word $(a,d)(a,d')$ with $d \neq \bot$ and $d \neq d'$.

In the monoid M	1_M	$\xrightarrow[d]{a}$	y	$\xrightarrow[d']{a}$	y^2
Two registers	$\begin{pmatrix}\bot\\ \bot\end{pmatrix}$		$\begin{pmatrix}d\\ \bot\end{pmatrix}$		$\begin{pmatrix}d\\ d'\end{pmatrix}$
Equivalence classes	$\overline{\theta_{=}}$		$\overline{\theta_{\neq}}$		$\overline{\theta_{\neq}}$

We must notice that the registers do not compute anything. For example, taking $\mathcal{D} = \mathbb{Q}$, with only one register we could have computed the difference $d' - d$

instead of putting the data d' in a second register. But this is not allowed in our model.

Example 2. The data language $\{(a, d_1) \dots (a, d_n)(a, d) \mid d \notin \{d_1, \dots, d_n\}\}$ over $\{a\}$ and $\mathcal{D}$ (where $\mathcal{D}$ is infinite) is not recognized by any finite monoid. Intuitively, an unbounded number of data should be stored, which is not allowed.

Remark 1. This definition of monoid recognizability is a quite natural extension of the monoid recognizability in formal language theory. Namely, if $\mathcal{D}$ reduces to $\{\perp\}$, then Σ and $\Sigma \times \mathcal{D}$ are in bijection and a formal language is recognizable if and only if its image is a monoid recognizable data language. It can also be shown that if $\mathcal{D}$ is finite and if $L \subseteq (\Sigma \times \mathcal{D})^*$ is a monoid recognizable data language, then L is also a recognizable formal language.

If M is a finite monoid and k an integer, the set of data languages over Σ and $\mathcal{D}$ recognized by M using k registers, is denoted by $\mathcal{L}_{M,k}(\Sigma, \mathcal{D})$, or simply $\mathcal{L}_{M,k}$. We also set $\mathcal{L}_M = \bigcup_k \mathcal{L}_{M,k}$ and $\mathcal{L}_k = \bigcup_M \mathcal{L}_{M,k}$. The set of recognizable data languages has many interesting closure properties.

Proposition 1. *The set $\mathcal{L}_{M,k}$ is closed under complementation. If $L_1 \in \mathcal{L}_{M_1,k_1}$ and $L_2 \in \mathcal{L}_{M_2,k_2}$, then $L_1 \cup L_2$ and $L_1 \cap L_2$ are in $\mathcal{L}_{M_1 \times M_2, k_1+k_2}$.*

From the algebraic point of view, the soundness of our definition is assessed by the following result. It shows essentially that increasing the number of registers cannot help if the monoid is not powerful enough. Hence the structure of the monoid is really fundamental and plays a role similar to what happens in the framework of formal languages.
If M and M' are monoids, recall that M *divides* M' if M is a quotient of a submonoid of M' [Pin86].

Proposition 2. *If M and M' are finite monoids such that neither M divides M' nor M' divides M, then $\mathcal{L}_M \neq \mathcal{L}_{M'}$.*

Proof. Let L be a language on Σ and define

$$L_{\mathcal{D}} = \{(a_1, d_1) \dots (a_n, d_n) \mid a_1 \dots a_n \in L \text{ and } d_i \in \mathcal{D}\}$$

Let M be a finite monoid. Then, we will prove that

$$L \text{ is recognized by } M \iff L_{\mathcal{D}} \text{ is recognized by } M$$

This will establish the result as the hypothesis on M and M' implies that the class of formal languages they respectively recognize are incomparable.
We prove the two implications separately:
Assume that L is recognized by M. There exists a morphism $\varphi : \Sigma^* \longrightarrow M$ and some $P \subseteq M$ such that $L = \varphi^{-1}(P)$. Taking $k = 0$, it is easy to see that M recognizes $L_{\mathcal{D}}$.

Assume that $L_{\mathcal{D}}$ is recognized by M with k, $\sim$, φ as in Definition 1. In particular, if $a_1 \dots a_n$ is in Σ^*, the image of the data word $(a_1, \bot) \dots (a_n, \bot)$ in $(\Sigma \times \mathcal{D}^k/_\sim)^*$ is $(a_1, \overline{\bot^k}) \dots (a_n, \overline{\bot^k})$. We define a morphism $\psi : \Sigma^* \longrightarrow M$ by $\psi(a) = \varphi((a, \overline{\bot^k}))$. Then,

$$\begin{aligned} a_1 \dots a_n \in L &\iff (a_1, \bot) \dots (a_n, \bot) \in L_{\mathcal{D}} \\ &\iff \varphi((a_1, \overline{\bot^k}) \dots (a_n, \overline{\bot^k})) \in P \\ &\iff \psi(a_1 \dots a_n) \in P \end{aligned}$$

Thus, M recognizes the language L and the conclusion easily follows. □

The following statements make precise the relative role of the monoid and of the registers. For example, each additional register strictly increases the class of languages being recognized, as in timed automata each additional clock increases also the power of the automata [HKWT95]. On the other hand, if the monoid and the alphabet are fixed, then the hierarchy on registers collapse.

Proposition 3. *1. If M is a fixed finite monoid, the sequence $(\mathcal{L}_{M,k}(\Sigma, \mathcal{D}))_k$ collapses, more precisely, $\mathcal{L}_{M,2^{|\Sigma \times M|}-1} = \mathcal{L}_{M,2^{|\Sigma \times M|}}$.*

2. The sequence $(\mathcal{L}_k(\Sigma, \mathcal{D}))_k$ is strictly monotonic.

3. Let M_0 be the finite monoid $\{1, 0, x\}$ with $x^2 = x$. For each integer k, there exists a finite alphabet Σ_k and a language L_k over Σ_k such that $L_k \in \mathcal{L}_{M_0,k}(\Sigma_k, \mathcal{D}) \setminus \bigcup_{k'<k} \mathcal{L}_{k'}$.

Proof (Sketch).

1. Updates are parameterized by a pair of $M \times \Sigma$, thus, considering a data language recognized by a finite monoid M with k registers, we define the function

$$\begin{aligned} \lambda : \{1 \dots k\} &\longrightarrow \{up_{m,a} \mid (m, a) \in M \times \Sigma\} \\ i &\longmapsto \{up_{m,a} \mid (m, a) \in M \times \Sigma \text{ and } i \in up_{m,a}\} \end{aligned}$$

   and the equivalence $i \cong j \iff \lambda(i) = \lambda(j)$. Intuitively, if $i \cong j$, the registers i and j play the same role, in that they are updated exactly at the same steps. Thus, we can keep only one register for each class. Moreover, the class of registers i such that $\lambda(i) = \emptyset$ is not useful.
2. The data language $L_k = \{(a, d_1) \dots (a, d_n) \mid i \equiv j \mod (k-1) \implies d_i = d_j\}$ over $\{a\}$ and $\mathcal{D}$ ($\mathcal{D}$ is supposed to be infinite) is recognized by a finite monoid with k registers, but is recognized by no finite monoid with strictly less than k registers.
3. We define $\Sigma_k = \{a_0, a_1, \dots, a_{k-1}\}$ and for each $i = 1 \dots k-1$, we define a function μ_i such that for each data word $u \in (\Sigma_k \times \mathcal{D})^*$, $\mu_i(u)$ is the data d (if it exists) such that $u = u'\ (a_i, d)\ u''$ where u'' does not contain any a_i. If this data does not exists, $\mu_i(u)$ is $\bot$. We define the data language

$$\begin{aligned} L'_k = \{u\ (a_0, d'_1) \dots (a_0, d'_n) \mid u \in ((\Sigma_k \setminus \{a_0\}) \times \mathcal{D})^* \\ \text{and for each } j,\ d'_j \in \cup_{i=1}^{k-1} \{\mu_i(u)\}\ \} \end{aligned}$$

   The language L'_k is recognized by M_0 using k registers but is recognized by no finite monoid using strictly less than k registers. □

4 Data Automata

In this section, we define a notion of recognizability by data automata and prove its equivalence with monoid recognizability.

Definition 2. *A* data automaton *over Σ and $\mathcal{D}$ is a tuple $\mathcal{A} = (Q, q_0, F, k, \sim, T)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, k is an integer, $\sim$ is an equivalence relation of finite index defined on $\mathcal{D}^k$ and $T \subseteq (Q \times \mathcal{D}^k/_\sim \times \Sigma \times \mathcal{U} \times \mathcal{D}^k/_\sim \times Q)$ is a finite set of transitions ($\mathcal{U}$ is a set of updates) such that the following determinism hypothesis holds: for each tuple $(q, g, a) \in Q \times \mathcal{D}^k/_\sim \times \Sigma$, there is a (unique) update up such that any transition $(q, g, a, up', g', q') \in T$ satisfies $up' = up$ and if (q, g, a, up, g', q'_1) and (q, g, a, up, g', q'_2) are in T, then $q'_1 = q'_2$.*

A data word $(a_1, d_1) \dots (a_n, d_n)$ is accepted by the data automaton $\mathcal{A}$ if there exists a path in $\mathcal{A}$

$$q_0 \xrightarrow[d_1]{g_1, a_1, up_1, g'_1} q_1 \xrightarrow[d_2]{g_2, a_2, up_2, g'_2} q_2 \dots \xrightarrow[d_n]{g_n, a_n, up_n, g'_n} q_n$$

such that the sequence $(\theta_i)_{i=0\dots n}$ defined by

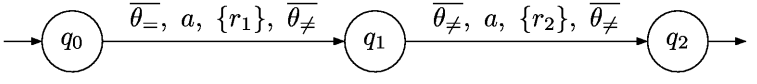
$$\theta_0 = \bot^k \quad \text{and} \quad \theta_{i+1} = up_{i+1}(\theta_i, d_{i+1})$$

satisfies $\overline{\theta_{i-1}} = g_i$ for $1 \leq i \leq n$, $\overline{\theta_i} = g'_i$ for $1 \leq i \leq n$ and $q_n \in F$.
The set of data words that are accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Example 3. The data language described in Example 1,

$$L = \{(a, d)(a, d') \mid d \neq \bot,\ d \neq d'\}$$

is recognized by the following data automaton ($\overline{\theta_=}$ and $\overline{\theta_\neq}$ are defined in Example 1):



We claim that this notion of recognizability by data automata is equivalent to the notion of monoid recognizability in the following sense:

Theorem 1. *Let L be a data language over Σ and $\mathcal{D}$. Then L is recognized by a data automaton if and only if it is recognized by a finite monoid.*

We thus have a result similar to the formal language case. As it appears below, the transformations from monoids to automata and from automata to monoids are very close to the ones used in formal languages. We believe that this similarity emphasizes the appropriateness of our approach.

Proof (Sketch). **If Implication.** First, assume that $L \subseteq (\Sigma \times \mathcal{D})^*$ is recognized by a finite monoid M, with the notations of Definition 1. We construct a data automaton over Σ and $\mathcal{D}$, $\mathcal{A} = (Q, q_0, F, k, \sim, T)$, as follows:

- k and $\sim$ comes from the monoid recognizer,
- $Q = M$, $q_0 = 1_M$ and $F = P$,
- $T = \{(m, g, a, up_{m,a}, g', m') \mid m \in M,\ g, g' \in \mathcal{D}^k/_\sim,\ a \in \Sigma,\ m' = m\varphi(a, g')\}$.

One can prove that $\mathcal{A}$ is a valid deterministic data automaton and that the language accepted by $\mathcal{A}$ is precisely L.

Only If Implication. Now, assume that $L \subseteq (\Sigma \times \mathcal{D})^*$ is recognized by the data automaton $\mathcal{A} = (Q, q_0, F, k, \sim, T)$. We define M as the set of functions from $Q \times \mathcal{D}^k/_\sim$ into itself. We claim that L is recognized by M. The morphism $\varphi : (\Sigma \times \mathcal{D}^k/_\sim)^* \longrightarrow M$ is induced by the function $(a, g') \longmapsto \Big[(q, g) \mapsto (q', g')\Big]$ where q' is the unique state for which there exists a transition (q, g, a, up, g', q') in $\mathcal{A}$ (the unicity of q' comes from the determinism of $\mathcal{A}$). For any $m \in M$, suppose $m((q_0, g_0)) = (q, g)$ (g_0 is the equivalence class of $\bot^k$). Because of determinism again, for any a, there is a unique up such that there exists a transition $(q, g, a, up, -, -)$ and we define $up_{m,a} = up$. We finally define $P = \{m \mid m((q_0, \overline{\bot^k})) \in F \times \mathcal{D}^k/_\sim\}$. It is then possible to prove that $L(M) = L$. The equivalence between monoids and automata is thus proved. □

We can notice that the translations from monoids to automata and vice-versa do not change neither the set of updates, nor the number of registers and the equivalence.

We say that a data language is *recognizable* if it is recognized by some data automaton (which is equivalent to being recognized by a finite monoid).

5 Decidability of the Emptiness Problem

We first note that the general class of recognizable data languages is undecidable: we can easily simulate a two counter machine [Min67] using a data automaton. We propose a condition that determines a class of data automata for which the emptiness problem is decidable.

As a preliminary, given a register update up, we define a relation on $\mathcal{D}^k/_\sim$, denoted by $\xrightarrow{up}$, in the following way:

$$\overline{\theta} \xrightarrow{up} \overline{\theta'} \quad \textit{iff} \quad \exists v \in \overline{\theta},\ \exists d \in \mathcal{D},\ up(v, d) \in \overline{\theta'}$$

In order to capture decidability in our model, we define the following condition:

$$\textsc{Condition}\ (\dagger)\text{: } \overline{\theta} \xrightarrow{up} \overline{\theta'} \quad \textit{iff} \quad \forall v \in \overline{\theta},\ \exists d \in \mathcal{D},\ up(v, d) \in \overline{\theta'}$$

We will prove that this simple condition ensures the decidability of the emptiness problem. The principle of the proof of this result is very similar to the one of region construction as defined by Alur and Dill [AD94].

Theorem 2. *Let L be a recognizable data language over Σ and $\mathcal{D}$. Assume L is recognized by the finite monoid M with an equivalence and updates that satisfy the condition (†). Then the emptiness of L is decidable in complexity* PSPACE.

Proof (Sketch). Let $L \subseteq (\Sigma \times \mathcal{D})^*$ be a recognizable data language. We assume that M is a monoid which recognizes L and that k, $\sim$, φ and $up_{m,a}$ satisfy the condition (†) and are correctly defined in order to recognize L. As in the proof of Theorem 1, we construct a data automaton $\mathcal{A}$ whose transitions are

$$m \xrightarrow{g,\ a,\ up_{m,a},\ g'} m\varphi(a, g')$$

Of course, $L = L(\mathcal{A})$. From $\mathcal{A}$, we construct a finite automaton $\mathcal{B} = (Q, I, F, T)$ where $Q = M \times \mathcal{D}^k/\!\sim$, $I = (1_M, \overline{\perp^k})$, $F = P \times \mathcal{D}^k/\!\sim$ and T is defined by

$$((m, g), a, (m', g')) \in T \iff m \xrightarrow{g,\ a,\ up_{m,a},\ g'} m' \text{ and } g \xrightarrow{up_{m,a}} g'$$

We can prove that, as condition (†) holds, this finite automaton accepts

$$\textsc{Undata}(L) = \{a_1 \dots a_n \mid \exists d_1, \dots, d_n,\ (a_1, d_1) \dots (a_n, d_n) \in L\}$$

□

It remains to study the decidability of condition (†).
We first define

$$\begin{cases} \widehat{up}(\overline{\theta}) = \{v' \mid \exists v \in \overline{\theta},\ \exists d \in \mathcal{D},\ v' = up(v, d)\} \\ \widehat{up}^{-1}(\overline{\theta'}) = \{v \mid \exists d \in \mathcal{D},\ up(v, d) \in \overline{\theta'}\} \end{cases}$$

With these definitions, it is obvious that

$$\textsc{Condition (†)} \iff \left[\widehat{up}(\overline{\theta}) \cap \overline{\theta'} \neq \emptyset \implies \widehat{up}^{-1}(\overline{\theta'}) \cap \overline{\theta} = \overline{\theta}\right]$$

Since our updates do not compute anything, from a given equivalence class $\overline{\theta}$, the sets $\widehat{up}(\overline{\theta})$ and $\widehat{up}^{-1}(\overline{\theta})$ can be obtained "easily". Indeed, $\widehat{up}$ has just for effect to put in all the registers of up any value in $\mathcal{D}$ and $\widehat{up}^{-1}$ is just a projection on the registers which are not updated by up. Therefore, as soon as we are able to decide both the emptiness of $\widehat{up}(\overline{\theta}) \cap \overline{\theta'}$ and the equality $\widehat{up}^{-1}(\overline{\theta'}) \cap \overline{\theta} = \overline{\theta}$, the condition (†) becomes decidable.
Note that it is in particular the case when the equivalence $\sim$ is given by a set of linear inequations.

6 Extensions of the Model

6.1 Non-deterministic Models

Up to now, we only considered models that are deterministic, *i.e.* for each data word, there is a unique possible execution on it. Now, we will consider a non-deterministic version of the models. We thus define non-deterministic data automata as in Definition 2, but without the determinism condition. We also say

that a finite monoid M non-deterministically recognizes a data language L whenever there exists, k, $\sim$, φ as in Definition 1, but for each $(m, a) \in M \times \Sigma$, there exists a finite set of updates $U_{m,a}$ (instead of a unique update $up_{m,a}$) such that the conditions in Definition 1 hold. Some properties which are true for deterministic data automata are also true for non-deterministic data automata:

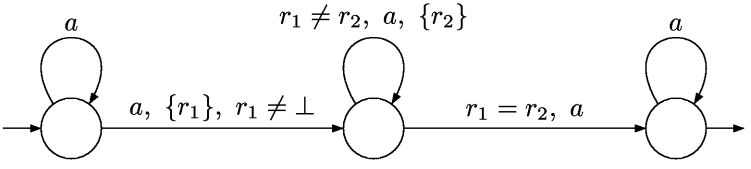
Proposition 4. *Let L be a data language over Σ and $\mathcal{D}$. Then,*

- *L is non-deterministically recognized by a finite monoid if and only if it is recognized by a non-deterministic data automaton.*

We say that L is nd-recognizable *whenever L is accepted by some non-deterministic data automaton.*

- *Condition (†) ensures the decidability of the emptiness problem, i.e. if L is recognized by a non-deterministic data automaton that satisfies the condition (†), then we can test for its emptiness.*
- *The class of nd-recognizable data languages is strictly more expressive than the class of recognizable data languages.*

Proof (Sketch for the last point). Consider the data language L accepted by the following non-deterministic data automaton:



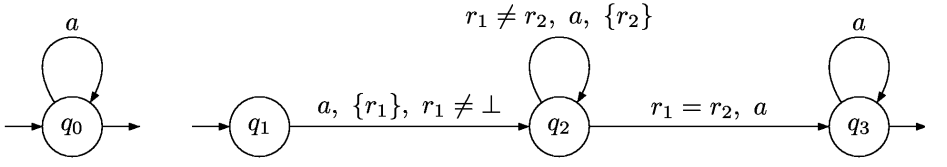
We have that $L = \{(a, d_1) \dots (a, d_n) \mid \exists 1 \leq i < j < n,\ d_i = d_j \neq \bot\}$. Moreover, one can prove that L is not recognized by any (deterministic) data automaton. □

Corollary 1. *The class of recognizable data languages is not closed under concatenation.*

Proof. Consider the previous data language L. Although it is not recognizable, this language is the concatenation of the two following recognizable data languages:

$$\{(a, d_1) \dots (a, d_p) \mid d_i \in \mathcal{D}\} \quad \text{and} \quad \{(a, d_0) \dots (a, d_n) \mid \exists 1 \leq j < n,\ d_j = d_0\}$$

which are recognized by the following data automata:



□

Proposition 5. *The class of nd-recognizable data languages is closed under union, intersection, concatenation and finite iteration. It is not closed under complementation.*

6.2 More General Updates

The updates used in the model are very simple, we can only "write a data in a memory", but we cannot perform any calculation. So, the question is: does all what precedes generalizes to models in which updates can perform calculations. In this section, an update is now a general function $up : \mathcal{D}^k \times \mathcal{D} \longrightarrow \mathcal{D}^k$.

Considering the simple updates of registers, we showed that the monoid played a very important role: "different" monoids do not recognize the same data languages. Extending the updates, the relevance of the monoid is lost.

Proposition 6. *Let L be a language over the finite alphabet Σ. Assume that L is recognized by a finite monoid M. Then the data language*

$$L_M = \{(a_1, m_1) \dots (a_n, m_n) \mid a_1 \dots a_n \in L\}$$

over Σ and M is recognized by the monoid $N = \{1, x, y\}$ with $zx = x$ and $zy = y$.

Proof. We assume that $L \subseteq \Sigma^*$ is recognized by M. There exists a morphism $\varphi : \Sigma^* \longrightarrow M$, a subset $P \subseteq M$ such that $L = \varphi^{-1}(P)$. Let us now define $k = 1$ (there is only one register) and $\mathcal{D} = M$. Then, for each $z \in N$, for each $a \in \Sigma$, we define $up_{z,a} : M \times M \longrightarrow M$ by $up_{z,a}(m, d) = m\varphi(a)$. We define also a morphism $\psi : (\Sigma \times M)^* \longrightarrow N$ by:

$$\psi(a, m) = \begin{cases} x \text{ if } m \in P \\ y \text{ if } m \in M \setminus P \end{cases}$$

Then, using this construction, we can prove that N recognizes the data language L_M. □

However, allowing more general updates like functions $\mathcal{D}^k \times \mathcal{D} \longrightarrow \mathcal{D}^k$, the results on equivalence between monoids and automata and on decidability are always true, because these results do not depend on the updates.

7 Comparison with Timed Automata

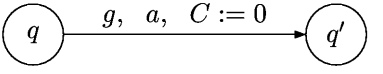
One of the main motivation of this work was to find an algebraic characterization of timed languages. It is clear that if we consider as data domain $\mathcal{D}$ a classical time domain, then our data languages reduce to timed languages (since we can easily handle the monotonicity condition on time).

Proposition 7. *Let $\mathcal{A}$ be a (deterministic) timed automata with n clocks. There exists a (deterministic) data automaton with $2n + 2$ registers which recognizes the same language.*

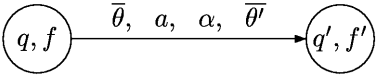
Proof (Sketch). We assume that the definition of a (deterministic) timed automaton is known, otherwise, we refer to [AD94].
Let us consider a deterministic timed automaton $\mathcal{A}$ with n clocks, $\{x_1, \ldots, x_n\}$. Without loss of generality, we can assume that there exists an equivalence on $\mathcal{D}^n$, namely $\equiv$, such that if g is a guard appearing in $\mathcal{A}$, then g is an equivalence class of $\equiv$. A clock x_0 is added to the set of clocks and represents the universal time, *i.e.* x_0 is never reset in $\mathcal{A}$. We construct a (deterministic) data automaton $\mathcal{B}$ with $2n+2$ registers in the following way:
The set of states of $\mathcal{B}$ is $Q \times \mathcal{F}$ where Q is the set of states of $\mathcal{A}$ and $\mathcal{F}$ is the set of functions $f : \{x_0, \ldots, x_n\} \longrightarrow \{0, \ldots, 2n+1\}$ such that for all $0 \leq i \leq n$, $f(x_i) \in \{i, n+1+i\}$. Intuitively, the value of the clock x_i will be alternatively kept by the two registers i and $n+1+i$.
The equivalence $\sim$ in $\mathcal{B}$ is defined by:

$$(\theta_i)_{0\leq i\leq 2n+1} \sim (\theta'_i)_{0\leq i\leq 2n+1}$$
$$\Updownarrow$$
$$\begin{cases} \forall f \in \mathcal{F},\ (\theta_{f(x_0)} - \theta_{f(x_i)})_{1\leq i\leq n} \equiv (\theta'_{f(x_0)} - \theta'_{f(x_i)})_{1\leq i\leq n}, \\ \left(\theta_0 < \theta_{n+1} \iff \theta'_0 < \theta'_{n+1}\right), \\ \left(\theta_0 > \theta_{n+1} \iff \theta'_0 > \theta'_{n+1}\right) \end{cases}$$

Consider a transition in $\mathcal{A}$:



For each function f in $\mathcal{F}$, we construct transitions in $\mathcal{B}$ in the following way:



where

- $\overline{\theta}$ is any equivalence class of $\sim$,
- $\alpha = \{0, 1, \ldots, 2n+1\} \setminus \{f(x_0), \ldots, f(x_n)\}$,
- $f' \in \mathcal{F}$ is such that

$$f'(x_0) = \begin{cases} 0 & \text{if } f(x_0) = n+1 \\ n+1 & \text{if } f(x_0) = 0 \end{cases}$$
$$f'(x_i) = \begin{cases} f(x_i) & \text{if } x_i \notin C \\ n+1+i & \text{if } x_i \in C \text{ and } f(x_i) = i \\ i & \text{if } x_i \in C \text{ and } f(x_i) = n+1+i \end{cases}$$

- $\overline{\theta'}$ is any equivalence class of $\sim$ such that

$$(\beta_i)_{0\leq i\leq 2n+1} \in \overline{\theta'} \implies (\beta_{f'(x_0)} - \beta_{f(x_i)})_{1\leq i\leq n} \in g \text{ and } \beta_{f'(x_0)} > \beta_{f(x_0)}$$

The timed automaton $\mathcal{B}$ that we just constructed is deterministic and recognizes the same timed language as $\mathcal{A}$. □

Hence any timed language accepted by some deterministic timed automaton (as defined by [AD94]) is also recognized by a data automaton with the timed domain as data domain.
Conversely, data automata allow to recognize a much larger class of languages. Indeed all the languages accepted by the extension of timed automata proposed in [CG00] are also recognized by data automata. And even, for example, the language $\{(a,\tau)(a,2\tau)\dots(a,n\tau) \mid \tau \in \mathbb{Q}_+\}$ is recognized by a data automaton whereas it is known that this language cannot be recognized by a timed automaton, even in the extension proposed by [DZ98].
We can also define more exotic languages which are monoid recognizable as for instance the set $\{(a,t_1)\dots(a,t_n) \mid \forall i,\ t_i \text{ is a prime number}\}$. Namely, it suffices to consider a monoid with 2 elements, 1 register and an equivalence relation of index 2. The first class contains all the prime numbers and the second class all the others. Note that the condition (†) holds.

8 Conclusion

We have proposed in this paper a notion of monoid recognizability for data languages. We also gave an automaton characterization of this notion. Hence, the picture for data languages is rather close to the one for classical formal languages. As an instance of our results, we can deal with timed languages.
This theory has now to be developed. For instance, a notion of aperiodic data language can naturally be defined and has to be studied.
In the timed framework, any timed language recognized by deterministic timed automata is monoid recognizable. But the exact relations with the numerous sets of timed languages that have been proposed in the literature, see for instance [HRS98], have to be investigated.
Another interesting direction will also consist in understanding the exact relation between the power of the monoid and the power of the updates. In this paper, we have investigated the two extreme cases. If updates on registers can only choose to store or to skip a data, then the structure of the monoid is crucial. On the contrary, if the updates can do heavy computations, then the monoid is nearly useless. All cases in between have still to be studied.

References

[ABB80] Autebert, J.-M., Beauquier, J., and Boasson, L. *Langages sur des alphabets infinis*. Discrete Applied Mathematics, vol. 2:1–20, 1980.

[ACM97] Asarin, E., Caspi, P., and Maler, O. *A Kleene Theorem for Timed Automata*. In *Proc. 12th IEEE Symp. Logic in Computer Science (LICS'97)*, pp. 160–171. IEEE Computer Society Press, June 1997.

[AD94] Alur, R. and Dill, D. *A Theory of Timed Automata*. Theoretical Computer Science, vol. 126(2):183–235, 1994.

[AFH94] Alur, R., Fix, L., and Henzinger, T. A. *Event-Clock Automata: a Determinizable Class of Timed Automata*. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94)*, vol. 818 of *Lecture Notes in Computer Science*, pp. 1–13. Springer-Verlag, June 1994.

[Asa98] Asarin, E. *Equations on Timed Languages*. In *Hybrid Systems: Computation and Control*, vol. 1386 of *Lecture Notes in Computer Science*. Springer-Verlag, Apr. 1998.

[BP99] Bouyer, P. and Petit, A. *Decomposition and Composition of Timed Automata*. In *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99)*, vol. 1644 of *Lecture Notes in Computer Science*, pp. 210–219. Springer-Verlag, July 1999.

[BP01] Bouyer, P. and Petit, A. *A Kleene/Büchi-like Theorem for Clock Languages*. Journal of Automata, Languages and Combinatorics, 2001. To appear.

[BPT01] Bouyer, P., Petit, A., and Thérien, D. *An Algebraic Characterization of Data and Timed Languages*. Research report LSV-01-1, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan, 2001.

[CG00] Choffrut, C. and Goldwurm, M. *Timed Automata with Periodic Clock Constraints*. Journal of Automata, Languages and Combinatorics, vol. 5(4):371–404, 2000.

[CGP99] Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[DZ98] Demichelis, F. and Zielonka, W. *Controlled Timed Automata*. In *Proc. 9th Int. Conf. Concurrency Theory (CONCUR'98)*, vol. 1466 of *Lecture Notes in Computer Science*, pp. 455–469. Springer-Verlag, Sep. 1998.

[HKWT95] Henzinger, T. A., Kopke, P. W., and Wong-Toi, H. *The Expressive Power of Clocks*. In *Proc. 22nd Int. Coll. Automata, Languages, and Programming (ICALP'95)*, vol. 944 of *Lecture Notes in Computer Science*, pp. 335–346. Springer-Verlag, July 1995.

[HRS98] Henzinger, T. A., Raskin, J.-F., and Schobbens, P.-Y. *The Regular Real-Time Languages*. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98)*, vol. 1443 of *Lecture Notes in Computer Science*, pp. 580–591. Springer-Verlag, July 1998.

[KF94] Kaminski, M. and Francez, N. *Finite-Memory Automata*. Theoretical Computer Science, vol. 134:329–363, 1994.

[Min67] Minsky, M. *Computation: Finite and Infinite Machines*. Prentice Hall Int., 1967.

[Pin86] Pin, J.-E. *Varieties of Formal Languages*. North Oxford, London et Plenum, New-York, 1986.

[RS97] Rozenberg, G. and Salomaa, A., eds. *Handbook of Formal Languages*. Springer-Verlag, 1997.

[Wil94] Wilke, T. *Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata*. In *Proc. 3rd Int. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, vol. 863 of *Lecture Notes in Computer Science*, pp. 694–715. Springer-Verlag, Sep. 1994.

A Faster–than Relation for Asynchronous Processes*

Gerald Lüttgen[1] and Walter Vogler[2]

[1] Department of Computer Science, Sheffield University, 211 Portobello Street, Sheffield S1 4DP, U.K., g.luettgen@dcs.shef.ac.uk
[2] Institut für Informatik, Universität Augsburg, D–86135 Augsburg, Germany, vogler@informatik.uni-augsburg.de

Abstract. This paper introduces a novel (bi)simulation–based faster–than preorder which relates asynchronous processes with respect to their worst–case timing behavior. The studies are conducted for a conservative extension of the process algebra CCS, called TACS, which permits the specification of maximal time bounds of actions. The most unusual contribution is in showing that the proposed faster–than preorder coincides with two other preorders, one of which considers the absolute times at which actions occur in system runs. The paper also develops the semantic theory of TACS, addressing congruence properties, equational laws, and abstractions from internal actions.

1 Introduction

Process algebras [5] provide a widely studied framework for reasoning about the behavior of concurrent systems. Early approaches, including Milner's CCS [15], focused on semantic issues of asynchronous processes, where the relative speeds between processes running in parallel is not bounded, i.e., one process may be arbitrarily slower or faster than another. This leads to a simple and mathematically elegant semantic theory analyzing the functional behavior of systems regarding their causal interactions with their environments. To include time as an aspect of system behavior, *timed process algebras* [4] were introduced. They usually model synchronous systems where processes running in parallel are under the regime of a common global clock and have a fixed speed. A well–known representative of discrete timed process algebras is Hennessy and Regan's TPL [11] which extends CCS by a timeout operator and a clock prefix demanding that exactly one time unit *must* pass before activating the argument process. Research papers on timed process algebras usually do not relate processes with respect to speed; the most notable exception is work by Moller and Tofts [17] which considers a faster–than preorder within a CCS–based setting, where processes are attached with lower time bounds. In practice, however, often upper time bounds are known to a system designer, determining how long a process may delay its execution. These can

* Research support was partly provided under NASA Contract No. NAS1–97046.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 262–276, 2001.
© Springer-Verlag Berlin Heidelberg 2001

be used to compare the *worst–case timing behavior* of processes. The assumption of upper time bounds for *asynchronous* processes is exploited in distributed algorithms and was studied by the second author [6,13,12,20,21,22] in settings equipped with DeNicola and Hennessy's testing semantics [9]. We re–emphasize that, in our context, "asynchronous" means that the relative speeds of system components are indeterminate.

In this paper we develop a novel (bi)simulation–based approach to compare asynchronous systems with respect to their worst–case timing behavior. To do so, we extend CCS by a rather specific notion of clock prefixing "$\sigma.$", where σ stands for one time unit or a single clock tick. In contrast to TPL we interpret $\sigma.P$ as a process which *may* delay *at most* one time unit before executing P. Similar to TPL we view the occurrence of actions as instantaneous. This results in a new process algebra extending CCS, to which we refer as *Timed Asynchronous Communicating Systems* (TACS). To make our intuition of upper bound delays precise, consider the processes $\sigma.a.\mathbf{0}$ and $a.\mathbf{0}$, where a denotes an action as in CCS. While the former process may delay an *enabled* communication on a by one time unit, the latter must engage in the communication, i.e., a is *non–urgent* in $\sigma.a.\mathbf{0}$ but *urgent* in $a.\mathbf{0}$. However, if a communication on a is not enabled, then process $a.\mathbf{0}$ may wait until some communication partner is ready. To enforce a communication resulting in the internal action τ, a time step in TACS is preempted by an urgent τ. This is similar to timed process algebras employing the *maximal progress assumption* [11] where, however, in contrast to TACS, any internal computation is considered to be urgent. For TACS we introduce a *faster–than preorder* which exploits upper time bounds: a process is faster than another if both are linked by a relation which is a strong bisimulation for actions and a simulation for time steps.

The main contribution of this paper is the formal underpinning of our preorder, justifying why it is a good candidate for a faster–than relation on processes. There are at least two very appealing alternative definitions for such a preorder. First, one could allow the slower process to perform extra time steps when simulating an action or time step of the faster process. Second is the question of how exactly the faster process can match a time step and the subsequent behavior of the slower one. For illustrating this issue, consider the runs $a\sigma\sigma b$ and $\sigma a\sigma b$ which might be exhibited by some processes. One can argue that the first run is faster than the second one since action a occurs earlier in the run and since action b occurs at absolute time two in both runs, measured from the start of each run. Accordingly, we define a second variant of our faster–than preorder, where a time step of the slower process is either simulated immediately by the faster one or might be performed later on. As a key result we prove that both variants coincide with our faster–than preorder. Subsequently, this paper develops the preorder's semantic theory: we characterize the coarsest precongruence contained in it, demonstrate that TACS with this precongruence is a conservative extension of CCS with bisimulation, and axiomatize our precongruence for finite sequential processes. We also study the corresponding weak faster–than

preorder which abstracts from internal computation. All proofs can be found in a technical report [14].

2 Timed Asynchronous Communicating Systems

The novel process algebra TACS conservatively extends CCS [15] by a concept of global, discrete time. This concept is introduced to CCS by including the clock prefixing operator "$\sigma.$" [11] with a non–standard interpretation: a process $\sigma.P$ *can at most* delay one time unit before having to execute process P, provided that P can engage in a communication with the environment or in some internal computation. The semantics of TACS is based on a notion of transition system that involves two kinds of transitions, *action transitions* and *clock transitions*. Action transitions, like in CCS, are local handshake communications in which two processes may synchronize to take a joint state change together. A clock represents the progress of time which manifests itself in a recurrent global synchronization event, the clock transition. As indicated before, action and clock transitions are not orthogonal concepts, since time can only pass if the process under consideration cannot engage in an urgent internal computation.

Syntax. Let Λ be a countable set of actions not including the distinguished unobservable, *internal* action τ. With every $a \in \Lambda$ we associate a *complementary action* $\overline{a}$. We define $\overline{\Lambda} =_{\mathrm{df}} \{\overline{a} \mid a \in \Lambda\}$ and take $\mathcal{A}$ to denote the set $\Lambda \cup \overline{\Lambda} \cup \{\tau\}$. Complementation is lifted to $\Lambda \cup \overline{\Lambda}$ by defining $\overline{\overline{a}} =_{\mathrm{df}} a$. As in CCS [15], an action a communicates with its complement $\overline{a}$ to produce the internal action τ. We let $a, b, \ldots$ range over $\Lambda \cup \overline{\Lambda}$ and $\alpha, \beta, \ldots$ over $\mathcal{A}$ and represent (potential) clock ticks by the symbol σ. The syntax of TACS is then defined as follows:

$$P \ ::= \ \mathbf{0} \mid x \mid \alpha.P \mid \sigma.P \mid P + P \mid P|P \mid P \setminus L \mid P[f] \mid \mu x.P$$

where x is a *variable* taken from a countably infinite set $\mathcal{V}$ of variables, $L \subseteq \mathcal{A} \setminus \{\tau\}$ is a *restriction set*, and $f : \mathcal{A} \to \mathcal{A}$ is a *finite relabeling*. A finite relabeling satisfies the properties $f(\tau) = \tau$, $f(\overline{a}) = \overline{f(a)}$, and $|\{\alpha \mid f(\alpha) \neq \alpha\}| < \infty$. The set of all terms is abbreviated by $\widehat{\mathcal{P}}$, and we define $\overline{L} =_{\mathrm{df}} \{\overline{a} \mid a \in L\}$. Moreover, we use the standard definitions for the semantic *sort* $\mathsf{sort}(P) \subseteq \Lambda \cup \overline{\Lambda}$ of some term P, *open* and *closed* terms, and *contexts* (terms with a "hole"). A variable is called *guarded* in a term if each occurrence of the variable is within the scope of an action prefix. Moreover, we require for terms of the form $\mu x.P$ that x is guarded in P. We refer to closed and guarded terms as *processes*, with the set of all processes written as $\mathcal{P}$.

Semantics. The *operational semantics* of a TACS term $P \in \widehat{\mathcal{P}}$ is given by a labeled transition system $\langle \widehat{\mathcal{P}}, \mathcal{A} \cup \{\sigma\}, \longrightarrow, P \rangle$, where $\widehat{\mathcal{P}}$ is the set of states, $\mathcal{A} \cup \{\sigma\}$ the alphabet, $\longrightarrow \subseteq \widehat{\mathcal{P}} \times \mathcal{A} \cup \{\sigma\} \times \widehat{\mathcal{P}}$ the transition relation, and P the start state. Before we proceed, it is convenient to introduce sets $\mathcal{U}(P)$, for all terms $P \in \widehat{\mathcal{P}}$, which include the *urgent actions* in which P can initially engage, as discussed in the introduction. These sets are inductively defined along the

structure of P, as shown in Table 1. Strictly speaking, $\mathcal{U}(P)$ does not necessarily contain *all* urgent actions. For example, for $P = \tau.0 + \sigma.a.\mathbf{0}$ we have $\mathcal{U}(P) = \{\tau\}$, although action a is semantically also urgent, because the clock transition of P is preempted according to our notion of maximal progress. However, in the sequel we need the urgent action set of P only for determining whether P can initially perform an urgent τ. For this purpose, our syntactic definition of urgent action sets suffices since $\tau \in \mathcal{U}(P)$ *if and only if* τ *is semantically urgent in* P.

Table 1. Urgent action sets

$\mathcal{U}(\sigma.P)$	$=_{\mathrm{df}} \emptyset$	$\mathcal{U}(\mathbf{0}) = \mathcal{U}(x)$	$=_{\mathrm{df}} \emptyset$	$\mathcal{U}(P \setminus L)$	$=_{\mathrm{df}} \mathcal{U}(P) \setminus (L \cup \overline{L})$
$\mathcal{U}(\alpha.P)$	$=_{\mathrm{df}} \{\alpha\}$	$\mathcal{U}(P + Q)$	$=_{\mathrm{df}} \mathcal{U}(P) \cup \mathcal{U}(Q)$	$\mathcal{U}(P[f])$	$=_{\mathrm{df}} \{f(\alpha) \mid \alpha \in \mathcal{U}(P)\}$
$\mathcal{U}(\mu x.P)$	$=_{\mathrm{df}} \mathcal{U}(P)$	$\mathcal{U}(P \mid Q)$	$=_{\mathrm{df}} \mathcal{U}(P) \cup \mathcal{U}(Q) \cup \{\tau \mid \mathcal{U}(P) \cap \overline{\mathcal{U}(Q)} \neq \emptyset\}$		

Now, the operational semantics for action transitions and clock transitions can be defined via the *structural operational rules* displayed in Tables 2 and 3, respectively. For action transitions, the rules are exactly the same as for CCS, with the exception of our new clock–prefix operator. For clock transitions, our semantics is set up such that, if $\tau \in \mathcal{U}(P)$, then a clock tick σ of P is inhibited. For the sake of simplicity, let us write $P \xrightarrow{\gamma} P'$ instead of $\langle P, \gamma, P' \rangle \in \longrightarrow$, for $\gamma \in \mathcal{A} \cup \{\sigma\}$, and say that P *may engage in* γ *and thereafter behave like* P'. Sometimes it is also convenient to write $P \xrightarrow{\gamma}$ for $\exists P'.\, P \xrightarrow{\gamma} P'$.

Table 2. Operational semantics for TACS (action transitions)

Act	$\dfrac{-}{\alpha.P \xrightarrow{\alpha} P}$	Pre	$\dfrac{P \xrightarrow{\alpha} P'}{\sigma.P \xrightarrow{\alpha} P'}$	Rec	$\dfrac{P \xrightarrow{\alpha} P'}{\mu x.P \xrightarrow{\alpha} P'[\mu x.P/x]}$
Sum1	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$	Sum2	$\dfrac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$		
Com1	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	Com2	$\dfrac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$	Com3	$\dfrac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
Rel	$\dfrac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$	Res	$\dfrac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha \notin L \cup \overline{L}$		

The *action–prefix* term $\alpha.P$ may engage in action α and then behave like P. If $\alpha \neq \tau$, then it may also *idle*, i.e., engage in a clock transition to itself, as process $\mathbf{0}$ does. The *clock–prefix* term $\sigma.P$ can engage in a clock transition to P and, additionally, it can perform any action transition that P can, since σ represents a delay of *at most* one time unit. The *summation operator* $+$ denotes nondeterministic choice such that $P + Q$ may behave like P or Q. Time has to

proceed equally on both sides of summation, whence $P+Q$ can engage in a clock transition and delay the nondeterministic choice if and only if both P and Q can. Consequently, e.g., process $\sigma.a.\mathbf{0} + \tau.\mathbf{0}$ cannot engage in a clock transition; in particular, a has to occur without delay if it occurs at all. The *restriction operator* $\backslash L$ prohibits the execution of actions in $L \cup \overline{L}$ and, thus, permits the scoping of actions. $P[f]$ behaves exactly as P where actions are renamed by the *relabeling* f. The term $P|Q$ stands for the *parallel composition* of P and Q according to an interleaving semantics with synchronized communication on complementary actions, resulting in the internal action τ. Again, time has to proceed equally on both sides of the operator. The side condition ensures that $P|Q$ can only progress on σ, if it cannot engage in an urgent τ. Finally, $\mu x.\, P$ denotes *recursion*, i.e., $\mu x.\, P$ behaves as a distinguished solution to the equation $x = P$.

Table 3. Operational semantics for TACS (clock transitions)

tNil	$\dfrac{-}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}}$	tRec	$\dfrac{P \xrightarrow{\sigma} P'}{\mu x.P \xrightarrow{\sigma} P'[\mu x.P/x]}$	tRes	$\dfrac{P \xrightarrow{\sigma} P'}{P \setminus L \xrightarrow{\sigma} P' \setminus L}$
tAct	$\dfrac{-}{a.P \xrightarrow{\sigma} a.P}$	tSum	$\dfrac{P \xrightarrow{\sigma} P' \quad Q \xrightarrow{\sigma} Q'}{P+Q \xrightarrow{\sigma} P'+Q'}$	tRel	$\dfrac{P \xrightarrow{\sigma} P'}{P[f] \xrightarrow{\sigma} P'[f]}$
tPre	$\dfrac{-}{\sigma.P \xrightarrow{\sigma} P}$	tCom	$\dfrac{P \xrightarrow{\sigma} P' \quad Q \xrightarrow{\sigma} Q'}{P\|Q \xrightarrow{\sigma} P'\|Q'}\ \tau \notin \mathcal{U}(P\|Q)$		

The operational semantics for TACS possesses several important properties [11]. First, it is *time–deterministic*, i.e., processes react deterministically to clock ticks, reflecting the intuition that progress of time does not resolve choices. Formally, $P \xrightarrow{\sigma} P'$ and $P \xrightarrow{\sigma} P''$ implies $P' = P''$, for all $P, P', P'' \in \widehat{\mathcal{P}}$. Second, according to our variant of *maximal progress*, $P \xrightarrow{\sigma}$ if and only if $\tau \notin \mathcal{U}(P)$, for all $P \in \widehat{\mathcal{P}}$.

3 Design Choices for Faster–than Relations

In the following we define a reference faster–than relation, called *naive faster–than preorder*, which is inspired by Milner's notions of *simulation* and *bisimulation* [15]. Our main objective is to convince the reader that this simple faster–than preorder with its concise definition is not chosen arbitrarily. This is done by showing that it coincides with two other preorders which formalize a notion of faster–than as well and which are possibly more intuitive.

Definition 1 (Naive faster–than preorder). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *naive faster–than relation* if, for all $\langle P, Q \rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \xrightarrow{\alpha} P'$ implies $\exists Q'.\, Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$.

2. $Q \xrightarrow{\alpha} Q'$ implies $\exists P'.\, P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$.
3. $P \xrightarrow{\sigma} P'$ implies $\exists Q'.\, Q \xrightarrow{\sigma} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$.

We write $P \sqsupseteq_{\mathrm{n}} Q$ if $\langle P, Q \rangle \in \mathcal{R}$ for some naive faster–than relation $\mathcal{R}$.

Note that the behavioral relation $\sqsupseteq_{\mathrm{n}}$, as well as all other behavioral relations on processes defined in the sequel, can be extended to open terms by the usual means of closed substitution [15]. It is fairly easy to see that $\sqsupseteq_{\mathrm{n}}$ is a preorder, i.e., it is transitive and reflexive; moreover, $\sqsupseteq_{\mathrm{n}}$ is the largest naive faster–than relation. Intuitively, $P \sqsupseteq_{\mathrm{n}} Q$ holds if P is faster than (or as fast as) Q, and if both processes are functionally equivalent (cf. Clauses (1) and (2)). Here, "*P is faster than Q*" means the following: if P may let time pass and the environment of P has to wait, then this should also be the case if one considers the slower (or equally fast) process Q instead (cf. Clause (3)). However, if Q lets time pass, then P is not required to match this behavior. Observe that we use bounded delays and, accordingly, are interested in worst–case behavior. Hence, clock transitions of the fast process must be matched, but not those of the slow process; behavior after an unmatched clock transition can just as well occur quickly without the time step, whence it is catered for in Clause (2).

As the naive faster–than preorder is the basis of our approach, it is very important that its definition is intuitively convincing. There are two immediate questions which arise from our definition.

Question I. The first question concerns the observation that Clauses (1) and (3) of Def. 1 require that an action or a time step of P must be matched with just this action or time step by Q. What if we are less strict? Maybe we should allow the slower process Q to perform some additional time steps when matching the behavior of P. This idea is formalized in the following variant of our faster–than preorder. Here, $\xrightarrow{\sigma}{}^{+}$ and $\xrightarrow{\sigma}{}^{*}$ stand for the transitive and the transitive reflexive closure of the clock transition relation $\xrightarrow{\sigma}$, respectively.

Definition 2 (Delayed faster–than preorder). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *delayed faster–than relation* if, for all $\langle P, Q \rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \xrightarrow{\alpha} P'$ implies $\exists Q'.\, Q \xrightarrow{\sigma}{}^{*} \xrightarrow{\alpha} \xrightarrow{\sigma}{}^{*} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$.
2. $Q \xrightarrow{\alpha} Q'$ implies $\exists P'.\, P \xrightarrow{\alpha} P'$ and $\langle P', Q' \rangle \in \mathcal{R}$.
3. $P \xrightarrow{\sigma} P'$ implies $\exists Q'.\, Q \xrightarrow{\sigma}{}^{+} Q'$ and $\langle P', Q' \rangle \in \mathcal{R}$.

We write $P \sqsupseteq_{\mathrm{d}} Q$ if $\langle P, Q \rangle \in \mathcal{R}$ for some delayed faster–than relation $\mathcal{R}$.

As usual one can derive that $\sqsupseteq_{\mathrm{d}}$ is a preorder and that it is the largest delayed faster–than relation. In the following we will show that both preorders $\sqsupseteq_{\mathrm{n}}$ and $\sqsupseteq_{\mathrm{d}}$ coincide; the proof of this result is based on a syntactic relation $\succ$ on terms.

Definition 3. The relation $\succ \subseteq \widehat{\mathcal{P}} \times \widehat{\mathcal{P}}$ is defined as the smallest relation satisfying the following properties, for all $P, P', Q, Q' \in \widehat{\mathcal{P}}$.

Always:	(1) $P \succ P$	(2) $P \succ \sigma.P$
If $P' \succ P$ and $Q' \succ Q$:	(3) $P'\|Q' \succ P\|Q$	(4) $P' + Q' \succ P + Q$
	(5) $P' \setminus L \succ P \setminus L$	(6) $P'[f] \succ P[f]$
If $P' \succ P$ and x guarded in P:	(7) $P'[\mu x.\, P'/x] \succ \mu x.\, P$	

Note that relation $\succ$ is not transitive and that it is also defined for open terms. Its essential properties are: (a) $P \xrightarrow{\sigma} P'$ implies $P' \succ P$, for any terms $P, P' \in \widehat{\mathcal{P}}$, and (b) $\succ$ satisfies the clauses of Def. 1, also on open terms; hence, $\succ_{|\mathcal{P}\times\mathcal{P}} \subseteq \sqsupseteq_{\mathrm{n}}$. Crucial for this are Clauses (2) and (7) of the above definition. For (a) we clearly must include Clause (2). Additionally, Clause (7) covers the unwinding of recursion; for its motivation consider, e.g., the transition $\mu x.\, \sigma.a.\sigma.b.x \xrightarrow{\sigma} a.\sigma.b.\mu x.\, \sigma.a.\sigma.b.x$.

Theorem 4 (Coincidence I). *The preorders $\sqsupseteq_n$ and $\sqsupseteq_d$ coincide.*

Question II. We now turn to a second question which might be raised regarding the definition of the naive faster–than preorder $\sqsupseteq_{\mathrm{n}}$. Should one add a fourth clause to the definition of $\sqsupseteq_{\mathrm{n}}$ that permits, but not requires, the faster process P to match a clock transition of the slower process Q? More precisely, P might be able to do whatever Q can do after a time step, or P might itself have to perform a time step to match Q. Hence, a candidate for a fourth clause is

$$(4)\quad Q \xrightarrow{\sigma} Q' \text{ implies } \langle P, Q'\rangle \in \mathcal{R} \text{ or } \exists P'.\, P \xrightarrow{\sigma} P' \text{ and } \langle P', Q'\rangle \in \mathcal{R}\,.$$

Unfortunately, this requirement is not as sensible as it might appear at first sight. Consider the processes $P =_{\mathrm{df}} \sigma^n.a.\mathbf{0} \,|\, a.\mathbf{0} \,|\, \overline{a}.\mathbf{0}$ and $Q =_{\mathrm{df}} \sigma^n.a.\mathbf{0} \,|\, \sigma^n.a.\mathbf{0} \,|\, \overline{a}.\mathbf{0}$, for $n \geq 1$. Obviously, we expect P to be faster than Q. However, Q can engage in a clock transition to $Q' =_{\mathrm{df}} \sigma^{n-1}.a.\mathbf{0} \,|\, \sigma^{n-1}.a.\mathbf{0} \,|\, \overline{a}.\mathbf{0}$. According to Clause (4) and since $P \not\xrightarrow{\sigma}$, we would require P to be faster than Q'. This conclusion, however, should obviously be deemed wrong according to our intuition of "faster than."

The point of this example is that process P, which is in some components faster than Q, cannot mimic a clock transition of Q with a matching clock transition. However, since P is equally fast in the other components, it cannot simply leave out the time step. The solution to this situation is to remember within the relation $\mathcal{R}$ how many clock transitions P missed out and, in addition, to allow P to perform these clock transitions later. Thus, the computation $Q \xrightarrow{\sigma}{}^{n} a.\mathbf{0} \,|\, a.\mathbf{0} \,|\, \overline{a}.\mathbf{0} \xrightarrow{a} \mathbf{0} \,|\, a.\mathbf{0} \,|\, \overline{a}.\mathbf{0} \xrightarrow{a} \mathbf{0} \,|\, \mathbf{0} \,|\, \overline{a}.\mathbf{0}$ of Q, where we have no clock transitions between the two action transitions labeled by a, can be matched by P with the computation $P \xrightarrow{a} \sigma^n.a.\mathbf{0} \,|\, \mathbf{0} \,|\, \overline{a}.\mathbf{0} \xrightarrow{\sigma}{}^{n} a.\mathbf{0} \,|\, \mathbf{0} \,|\, \overline{a}.\mathbf{0} \xrightarrow{a} \mathbf{0} \,|\, \mathbf{0} \,|\, \overline{a}.\mathbf{0}$. This matching is intuitively correct, since the first a occurs faster in the considered trace of P than in the trace of Q, while the second a occurs at the same absolute time, measured from the start of each computation.

Definition 5 (Family of faster–than preorders). A family $(\mathcal{R}_i)_{i\in\mathbb{N}}$ of relations over $\mathcal{P}$, indexed by natural numbers (including 0), is a *family of indexed–faster–than relations* if, for all $i \in \mathbb{N}$, $\langle P, Q\rangle \in \mathcal{R}_i$, and $\alpha \in \mathcal{A}$:

1. $P \xrightarrow{\alpha} P'$ implies $\exists Q'.\, Q \xrightarrow{\alpha} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}_i$.
2. $Q \xrightarrow{\alpha} Q'$ implies $\exists P'.\, P \xrightarrow{\alpha} P'$ and $\langle P', Q'\rangle \in \mathcal{R}_i$.
3. $P \xrightarrow{\sigma} P'$ implies (a) $\exists Q'.\, Q \xrightarrow{\sigma} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}_i$, or
   (b) $i > 0$ and $\langle P', Q\rangle \in \mathcal{R}_{i-1}$.

4. $Q \xrightarrow{\sigma} Q'$ implies (a) $\exists P'.\, P \xrightarrow{\sigma} P'$ and $\langle P', Q' \rangle \in \mathcal{R}_i$, or
   (b) $\langle P, Q' \rangle \in \mathcal{R}_{i+1}$.

We write $P \sqsupseteq_{\approx_i} Q$ if $\langle P, Q \rangle \in \mathcal{R}_i$ for some family of indexed–faster–than relations $(\mathcal{R}_i)_{i \in \mathbb{N}}$.

Intuitively, $P \sqsupseteq_{\approx_i} Q$ means that process P is faster than process Q provided that P may delay up to i additional clock ticks which Q does not need to match. Observe that there exists a family of largest indexed–faster–than relations, but it is not clear that these relations are transitive. We establish, however, a stronger result by showing that our naive faster–than preorder $\sqsupseteq_{\approx_{\mathrm{n}}}$ coincides with $\sqsupseteq_{\approx_0}$. The proof of this result uses a family of purely syntactic relations $\succ_i$, for $i \in \mathbb{N}$.

Definition 6. The relations $\succ_i \subseteq \widehat{\mathcal{P}} \times \widehat{\mathcal{P}}$, for $i \in \mathbb{N}$, are defined as the smallest relations such that, for all $P, P', Q, Q', P_1, \ldots, P_n \in \widehat{\mathcal{P}}$ and $i, j \in \mathbb{N}$:

Always:	(1)	$P \succ_i P$	
If $P_1 \succ P_2 \succ \cdots \succ P_n$:	(2a)	$P_1 \succ_i \sigma^j.P_n$	
If $P' \succ_i P$ and $Q' \succ_i Q$:	(2b)	$\sigma.P' \succ_{i+1} P$	
	(3)	$P'\|Q' \succ_i P\|Q$	(4) $P' + Q' \succ_i P + Q$
	(5)	$P' \setminus L \succ_i P \setminus L$	(6) $P'[f] \succ_i P[f]$
If $P' \succ_i P$, x guarded in P:	(7a)	$P'[\mu x.\, P/x] \succ_i \mu x.\, P$	
If $P' \succ_i P$, x guarded in P':	(7b)	$\mu x.\, P' \succ_i P[\mu x.\, P'/x]$	

Our syntactic relations satisfy the following useful properties:

1. $\succ_i \subseteq \succ_{i+1}$, for all $i \in \mathbb{N}$.
2. $\succ \subseteq \succ_0$; in particular, $P \xrightarrow{\sigma} P'$ implies $P' \succ_0 P$, for any $P, P' \in \widehat{\mathcal{P}}$.
3. $P' \succ P$ (whence, $P \xrightarrow{\sigma} P'$) implies $P \succ_i P'$, for all $i > 0$ and any $P, P' \in \widehat{\mathcal{P}}$.

For the proof of the following theorem, a series of further lemmas is needed, which show in particular that the family of relations $\succ_i$ satisfies the conditions of an indexed–faster–than family.

Theorem 7 (Coincidence II). *The preorders $\sqsupseteq_{\approx_n}$ and $\sqsupseteq_{\approx_0}$ coincide.*

4 Semantic Theory of Our Faster–than Relation

A shortcoming of the naive faster–than preorder $\sqsupseteq_{\approx_{\mathrm{n}}}$, as introduced above, is that it is not compositional. As an example, consider the processes $P =_{\mathrm{df}} \sigma.a.\mathbf{0}$ and $Q =_{\mathrm{df}} a.\mathbf{0}$, for which $P \sqsupseteq_{\approx_{\mathrm{n}}} Q$ holds according to Def. 1. Intuitively, however, this should not be the case, as we expect $P = \sigma.Q$ to be strictly slower than Q. Technically, if we compose P and Q in parallel with process $R =_{\mathrm{df}} \overline{a}.\mathbf{0}$, then $P|R \xrightarrow{\sigma} a.\mathbf{0}|\overline{a}.\mathbf{0}$, but $Q|R \xrightarrow{\sigma}\!\!\!\!\!/\;\;$, since any clock transition of $Q|R$ is preempted due to $\tau \in \mathcal{U}(Q|R)$. Hence, $P|R \not\sqsupseteq_{\approx_{\mathrm{n}}} Q|R$, i.e., $\sqsupseteq_{\approx_{\mathrm{n}}}$ is *not* a precongruence.

The reason for P and Q being equally fast according to $\sqsupseteq_{\approx_{\mathrm{n}}}$ lies in our operational rules: we allow Q to delay arbitrarily since this might be necessary in a context where no communication on a is possible. As R shows, we have to take a refined view once we fix a context. In order to find the largest precongruence contained in $\sqsupseteq_{\approx_{\mathrm{n}}}$ we must take the urgent action sets of processes into account.

Definition 8 (Strong faster–than precongruence). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *strong faster–than relation* if, for all $\langle P, Q\rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \xrightarrow{\alpha} P'$ implies $\exists Q'.\, Q \xrightarrow{\alpha} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
2. $Q \xrightarrow{\alpha} Q'$ implies $\exists P'.\, P \xrightarrow{\alpha} P'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
3. $P \xrightarrow{\sigma} P'$ implies $\mathcal{U}(Q) \subseteq \mathcal{U}(P)$ and $\exists Q'.\, Q \xrightarrow{\sigma} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.

We write $P \sqsupseteq\!\!\!\approx Q$ if $\langle P, Q\rangle \in \mathcal{R}$ for some strong faster–than relation $\mathcal{R}$.

Again, it is easy to see that $\sqsupseteq\!\!\!\approx$ is a preorder, that it is contained in $\sqsupseteq\!\!\!\approx_{\mathrm{n}}$, and that $\sqsupseteq\!\!\!\approx$ is the largest strong faster–than relation. We also have that P is strictly faster than $\sigma.P$, for all $P \in \mathcal{P}$, which is to be expected intuitively.

Theorem 9 (Full abstraction). *The preorder $\sqsupseteq\!\!\!\approx$ is the* largest precongruence *contained in $\sqsupseteq\!\!\!\approx_n$.*

We conclude this section by showing that TACS is a conservative extension of CCS. As noted earlier, we can interpret any process not containing a σ–prefix as CCS process. Moreover, for all TACS processes, we can adopt the equivalence *strong bisimulation* [15], in signs $\sim$, which is defined just as $\sqsupseteq\!\!\!\approx$ when omitting the third clause of Def. 8. Additionally, we denote the process obtained from some process $P \in \mathcal{P}$ when deleting all σ's by σ*–strip*(P).

Theorem 10 (Conservativity). *Let P, $Q \in \mathcal{P}$.*

1. *Always $P \sqsupseteq\!\!\!\approx Q$ implies $P \sim Q$.*
2. *If P and Q do not contain any σ–prefixes, then $P \sqsupseteq\!\!\!\approx Q$ if and only if $Q \sqsupseteq\!\!\!\approx P$ if and only if $P \sim Q$.*
3. *Always $P \sim \sigma$–strip(P); furthermore, $P \xrightarrow{\sigma} P'$ implies $P \sim P'$.*

This shows that our strong faster–than preorder refines strong bisimulation. Moreover, if no bounded delays occur in some processes, then these processes run in zero–time, and our strong faster–than preorder coincides with strong bisimulation. That the bounded delays in TACS processes do not influence any "functional" behavior, is demonstrated in the third part of the above result.

Axiomatization. Next, we provide a sound and complete axiomatization of our strong faster–than precongruence $\sqsupseteq\!\!\!\approx$ for the class of finite sequential processes. According to standard terminology, a process is called *finite sequential* if it does neither contain any recursion operator nor any parallel operator. Although this class seems to be rather restrictive at first sight, it is simple and rich enough to demonstrate, by studying axioms, how exactly our semantic theory for $\sqsupseteq\!\!\!\approx$ in TACS differs from the one for strong bisimulation in CCS [15].

The axioms for our strong faster–than precongruence are shown in Table 4, where any axiom of the form $t = u$ should be read as two axioms $t \sqsupseteq u$ and $u \sqsupseteq t$. We write $\vdash t \sqsupseteq u$ if $t \sqsupseteq u$ can be derived from the axioms. Axioms (A1)–(A4), (D1)–(D4), and (C1)–(C5) are exactly the ones for strong bisimulation in CCS [15]. Hence, the semantic theory of our calculus is distinguished from the

one for strong bisimulation by the additional Axioms (P1)–(P5). Intuitively, Axiom (P1) reflects our notion of maximal progress or urgency, namely that a process, which can engage in an internal urgent action, cannot delay. Axiom (P2) states that, if an action occurs "urgent" and "non–urgent" in a term, then it is indeed urgent, i.e., the non–urgent occurrence of the action may be transformed into an urgent one. Axiom (P3) is similar in spirit, but cannot be derived from Axiom (P2) and the other axioms. Axiom (P4) is a standard axiom in timed process algebras and testifies to the fact that time is a deterministic concept which does not resolve choices. Finally, Axiom (P5) encodes our elementary intuition of σ–prefixes and speed within TACS, namely that any process t is faster than process $\sigma.t$ which might delay the execution of t by one clock tick.

Table 4. Axiomatization for finite sequential processes

(A1)	$t + u = u + t$		(D1)	$\mathbf{0}[f] = \mathbf{0}$		
(A2)	$t + (u + v) = (t + u) + v$		(D2)	$(\alpha.t)[f] = f(\alpha).(t[f])$		
(A3)	$t + t = t$		(D3)	$(\sigma.t)[f] = \sigma.(t[f])$		
(A4)	$t + \mathbf{0} = t$		(D4)	$(t + u)[f] = t[f] + u[f]$		
(P1)	$\sigma.t + \tau.u = t + \tau.u$		(C1)	$\mathbf{0} \setminus L = \mathbf{0}$		
(P2)	$a.t + \sigma.a.u = a.t + a.u$		(C2)	$(\alpha.t) \setminus L = \mathbf{0}$	$\alpha \in L \cup \overline{L}$	
(P3)	$t + \sigma.t = t$		(C3)	$(\alpha.t) \setminus L = \alpha.(t \setminus L)$	$\alpha \notin L \cup \overline{L}$	
(P4)	$\sigma.(t + u) = \sigma.t + \sigma.u$		(C4)	$(\sigma.t) \setminus L = \sigma.(t \setminus L)$		
(P5)	$t \sqsupseteq \sigma.t$		(C5)	$(t + u) \setminus L = (t \setminus L) + (u \setminus L)$		

The correctness of our axioms relative to $\sqsupseteq_{\approx}$ can be established as usual [15]; note that all axioms are sound for arbitrary processes, not only for finite sequential ones. To prove the completeness of our axiomatization for finite sequential processes, we use a fairly involved notion of normal form; see [14] for details.

Theorem 11 (Correctness & completeness). *For finite sequential processes t and u we have: $\vdash t \sqsupseteq u$ if and only if $t \sqsupseteq_{\approx} u$.*

How to extend our axiomatization to cover parallel composition, too, is non–trivial and still an open problem. The difficulty lies in the lack of a suitable expansion law: observe that $\sigma.a.\mathbf{0} \,|\, \sigma.b.\mathbf{0}$ is strictly faster than $\sigma.a.\sigma.b.\mathbf{0} + \sigma.b.\sigma.a.\mathbf{0}$. However, since σ is synchronized, a more sensible expansion law would try to equate $\sigma.a.\mathbf{0} \,|\, \sigma.b.\mathbf{0}$ with $\sigma.(a.\mathbf{0} \,|\, b.\mathbf{0})$. But this law does not hold, since the latter process can engage in an a–transition to $\mathbf{0} \,|\, b.\mathbf{0}$ and is therefore strictly faster. Thus, our situation is the same as in Moller and Tofts' paper [17] which also considers a bisimulation–type faster–than relation for asynchronous processes, but which deals with best–case rather than worst–case timing behavior. It turns out that the axioms for the sequential sub–calculus given in [17] are all true in our setting; however, we have the additional Axioms (P1) and (P2) which both are valid since σ is just a potential delay that can occur in certain contexts. Note that also Moller and Tofts do not treat parallel composition completely.

Abstracting from internal computation. The strong faster–than precongruence requires that two systems have to match each others action transitions exactly, even those labeled with the internal action τ. Instead, one would like to abstract from τ's and develop a faster–than precongruence from the point of view of an external observer, as in CCS [15].

We start off with the definition of a *naive weak faster–than preorder* which requires us to introduce the following auxiliary notations. For any action α we define $\hat{\alpha} =_{\mathrm{df}} \epsilon$, if $\alpha = \tau$, and $\hat{\alpha} =_{\mathrm{df}} \alpha$, otherwise. Further, we let $\stackrel{\epsilon}{\Longrightarrow} =_{\mathrm{df}} \stackrel{\tau}{\longrightarrow}^*$ and write $P \stackrel{\alpha}{\Longrightarrow} Q$ if there exist R and S such that $P \stackrel{\epsilon}{\Longrightarrow} R \stackrel{\alpha}{\longrightarrow} S \stackrel{\epsilon}{\Longrightarrow} Q$.

Definition 12 (Naive weak faster–than preorder). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *naive weak faster–than relation* if, for all $\langle P, Q\rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \stackrel{\alpha}{\longrightarrow} P'$ implies $\exists Q'.\, Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
2. $Q \stackrel{\alpha}{\longrightarrow} Q'$ implies $\exists P'.\, P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
3. $P \stackrel{\sigma}{\longrightarrow} P'$ implies $\exists Q', Q'', Q'''.\, Q \stackrel{\epsilon}{\Longrightarrow} Q'' \stackrel{\sigma}{\longrightarrow} Q''' \stackrel{\epsilon}{\Longrightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.

We write $P \sqsupseteq_{\mathrm{n}} Q$ if $\langle P, Q\rangle \in \mathcal{R}$ for some naive weak faster–than relation $\mathcal{R}$.

Since no urgent action sets are considered, it is easy to see that $\sqsupseteq_{\mathrm{n}}$ is not a precongruence (cf. Def. 8).

Definition 13 (Weak faster–than preorder). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak faster–than relation* if, for all $\langle P, Q\rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \stackrel{\alpha}{\longrightarrow} P'$ implies $\exists Q'.\, Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
2. $Q \stackrel{\alpha}{\longrightarrow} Q'$ implies $\exists P'.\, P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $\langle P', Q'\rangle \in \mathcal{R}$.
3. $P \stackrel{\sigma}{\longrightarrow} P'$ implies $\exists Q', Q'', Q'''.\, Q \stackrel{\epsilon}{\Longrightarrow} Q'' \stackrel{\sigma}{\longrightarrow} Q''' \stackrel{\epsilon}{\Longrightarrow} Q'$, $\mathcal{U}(Q'') \subseteq \mathcal{U}(P)$, and $\langle P', Q'\rangle \in \mathcal{R}$.

We write $P \sqsupseteq Q$ if $\langle P, Q\rangle \in \mathcal{R}$ for some weak faster–than relation $\mathcal{R}$.

Hence, $\sqsupseteq$ is the largest weak faster–than relation and also a preorder. However, $\sqsupseteq$ is still not a precongruence for summation, but the summation fix used for other bisimulation–based timed process algebras proves effective for TACS, too.

Definition 14 (Weak faster–than precongruence). A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak faster–than precongruence relation* if, for all $\langle P, Q\rangle \in \mathcal{R}$ and $\alpha \in \mathcal{A}$:

1. $P \stackrel{\alpha}{\longrightarrow} P'$ implies $\exists Q'.\, Q \stackrel{\alpha}{\Longrightarrow} Q'$ and $P' \sqsupseteq Q'$.
2. $Q \stackrel{\alpha}{\longrightarrow} Q'$ implies $\exists P'.\, P \stackrel{\alpha}{\Longrightarrow} P'$ and $P' \sqsupseteq Q'$.
3. $P \stackrel{\sigma}{\longrightarrow} P'$ implies $\mathcal{U}(Q) \subseteq \mathcal{U}(P)$, and $\exists Q'.\, Q \stackrel{\sigma}{\longrightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.

We write $P \sqsupseteq Q$ if $\langle P, Q\rangle \in \mathcal{R}$ for a weak faster–than precongruence relation $\mathcal{R}$.

Theorem 15 (Full–abstraction). *The relation $\sqsupseteq$ is a precongruence for all operators except summation, and it is the largest such one contained in $\sqsupseteq_n$. Moreover, the relation $\sqsupseteq$ is the* largest precongruence *contained in $\sqsupseteq$, and hence the largest one contained in $\sqsupseteq_n$.*

5 Example: A 2–Place Storage

We demonstrate the utility of TACS by means of a small example dealing with two implementations of a 2–place storage in terms of an array and a buffer, respectively. Both can be defined using some definition of a 1–place buffer, e.g., $B_e =_{\mathrm{df}} \mu x.\sigma.in.\overline{out}.x$, which can alternately engage in communications with the environment on channels *in* and *out* [15]. Observe that we assume a communication on channel *out* to be urgent, while process B_e may autonomously delay a communication on channel *in* by one clock tick. Finally, subscript e of process B_e should indicate that the 1–place buffer is initially empty. On the basis of B_e, one may now define a 2–place array 2ARR and a 2–place buffer 2BUF as follows: $2\textsc{arr} =_{\mathrm{df}} B_e \,|\, B_e$ and $2\textsc{buf} =_{\mathrm{df}} (B_e[c/out] \,|\, B_e[c/in]) \setminus \{c\}$. While 2ARR is simply the parallel composition of two 1–place buffers, 2BUF is constructed by sequencing two 1–place buffers, i.e., by taking the output of the first 1–place buffer to be the input of the second one. Intuitively, we expect the array to behave functionally identical to the buffer, i.e., both should alternate between *in* and *out* actions. However, 2ARR should be faster than 2BUF since it can always output some of its contents immediately. In contrast, 2BUF needs to pass any item from the first to the second buffer cell, before it can output the item [15].

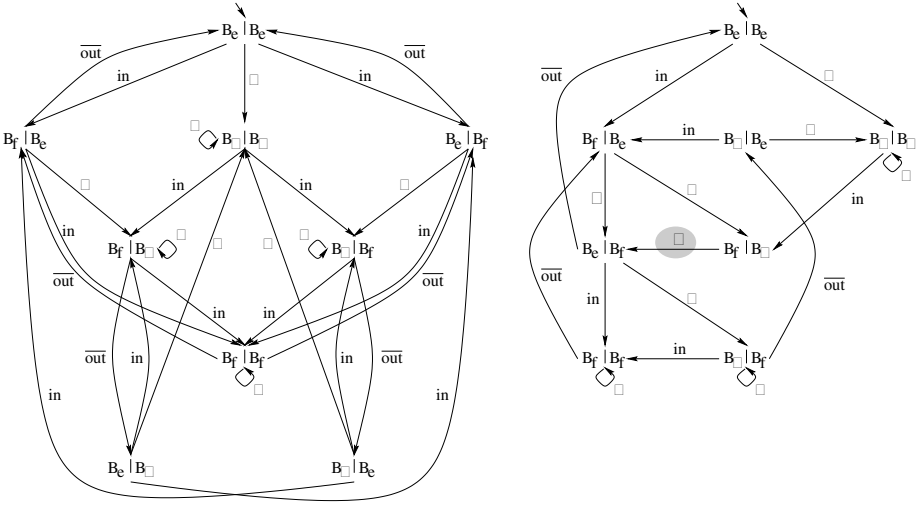


Fig. 1. Semantics of the array variant (left) and the buffer variant (right).

The semantics of the 2–place array 2ARR and our 2–place buffer 2BUF are depicted in Fig. 1 on the left and right, respectively. For notational convenience we let B_σ stand for the process $\mathrm{in}.\overline{\mathrm{out}}.B_e$ and B_f for $\overline{\mathrm{out}}.B_e$. Moreover, we leave out the restriction operator $\setminus\{c\}$ in the terms depicted for the buffer variant. The highlighted τ–transition indicates an urgent internal step of the buffer. Hence, process $(\mathrm{B_f}|\mathrm{B}_\sigma) \setminus \{c\}$ cannot engage in a clock transition. The other τ–

transition depicted in Fig. 1 is non–urgent. As desired, our semantic theory for TACS relates 2ARR and 2BUF. Formally, this may be witnessed by the weak faster–than relation given in Table 5, whence $\text{2ARR} \sqsupseteq\!\!\approx \text{2BUF}$. Moreover, since both 2ARR and 2BUF do not possess any initial internal transitions, they can also easily be proved to be weak faster–than precongruent, according to Def. 14. Thus, $\text{2ARR} \sqsupseteq\!\!\approx \text{2BUF}$, i.e., the 2–place array is faster than the 2–place buffer in all contexts, although functionally equivalent, which matches our intuition.

Table 5. Pairs in the considered weak faster–than relation

$\langle (\mathrm{B_e}\|\mathrm{B_e})\ ,\ (\mathrm{B_e}\|\mathrm{B_e}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_f}\|\mathrm{B_e})\ ,\ (\mathrm{B_f}\|\mathrm{B_e}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_e}\|\mathrm{B_f})\ ,\ (\mathrm{B_f}\|\mathrm{B_e}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_f}\|\mathrm{B_e})\ ,\ (\mathrm{B_e}\|\mathrm{B_f}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_f}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_f}\|\mathrm{B_\sigma}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_f}\|\mathrm{B_f})\ ,\ (\mathrm{B_f}\|\mathrm{B_f}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_f}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_\sigma}\|\mathrm{B_f}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_e}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_e}\|\mathrm{B_e}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_\sigma}\|\mathrm{B_e})\ ,\ (\mathrm{B_e}\|\mathrm{B_e}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_\sigma}\|\mathrm{B_f})\ ,\ (\mathrm{B_e}\|\mathrm{B_f}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_f}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_e}\|\mathrm{B_f}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_e}\|\mathrm{B_f})\ ,\ (\mathrm{B_e}\|\mathrm{B_f}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_f}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_f}\|\mathrm{B_e}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_\sigma}\|\mathrm{B_f})\ ,\ (\mathrm{B_f}\|\mathrm{B_e}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_\sigma}\|\mathrm{B_e})\ ,\ (\mathrm{B_\sigma}\|\mathrm{B_e}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_\sigma}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_\sigma}\|\mathrm{B_\sigma}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_\sigma}\|\mathrm{B_f})\ ,\ (\mathrm{B_f}\|\mathrm{B_\sigma}) \setminus \{c\}\rangle$	$\langle (\mathrm{B_\sigma}\|\mathrm{B_f})\ ,\ (\mathrm{B_\sigma}\|\mathrm{B_f}) \setminus \{c\}\rangle$
$\langle (\mathrm{B_e}\|\mathrm{B_\sigma})\ ,\ (\mathrm{B_\sigma}\|\mathrm{B_e}) \setminus \{c\}\rangle$		

6 Discussion and Related Work

The literature includes a large number of papers on timed process algebras [4]. We concentrate only on those which consider faster–than relations.

Research comparing the worst–case timing behavior of asynchronous systems initially centered around DeNicola and Hennessy's testing theory [9]; it was first conducted within the setting of Petri nets [6,13,20,21] and later for a TCSP–style [19] process algebra, called PAFAS [12,22]. The justification for adopting a testing approach is reflected in a fundamental result stating that the considered faster–than testing preorder based on continuous–time semantics coincides with the analogue testing preorder based on discrete–time semantics [12]. This result depends very much on the testing setting and is different from the sort of discretization obtained for timed automata. In PAFAS, every action has the same integrated upper time bound, namely 1. This gives a more realistic embedding of ordinary process terms, while a CCS–term in TACS runs in zero-time. In contrast, TACS allows one to specify arbitrary upper time bounds easily by nesting σ–prefixes. Also, the equational laws established for the faster–than testing preorder of PAFAS are quite complicated [22], while the simple axioms presented here provide a clear, comprehensive insight into our semantics.

Regarding other research of faster-than relations, our approach is most closely related to work by Moller and Tofts [17] who developed a bisimulation–based faster–than preorder within the discrete–time process algebra $\ell TCCS$ [16]. In their approach, asynchronous processes are modeled without any progress assumption. Instead, processes may idle arbitrarily long and, in addition, fixed delays may be specified. Hence, their setting is focused on best–case behavior, as the worst–case would be that for an arbitrary long time nothing happens. Moller and Tofts present an axiomatization of their faster–than preorder for finite sequential processes and discuss the problem of axiomatizing parallel composition,

for which only valid laws for special cases are provided. It has to be mentioned here that the axioms and the behavioral preorder of Moller and Tofts do not completely correspond. In fact, writing σ for what is actually written (1) in [17], $a.\sigma.b.\mathbf{0} + a.b.\mathbf{0}$ is equally fast as $a.b.\mathbf{0}$, which does not seem to be derivable from the axioms. Also, the intuition behind relating these processes is not so clear, since $a.a.\sigma.b.\mathbf{0} + a.a.b.\mathbf{0}$ is not necessarily faster than or equally fast as $a.a.b.\mathbf{0}$. Since the publication in 1991, also Moller and Tofts noticed this shortcoming of their preorder [*priv. commun.*]. The problem seems to lie in the way in which a transition $P \xrightarrow{a} P'$ of the faster process is matched: For intuitive reasons, the slower process must be allowed to perform time steps before engaging in a. Now the slower process is ahead in time, whence P' should be allowed some additional time steps. What might be wrong is that P' must perform these additional time steps immediately. We assume that a version of our indexed faster–than relation, which relaxes the latter requirement, would be more satisfactory. It would also be interesting to study the resulting preorder and compare it in detail to our faster–than precongruence.

A different idea for relating processes with respect to speed was investigated by Corradini et al. [8] within the so–called *ill–timed–but–well–caused* approach [1, 10]. The key of this approach is that components attach local time stamps to actions; however, actions occur as in an untimed algebra. Hence, in a sequence of actions exhibited by different processes running in parallel, local time stamps might decrease. Due to these "ill–timed" runs, the faster–than preorder of Corradini et al. is difficult to relate to our approach.

Other research compares the efficiency of untimed CCS–like terms by counting internal actions either within a testing framework [7,18] or a bisimulation–based setting [2,3]. Except in [7] which does not consider parallel composition, runs of parallel processes are seen to be the interleaved runs of their component processes. Consequently, e.g., $(\tau.a.\mathbf{0} \,|\, \tau.\overline{a}.b.\mathbf{0}) \setminus \{a\}$ is as efficient as $\tau.\tau.\tau.b.\mathbf{0}$, whereas in our setting $(\sigma.a.\mathbf{0} \,|\, \sigma.\overline{a}.b.\mathbf{0}) \setminus \{a\}$ is strictly faster than $\sigma.\sigma.\tau.b.\mathbf{0}$.

7 Conclusions and Future Work

To consider the worst–case efficiency of asynchronous processes, i.e., those processes whose functional behavior is not influenced by timing issues, we defined the process algebra TACS. This algebra conservatively extends CCS by a clock prefix which represents a delay of at most one time unit, and it takes time to be discrete. For TACS processes we then introduced a simple (bi)simulation–based faster–than preorder and showed this to coincide with two other variants of the preorder, both of which might be intuitively more convincing but which are certainly more complicated. We also developed a semantic theory for our preorder, including a coarsest precongruence result and an axiomatization for finite sequential processes, and investigated a corresponding "weak" preorder.

Regarding future work, we intend to extend our axiomatization to larger classes of processes and also to our weak faster–than preorder, as well as to implement TACS in an automated verification tool.

Acknowledgments. We would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] L. Aceto and D. Murphy. Timing and causality in process algebra. *Acta Inform.*, 33(4):317–350, 1996.
[2] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Inform.*, 29(8):737–760, 1992.
[3] S. Arun-Kumar and V. Natarajan. Conformance: A precongruence close to bisimilarity. In *STRICT '95*, Workshops in Comp., pp. 55–68. Springer-Verlag, 1995.
[4] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing: Real Time and Discrete Time*, ch. 10. In Bergstra et al. [5], 2001.
[5] J.A. Bergstra, A. Ponse, and S.A. Smolka, eds. *Handbook of Process Algebra*. Elsevier Science, 2001.
[6] E. Bihler and W. Vogler. Efficiency of token-passing MUTEX-solutions. In *ICATPN '98*, vol. 1420 of *LNCS*, pp. 185–204. Springer-Verlag, 1998.
[7] R. Cleaveland and A. Zwarico. A theory of testing for real time. In *LICS '91*, pp. 110–119. IEEE Computer Society Press, 1991.
[8] F. Corradini, R. Gorrieri, and M. Roccetti. Performance preorder and competitive equivalence. *Acta Inform.*, 34(11):805–835, 1997.
[9] R. DeNicola and M.C.B. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1983.
[10] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *TCS*, 140(1):73–94, 1995.
[11] M. Hennessy and T. Regan. A process algebra for timed systems. *Inform. and Comp.*, 117:221–239, 1995.
[12] L. Jenner and W. Vogler. Comparing the efficiency of asynchronous systems. In *ARTS '99*, vol. 1601 of *LNCS*, pp. 172–191. Springer-Verlag, 1999.
[13] L. Jenner and W. Vogler. Fast asynchronous systems in dense time. *TCS*, 254:379–422, 2001.
[14] G. Lüttgen and W. Vogler. A faster-than relation for asynchronous processes. Techn. Rep. 2001-2, ICASE, NASA Langley Research Center, USA, 2001.
[15] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[16] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCUR '90*, vol. 458 of *LNCS*, pp. 401–415. Springer-Verlag, 1990.
[17] F. Moller and C. Tofts. Relating processes with respect to speed. In *CONCUR '91*, vol. 527 of *LNCS*, pp. 424–438. Springer-Verlag, 1991.
[18] V. Natarajan and R. Cleaveland. An algebraic theory of process efficiency. In *LICS '96*, pp. 63–72. IEEE Computer Society Press, 1996.
[19] S. Schneider. An operational semantics for timed CSP. *Inform. and Comp.*, 116(2):193–213, 1995.
[20] W. Vogler. Faster asynchronous systems. In *CONCUR '95*, vol. 962 of *LNCS*, pp. 299–312. Springer-Verlag, 1995.
[21] W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. In *ICALP '97*, vol. 1256 of *LNCS*, pp. 538–548. Springer-Verlag, 1997.
[22] W. Vogler and L. Jenner. Axiomatizing a fragment of PAFAS. *ENTCS*, 39, 2000.

On the Power of Labels in Transition Systems

Jiří Srba*

BRICS**
Dept. of Computer Science, University of Aarhus, Denmark
srba@brics.dk

Abstract. In this paper we discuss the role of labels in transition systems with regard to bisimilarity and model checking problems. We suggest a general reduction from labelled transition systems to unlabelled ones, preserving bisimilarity and satisfiability of μ-calculus formulas. We apply the reduction to the class of transition systems generated by Petri nets and pushdown automata, and obtain several decidability/complexity corollaries for unlabelled systems. Probably the most interesting result is undecidability of strong bisimilarity for unlabelled Petri nets.

1 Introduction

Formal methods for verification of infinite-state systems have been an active area of research with a number of positive decidability results. In particular, verification techniques for concurrent systems defined by process algebras like CCS, ACP or CSP, pushdown systems, Petri nets, process rewrite systems and others have attracted a lot of attention. There are two central questions about decidability (complexity) of equivalence and model checking problems:

- **Equivalence checking** (see [Mol96]):
  Given two (infinite-state) systems, are they equal with regard to some equivalence notion?
- **Model checking** (see [BE97]):
  Given an (infinite-state) transition system and a formula ϕ of some suitable logic, does the system satisfy the property described by ϕ?

Both these problems have an interesting and unifying aspect in common. They can be defined independently on the computational model by means of *labelled transition systems*. All the models mentioned above give rise to a certain type of (infinite) labelled transition system and this is considered to be their desired semantics. Equivalence and model checking problems can be defined purely in terms of these transition systems.

In the first part of the paper we discuss the role of labels of such transition systems. There are two aspects of the branching structure described by a labelled

* The author is supported in part by the GACR, grant No. 201/00/0400.
** **B**asic **R**esearch in **C**omputer **S**cience,
Centre of the Danish National Research Foundation.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 277–291, 2001.
© Springer-Verlag Berlin Heidelberg 2001

transition system T. First, given a state of T, there can be several outgoing edges with different labels. Second, given a state of T and a label a, there can be several outgoing edges under the same label a. We claim that for our purposes only the second property is the essential one. In other words, given a labelled transition system, we can construct another transition system where all edges are labelled by the same label, i.e., the labels are in fact completely irrelevant. We call such systems *unlabelled transition systems*. What is important is the fact that our construction preserves the answers to both the questions we are interested in — equivalence checking (and we have chosen *strong bisimilarity* as the notion of equivalence) and model checking with *action-based modal μ-calculus* as the chosen logic for expressing properties of labelled transition systems.

In the second part we focus on two specific classes of infinite-state systems, namely *Petri nets* and *pushdown systems*. Petri nets are a typical example of fully parallel models of computation, whereas pushdown systems can model sequential stack-like process behaviours. Both Petri nets and pushdown systems generate (in general infinite) labelled transition systems. The question is whether the transformed unlabelled transition systems (given by the construction mentioned in the previous paragraph) are still definable by the chosen formalism of Petri nets resp. pushdown automata. The answer is shown to be positive for both our models — there are even polynomial time transformations. This implies several decidability/complexity results about bisimilarity and model checking problems for unlabelled Petri nets and pushdown systems.

Probably the most interesting corollary is the application of the transformation to Petri nets. We prove that strong bisimilarity for unlabelled Petri nets (where the set of labels is a singleton set) is undecidable. This is stronger result than undecidability of strong bisimilarity for labelled Petri nets given by Jancar [Jan95]. The undecidability for unlabelled Petri nets contrasts to a positive decidability result for the subclass of Petri nets which are deterministic [Jan95, Vog92], i.e., for any marking M and a label a there is at most one outgoing transition from M labelled by a. This again demonstrates that the role of labels is not important for decidability questions and what is crucial is the branching structure induced by transitions with the same label.

Note: full and extended version of this paper appears as [Srb01].

2 Basic Definitions

Definition 1 (Labelled transition system). *A* labelled transition system *is a triple $T = (S, \mathcal{A}ct, \longrightarrow)$ where S is a set of* states *(or* processes*), $\mathcal{A}ct$ is a set of* labels *(or* actions*) such that $S \cap \mathcal{A}ct = \emptyset$, and $\longrightarrow \subseteq S \times \mathcal{A}ct \times S$ is a* transition relation*, written $\alpha \xrightarrow{a} \beta$ for $(\alpha, a, \beta) \in \longrightarrow$.*

In what follows we assume that $\mathcal{A}ct$ is a finite set. As usual we extend the transition relation to the elements of $\mathcal{A}ct^*$. We also write $\alpha \longrightarrow^* \beta$ iff $\exists w \in \mathcal{A}ct^*$ such that $\alpha \xrightarrow{w} \beta$. A state β is *reachable* from a state α, iff $\alpha \longrightarrow^* \beta$. Moreover, we write $\alpha \not\longrightarrow$ for $\alpha \in S$ iff there is no $\beta \in S$ and $a \in \mathcal{A}ct$ such that $\alpha \xrightarrow{a} \beta$. We call a labelled transition system *normed* iff $\forall s \in S.\ \exists s' \in S$: $s \longrightarrow^* s' \not\longrightarrow$.

Definition 2. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s \in S$. By T_s we denote a labelled transition system restricted to states of T reachable from s. More precisely, $T_s = (S_s, \mathcal{A}ct, \longrightarrow_s)$ where $S_s = \{s' \in S \mid s \longrightarrow^* s'\}$ and $s_1 \xrightarrow{a}_s s_2$ iff $s_1 \xrightarrow{a} s_2$ and $s_1, s_2 \in S_s$.*

Now, we introduce the notion of *(strong) bisimilarity.*

Definition 3 (Bisimulation). *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system. A binary relation $R \subseteq S \times S$ is a relation of* bisimulation *iff whenever $(\alpha, \beta) \in R$ then for each $a \in \mathcal{A}ct$:*

- *if $\alpha \xrightarrow{a} \alpha'$ then $\beta \xrightarrow{a} \beta'$ for some β' such that $(\alpha', \beta') \in R$*
- *if $\beta \xrightarrow{a} \beta'$ then $\alpha \xrightarrow{a} \alpha'$ for some α' such that $(\alpha', \beta') \in R$.*

Two states $\alpha, \beta \in S$ are bisimilar in T*, written $\alpha \sim_T \beta$, iff there is a bisimulation R such that $(\alpha, \beta) \in R$.*

Bisimilarity has also an elegant characterisation in terms of *bisimulation games* [Tho93,Sti95]. A bisimulation game on a pair of states $\alpha, \beta \in S$ is a two-player game of an "attacker" and a "defender". The attacker chooses one of the states and makes an $\xrightarrow{a}$-move for some $a \in \mathcal{A}ct$. The defender must respond by making an $\xrightarrow{a}$-move from the other state under the same label a. Now the game repeats, starting from the new processes. If one player cannot move, the other player wins. If the game is infinite, the defender wins. States α and β are bisimilar iff the defender has a winning strategy (and non-bisimilar iff the attacker has a winning strategy).

Definition 4 (Unlabelled transition system). *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system. We call T* unlabelled transition system *whenever $\mathcal{A}ct$ is a singleton set, i.e., $|\mathcal{A}ct| = 1$.*

Remark 1. If it is the case that $|\mathcal{A}ct| = 1$ then (for our purposes) we simply write $\longrightarrow$ instead of $\xrightarrow{a}$. We also forget about the second component in the definition of a labelled transition system, i.e., we can denote an unlabelled transition system by $T = (S, \longrightarrow)$ where $\longrightarrow \subseteq S \times S$.

We define a powerful logic for labelled transition systems — modal μ-calculus.

Definition 5 (Syntax of modal μ-calculus). *Let $\mathcal{V}ar$ be a set of variables and $\mathcal{A}ct$ a set of action labels such that $\mathcal{V}ar \cap \mathcal{A}ct = \emptyset$. The syntax of modal μ-calculus is defined as follows:*

$$\phi ::= \mathtt{tt} \mid X \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \langle a \rangle \phi \mid \mu X.\phi$$

where tt *stands for "true", X ranges over $\mathcal{V}ar$ and a over $\mathcal{A}ct$. There is a standard restriction on the formulas: we consider only formulas where each occurrence of a variable X is within a scope of an even number of negation symbols.*

Given a labelled transition system $T = (S, \mathcal{A}ct, \longrightarrow)$, we interpret a formula ϕ as follows. Assume a valuation $\mathcal{V}al : \mathcal{V}ar \to 2^S$.

$$
\begin{aligned}
[\![\mathtt{tt}]\!]_{\mathcal{V}al,T} &= S \\
[\![X]\!]_{\mathcal{V}al,T} &= \mathcal{V}al(X) \\
[\![\phi_1 \wedge \phi_2]\!]_{\mathcal{V}al,T} &= [\![\phi_1]\!]_{\mathcal{V}al,T} \cap [\![\phi_2]\!]_{\mathcal{V}al,T} \\
[\![\neg\phi]\!]_{\mathcal{V}al,T} &= S \setminus [\![\phi]\!]_{\mathcal{V}al,T} \\
[\![\langle a \rangle \phi]\!]_{\mathcal{V}al,T} &= \{s \mid \exists s'.\ (s \xrightarrow{a} s' \ \wedge \ s' \in [\![\phi]\!]_{\mathcal{V}al,T})\} \\
[\![\mu X.\phi]\!]_{\mathcal{V}al,T} &= \bigcap \{S' \subseteq S \mid [\![\phi]\!]_{\mathcal{V}al[S'/X],T} \subseteq S'\}
\end{aligned}
$$

Here $\mathcal{V}al[S'/X]$ stands for a valuation function such that $\mathcal{V}al[S'/X](X) = S'$ and $\mathcal{V}al[S'/X](Y) = \mathcal{V}al(Y)$ for $X \neq Y$. We say that a formula ϕ is satisfied in a state s of T, and we write $T, s \models \phi$, if for all valuations $\mathcal{V}al$ we have $s \in [\![\phi]\!]_{\mathcal{V}al,T}$.

Remark 2. The logic defined above without the fixed-point operator $\mu X.\phi$ is called *Hennessy-Milner logic* [HM85].

3 From Labelled to Unlabelled Transition Systems

In this section we present a transformation from labelled transition systems to unlabelled ones, preserving bisimilarity and satisfiability of μ-calculus formulas.

Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system. We define a transformed unlabelled transition system $\widehat{T} = (\widehat{S}, \longrightarrow)$. We reuse the relation symbol $\longrightarrow$ without causing confusion, since in the system T it is a ternary relation and in $\widehat{T}$ it is a binary relation. W.l.o.g. assume that $\mathcal{A}ct = \{1, 2, \ldots, n\}$ for some $n > 0$. We define the system $\widehat{T} = (\widehat{S}, \longrightarrow)$ as follows:

$$
\begin{aligned}
\widehat{S} \quad &= S \cup \{r^k_{(s,a,s')} \mid 0 \leq k \leq a \ \wedge \ s \xrightarrow{a} s'\} \cup \{d^k_s \mid s \in S \ \wedge \ 0 \leq k \leq n\} \\
\longrightarrow \ &= \{(s, r^0_{(s,a,s')}),\ (r^0_{(s,a,s')}, s') \mid s \xrightarrow{a} s'\} \cup \\
&\quad \{(r^k_{(s,a,s')}, r^{k+1}_{(s,a,s')}) \mid s \xrightarrow{a} s' \ \wedge \ 0 \leq k < a\} \cup \\
&\quad \{(s, d^0_s) \mid s \in S\} \cup \{(d^k_s, d^{k+1}_s) \mid s \in S \ \wedge \ 0 \leq k < n\}.
\end{aligned}
$$

For a better understanding of the transformation take a look at Figure 1 where a way how to transform a transition $s \xrightarrow{a} s'$ is drawn. The idea consists in splitting each transition $s \xrightarrow{a} s'$ labelled by $a \in \mathbb{N}_0$ with an intermediate state (the $r^0_{(s,a,s')}$ state) out of which goes a newly added linear path of length a. The d_s states add a linear path of length $n+1$ to each state from S and serve for distinguishing the r-states from the original ones.

Notice that if T is a finite-state system then the size of $\widehat{T}$ is polynomially bounded by the size of T. In fact, we could add only one linear path of length $n+1$ with appropriate links into the path starting in the states from S and in the r^0-states. However, for technical convenience in Section 4, we use the previously described construction.

Remark 3. It is an easy observation that $\widehat{T}$ is a normed transition system.

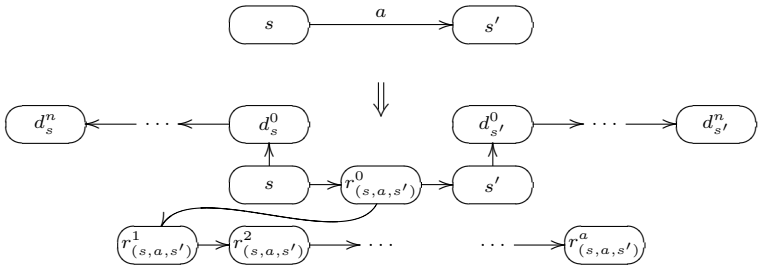


Fig. 1. Transformation of a transition $s \xrightarrow{a} s'$

3.1 Bisimilarity

Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and let $s \in S$. We define a *set of finite norms of* s by $\mathcal{N}(s) = \{|w| \mid \exists s' \in S : s \xrightarrow{w} s' \not\longrightarrow\}$ where $|w|$ is the length of w. The following proposition is a standard one.

Proposition 1. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s_1, s_2 \in S$. Then $s_1 \sim_T s_2$ implies that $\mathcal{N}(s_1) = \mathcal{N}(s_2)$.*

Our aim is to show that for a pair of states s_1 and s_2 of a labelled transition system T holds that $s_1 \sim_T s_2$ if and only if $s_1 \sim_{\widehat{T}} s_2$.

Lemma 1. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s_1, s_2 \in S$ be a pair of states. If $s_1 \sim_T s_2$ then $s_1 \sim_{\widehat{T}} s_2$.*

Proof. We can naturally define a wining strategy for the defender in $\widehat{T}$ under the assumption that $s_1 \sim_T s_2$. Details can be found in [Srb01]. □

Before showing the other implication, we prove the following property.

Property 1. The attacker in $\widehat{T}$ has a winning strategy from any pair of states $s_1, s_2 \in \widehat{S}$ such that $s_1 \notin S$ and $s_2 \in S$, or $s_1 \in S$ and $s_2 \notin S$.

Proof. Assume w.l.o.g. that $s_1 \notin S$ and $s_2 \in S$. The other case is symmetric. There are three possibilities if $s_1 \notin S$.

- Let $s_1 = d^k_s$ for some $s \in S$ and $0 \leq k \leq n$, or $s_1 = r^k_{(s,a,s')}$ for some $s, s' \in S$, $a \in \mathcal{A}ct$ and $0 < k \leq a$. In both these cases $n + 1 \notin \mathcal{N}(s_1)$ and $n + 1 \in \mathcal{N}(s_2)$. Because of Proposition 1 we get $s_1 \not\sim_{\widehat{T}} s_2$ and the attacker in $\widehat{T}$ has a winning strategy.
- Let $s_1 = r^0_{(s,a,s')}$ for some $s, s' \in S$ and $a \in \mathcal{A}ct$. Now the attacker has the following winning strategy in $\widehat{T}$. He makes a move $r^0_{(s,a,s')} \longrightarrow r^1_{(s,a,s')}$. Assume a defender's answer $s_2 \longrightarrow s'_2$ for an arbitrary $s'_2 \in \widehat{S}$. Obviously either $n \in \mathcal{N}(s'_2)$ or $n + 2 \in \mathcal{N}(s'_2)$ and $\max[\mathcal{N}(r^1_{(s,a,s')})] < n$. Again, using Proposition 1, the attacker has a winning strategy. □

Lemma 2. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s_1, s_2 \in S$ be a pair of states. If $s_1 \sim_{\widehat{T}} s_2$ then $s_1 \sim_T s_2$.*

Proof. Knowing that the defender has a winning strategy in $\widehat{T}$ from s_1 and s_2, we establish a winning strategy for the defender in T from s_1 and s_2. Suppose that the attacker's move in T is $s_i \xrightarrow{a} s'_i$ for $i \in \{1, 2\}$. Then it is possible to perform a series of two moves $s_i \longrightarrow r^0_{(s_i,a,s'_i)} \longrightarrow s'_i$ in $\widehat{T}$. Because of Property 1, the defender in $\widehat{T}$ has a response to this series of moves only by performing $s_{3-i} \longrightarrow r^0_{(s_{3-i},b,s'_{3-i})} \longrightarrow s'_{3-i}$ for some $b \in \mathcal{A}ct$ and $s'_{3-i} \in S$ where

$$s'_1 \sim_{\widehat{T}} s'_2. \tag{1}$$

Notice that $a = b$, otherwise the attacker has a winning strategy in $\widehat{T}$ from $r^0_{(s_i,a,s'_i)}$ and $r^0_{(s_{3-i},b,s'_{3-i})}$ by performing a move $r^0_{(s_i,a,s'_i)} \longrightarrow r^1_{(s_i,a,s'_i)}$. Using Property 1, the defender must answer with $r^0_{(s_{3-i},b,s'_{3-i})} \longrightarrow r^1_{(s_{3-i},b,s'_{3-i})}$. However, the attacker has a winning strategy now since $a - 1 \in \mathcal{N}(r^1_{(s_i,a,s'_i)})$ and $a - 1 \notin \mathcal{N}(r^1_{(s_{3-i},b,s'_{3-i})})$ whenever $a \neq b$ — Proposition 1. This implies that the defender in T can perform $s_{3-i} \xrightarrow{a} s'_{3-i}$ and because of (1), the defender in T has a winning strategy from s'_1 and s'_2. Thus $s_1 \sim_T s_2$. □

By Lemma 1 and Lemma 2 we can conclude with the following theorem.

Theorem 1. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s_1, s_2 \in S$ be a pair of states. Let $\widehat{T}$ be the corresponding unlabelled transition system. Then*

$$s_1 \sim_T s_2 \quad \textit{if and only if} \quad s_1 \sim_{\widehat{T}} s_2.$$

3.2 Model Checking

We turn our attention to the model checking problem now. We show that there is a polynomial time transformation of any μ-calculus formula ϕ into $\widehat{\phi}$ such that $T, s \models \phi$ iff $\widehat{T}, s \models \widehat{\phi}$. When interpreting a μ-calculus formula on an unlabelled transition system $\widehat{T}$, we write $\Diamond$ instead of $\langle a \rangle$, since $a \in \mathcal{A}ct$ is the only label and hence it is irrelevant. We also define a dual operator $\Box$ as $\Box\phi \equiv \neg\Diamond\neg\phi$ and ff as $\mathsf{ff} \equiv \neg\mathsf{tt}$.

Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system such that $\mathcal{A}ct = \{1, 2, \ldots, n\}$ and let $\widehat{T} = (\widehat{S}, \longrightarrow)$ be the corresponding unlabelled system. First of all, we write a formula $\mathcal{L}(a)$ such that

$$[\![\mathcal{L}(a)]\!]_{\mathcal{V}al', \widehat{T}} = \{r^0_{(s,a,s')} \mid \exists s, s' \in S : s \xrightarrow{a} s'\} \tag{2}$$

for any valuation $\mathcal{V}al' : \mathcal{V}ar \to 2^{\widehat{S}}$. We define $\mathcal{L}(a) \equiv \Diamond^{n+1}\mathsf{tt} \wedge \Diamond(\Box^a\mathsf{ff} \wedge \Diamond^{a-1}\mathsf{tt})$ where $\Diamond^0\phi \equiv \phi$ and $\Diamond^{k+1}\phi \equiv \Diamond(\Diamond^k\phi)$, and similarly $\Box^0\phi \equiv \phi$ and $\Box^{k+1}\phi \equiv \Box(\Box^k\phi)$. Let $\widehat{T}, s_1 \models \mathcal{L}(a)$. The left subformula in $\mathcal{L}(a)$, namely $\Diamond^{n+1}\mathsf{tt}$, ensures

that the state s_1 is not of the form $r^k_{(s,b,s')}$ for $k > 0$, nor of the form d^k_s for $k \geq 0$. The second subformula in the conjunction says that there is a one step transition from s_1, reaching a state s'_1 of the form $r^1_{(s,b,s')}$ — should $s'_1 \in S$, or s'_1 be of the form $r^0_{(s,b,s')}$, or s'_1 be of the form d^0_s, then the formula $\Box^a\mathsf{ff}$ can never be satisfied. Moreover, the formula $\Box^a\mathsf{ff}$ guarantees that there are at most $a-1$ transitions from $r^1_{(s,b,s')}$ and the formula $\Diamond^{a-1}\mathsf{tt}$ finally ensures that at least $a-1$ transitions can be performed from $r^1_{(s,b,s')}$. Hence $a = b$ and (2) is established.

Let us now consider another formula defined by $\mathcal{S}tate \equiv \Diamond\mathsf{tt} \wedge \Box\Diamond^n\mathsf{tt}$. Obviously, $[\![\mathcal{S}tate]\!]_{\mathcal{V}al',\widehat{T}} = S$ for any valuation $\mathcal{V}al' : \mathcal{V}ar \to 2^{\widehat{S}}$. We are now ready to define $\widehat{\phi}$ for a given μ-calculus formula ϕ. The definition follows:

$$\begin{aligned}
\widehat{\mathsf{tt}} &= \mathsf{tt} \wedge \mathcal{S}tate \\
\widehat{X} &= X \wedge \mathcal{S}tate \\
\widehat{\phi_1 \wedge \phi_2} &= \widehat{\phi_1} \wedge \widehat{\phi_2} \wedge \mathcal{S}tate \\
\widehat{\neg\phi} &= \neg\widehat{\phi} \wedge \mathcal{S}tate \\
\widehat{\mu X.\phi} &= (\mu X.\widehat{\phi}) \wedge \mathcal{S}tate \\
\widehat{\langle a\rangle\phi} &= \Diamond\big(\mathcal{L}(a) \wedge \Diamond\widehat{\phi}\big) \wedge \mathcal{S}tate.
\end{aligned}$$

Theorem 2. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a labelled transition system and $s \in S$. Let ϕ be a μ-calculus formula. Then*

$$T, s \models \phi \quad \textit{if and only if} \quad \widehat{T}, s \models \widehat{\phi}.$$

Proof. By structural induction on ϕ it is provable that

$$[\![\phi]\!]_{\mathcal{V}al,T} = [\![\widehat{\phi}]\!]_{\mathcal{V}al',\widehat{T}}$$

for arbitrary valuations $\mathcal{V}al : \mathcal{V}ar \to 2^S$ and $\mathcal{V}al' : \mathcal{V}ar \to 2^{\widehat{S}}$ such that $\mathcal{V}al(X) = \mathcal{V}al'(X) \cap S$ for all $X \in \mathcal{V}ar$. Full proof can be found in [Srb01]. □

Remark 4. Let us consider temporal operators $EF\phi$ and $EG\phi$ defined by $EF\phi \equiv \mu X.\phi \vee \langle -\rangle X$ and $EG\phi \equiv \neg\mu X.\neg\phi \vee \big(\neg\langle -\rangle\neg X \wedge \langle -\rangle\mathsf{tt}\big)$ such that $\langle -\rangle\phi \equiv \bigvee_{a\in\mathcal{A}ct}\langle a\rangle\phi$. We define the transformed formulas $\widehat{EF\phi}$ (using only EF operator) and $\widehat{EG\phi}$ (using only EG operator) as follows:

$$\begin{aligned}
&\widehat{EF\phi} = EF\widehat{\phi} \;\wedge\; \mathcal{S}tate \\
&\widehat{EG\phi} = EG\left(\big(\mathcal{S}tate \vee \textstyle\bigvee_{a\in\mathcal{A}ct}\mathcal{L}(a)\big) \;\wedge\; \mathcal{S}tate \implies \widehat{\phi}\right) \wedge \mathcal{S}tate.
\end{aligned}$$

Note that still $[\![\widehat{\phi}]\!]_{\mathcal{V}al',\widehat{T}} \subseteq S$ for any formula ϕ and any valuation $\mathcal{V}al' : \mathcal{V}ar \to 2^{\widehat{S}}$. Let $s \in S$. Then $T, s \models EF\phi$ iff $\widehat{T}, s \models \widehat{EF\phi}$. If moreover T_s satisfies condition

$$\forall s' \in S_s.\ \exists s'' \in S_s.\ \exists a \in \mathcal{A}ct : s' \xrightarrow{a} s'' \tag{3}$$

then $T, s \models EG\phi$ iff $\widehat{T}, s \models \widehat{EG\phi}$. This enables to transform formulas of even weaker logics than modal μ-calculus (such as Hennessy-Milner logic, possibly equipped with the operator EF, respectively EG) into unlabelled formulas of the same logic. Hennessy-Milner logic with the operators EF and EG is called *unified system of branching-time logic* (UB) [BAMP83] and the fragments of UB containing only the operator $EF\phi$ ($EG\phi$) are referred to as EF-logic (EG-logic).

Similarly, the until operators $E\,[\phi U \psi]$ and $A\,[\phi U \psi]$ of CTL [CE81] — defined by $E\,[\phi U \psi] \equiv \mu X.\psi \vee (\phi \wedge \langle - \rangle X)$ and $A\,[\phi U \psi] \equiv \mu X.\psi \vee \big(\phi \wedge \langle - \rangle \mathtt{tt} \wedge \neg\langle - \rangle \neg X\big)$ — can be transformed:

$$\widehat{E\,[\phi U \psi]} = E\,[(\mathcal{S}tate \implies \widehat{\phi})\ U\ \widehat{\psi}]\ \wedge\ \mathcal{S}tate$$
$$\widehat{A\,[\phi U \psi]} = \widehat{\chi^{AU}} \quad \text{where } \chi^{AU} = \neg\Big(E\,[\neg\psi\ U\ (\neg\phi \wedge \neg\psi)]\ \vee\ EG(\neg\psi)\Big).$$

In the case of $A\,[\phi U \psi]$ we use the equivalence $A\,[\phi U \psi] \iff \chi^{AU}$ from [CES86]. Again, for any $s \in S$ it holds that $T, s \models E\,[\phi U \psi]$ iff $\widehat{T}, s \models \widehat{E\,[\phi U \psi]}$. Moreover $T, s \models A\,[\phi U \psi]$ iff $\widehat{T}, s \models \widehat{A\,[\phi U \psi]}$ under the assumption of condition (3). This enables to transform also the logic CTL.

4 Applications

In this section we show how the previous results can be applied to bisimilarity/model checking of infinite-state systems. We focus in particular on a typical representative of parallel models — Petri nets (see e.g. [Pet81]) — and sequential processes — pushdown systems (see e.g. [Mol96]). We have to show that the class of transition systems generated by these models is closed under the transformation from labelled to unlabelled systems as presented in the previous section.

First of all, we remind the reader of the fact that our transformation works immediately for finite-state transition systems. In the following corollary we consider the model checking problem with these logics: Hennessy-Milner logic, EF-logic, EG-logic, UB, CTL and modal μ-calculus.

Corollary 1. *Let $T = (S, \mathcal{A}ct, \longrightarrow)$ be a finite-state labelled transition system, i.e., $|S|, |\mathcal{A}ct| < \infty$. There is a polynomial time reduction from the bisimilarity (model) checking problem for T to the bisimilarity (model) checking problem for $\widehat{T}$, where $\widehat{T}$ is an unlabelled (and finite-state) transition system.*

Proof. Immediately from Theorem 1, Theorem 2 and Remark 4. In the case of EG-logic, UB and CTL we can ensure the validity of condition (3) of Remark 4 by adding a self-loop $s \xrightarrow{u} s$ (u is a fresh action) to every state $s \in S$ such that $s \not\longrightarrow$. This does not influence satisfiability of EG, UB and CTL formulas. □

4.1 Petri Nets

It is a well known fact that the bisimilarity checking problem is undecidable for labelled Petri nets [Jan95]. The technique of the proof is based on a reduction from the counter machine of Minsky and the labelling is essential for the

reduction. It is also known that bisimilarity is decidable for the class of Petri nets which are deterministic up to bisimilarity [Jan95], i.e., $\mathcal{F}$-deterministic nets of Vogler [Vog92]. Bisimilarity between a labelled Petri net and a finite-state system is decidable [JM95,JKM98] and EXPSPACE-hard (see e.g. comments in [May00]).

Model checking of even weak temporal logics on labelled transition systems generated by Petri nets is quite pessimistic. The only decidable logic is (trivially) Hennessy-Milner logic. The EF-logic is undecidable [Esp97] and model checking with EG is also undecidable, even for BPP [EK95] — BPP is a strict subclass of labelled Petri nets where each transition has exactly one input place.

Definition 6 (Labelled Petri net). *A* labelled Petri net *is a tuple* $N = (P, T, F, L, \lambda)$*, where* P *is a finite set of* places*,* T *is a finite set of* transitions *such that* $T \cap P = \emptyset$*,* $F \subseteq (P \times T) \cup (T \times P)$ *is a* flow relation*,* L *is a finite set of* labels *and* $\lambda : T \to L$ *is a* labelling function*.*

A *marking* M of a net N is a mapping $M : P \to \mathbb{N}_0$, i.e., each place is assigned a nonnegative number of *tokens*. We define ${}^\bullet t = \{p \mid (p,t) \in F\}$ and $t^\bullet = \{p \mid (t,p) \in F\}$ for a transition $t \in T$. We say that $t \in T$ is *enabled* in a marking M iff $\forall p \in {}^\bullet t.\ M(p) > 0$. If t is enabled in M then it can be *fired*, producing a marking M' such that:

- $M'(p) = M(p)$ for all $p \in \big(P \setminus ({}^\bullet t \cup t^\bullet)\big) \cup ({}^\bullet t \cap t^\bullet)$
- $M'(p) = M(p) - 1$ for all $p \in {}^\bullet t \setminus t^\bullet$
- $M'(p) = M(p) + 1$ for all $p \in t^\bullet \setminus {}^\bullet t$.

Then we write $M[t\rangle M'$. W.l.o.g. we assume that if $M[t_1\rangle M'$ and $M[t_2\rangle M'$, then $\lambda(t_1) \neq \lambda(t_2)$ for any pair of markings M, M' and transitions t_1, t_2.

Definition 7 (Labelled transition system $T(N)$).
Let $N = (P, T, F, L, \lambda)$ *be a labelled Petri net. We define a corresponding labelled transition system* $T(N)$ *as* $T(N) = ([P \to \mathbb{N}_0], L, \longrightarrow)$ *where* $M \xrightarrow{a} M'$ *whenever* $M[t\rangle M'$ *and* $a = \lambda(t)$ *for* $M, M' \in [P \to \mathbb{N}_0]$ *and* $t \in T$*.*

Now, we define unlabelled Petri nets.

Definition 8 (Unlabelled Petri net). *An* unlabelled Petri net *is a labelled Petri net* $N = (P, T, F, L, \lambda)$ *such that* $|L| = 1$*.*

Remark 5. Whenever $|L| = 1$, let us say $L = \{a\}$, we omit L and λ from the definition of the net N and instead of $M \xrightarrow{a} M'$ in $T(N)$ we simply write $M \longrightarrow M'$.

Let $N = (P, T, F, L, \lambda)$ be a labelled Petri net. W.l.o.g. assume that $L = \{1, \ldots, n\}$ for some $n > 0$. We construct an unlabelled Petri net $N' = (P', T', F')$ and a mapping $\psi : (P \to \mathbb{N}_0) \to (P' \to \mathbb{N}_0)$ such that $\widehat{T(N)_{M_1}}$ and $T(N')_{\psi(M_1)}$ are isomorphic unlabelled transition systems for any marking M_1 of N. Let us recall that $\widehat{T(N)_{M_1}}$ is the transition system restricted to markings reachable from M_1 and $T(N')_{\psi(M_1)}$ is restricted to markings reachable from $\psi(M_1)$ — see Definition 2. The net N' is defined as follows:

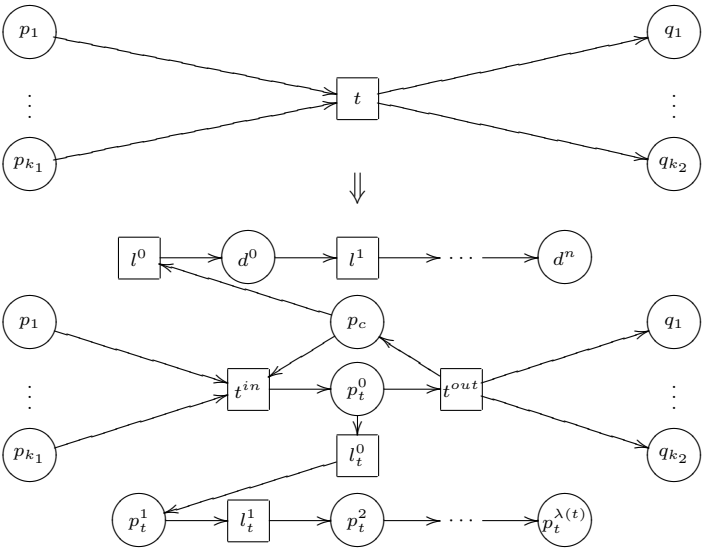


Fig. 2. Transformation of a transition t

$$P' = P \cup \{p_t^k \mid t \in T \ \wedge \ 0 \leq k \leq \lambda(t)\} \cup \{p_c\} \cup \{d^k \mid 0 \leq k \leq n\}$$

$$T' = \{t^{in}, t^{out} \mid t \in T\} \cup \{l_t^k \mid t \in T \ \wedge \ 0 \leq k < \lambda(t)\} \cup \{l^k \mid 0 \leq k \leq n\}$$

$$\begin{aligned} F' = {} & \{(p, t^{in}) \mid (p,t) \in F\} \cup \{(t^{out}, p) \mid (t,p) \in F\} \cup \\ & \{(t^{in}, p_t^0), (p_t^0, t^{out}) \mid t \in T\} \cup \\ & \{(p_t^k, l_t^k), (l_t^k, p_t^{k+1}) \mid t \in T \ \wedge \ 0 \leq k < \lambda(t)\} \cup \\ & \{(p_c, t^{in}), (t^{out}, p_c) \mid t \in T\} \cup \\ & \{(p_c, l^0)\} \cup \{(l^k, d^k), (d^k, l^{k+1}) \mid 0 \leq k < n\} \cup \{(l^n, d^n)\}. \end{aligned}$$

In this construction each transition t with input places $p_1, \ldots, p_{k_1}$ and output places $q_1, \ldots, q_{k_2}$ is transformed into a set of transitions shown in Figure 2. Now, we give the mapping ψ. Let $M \in (P \to \mathbb{N}_0)$. Then $\psi(M) : P' \to \mathbb{N}_0$ is defined by

$$\psi(M)(p) = \begin{cases} 1 & \text{if } p = p_c \\ M(p) & \text{if } p \in P \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 3. *Let $N = (P, T, F, L, \lambda)$ be a labelled Petri net and $N' = (P', T', F')$ the unlabelled Petri net defined above. Then $\widehat{T(N)_{M_1}}$ and $T(N')_{\psi(M_1)}$ are isomorphic unlabelled transition systems for any $M_1 \in [P \to \mathbb{N}_0]$.*

Proof. Assume that $\widehat{T(N)_{M_1}} = (S_1, \longrightarrow_1)$ and $T(N')_{\psi(M_1)} = (S_2, \longrightarrow_2)$. Recall that $S_1 \subseteq [P \to \mathbb{N}_0] \cup \{r^k_{(M,\lambda(t),M')} \mid M[t\rangle M' \ \wedge \ 0 \leq k \leq \lambda(t)\} \cup \{d^k_M \mid M \in$

$[P \to \mathbb{N}_0] \ \wedge \ 0 \leq k \leq n\}$ and $S_2 \subseteq [P' \to \mathbb{N}_0]$. We define a mapping $f : S_1 \to S_2$ by

$$f(s_1) = \begin{cases} \psi(s_1) & \text{if } s_1 \in [P \to \mathbb{N}_0] \\ \overline{M} & \text{if } s_1 = r^k_{(M,\lambda(t),M')} \text{ such that } M[t\rangle M' \\ \overline{\overline{M}} & \text{if } s_1 = d^k_M \text{ such that } M \in [P \to \mathbb{N}_0] \end{cases}$$

where

$$\overline{M}(p) = \begin{cases} M(p) & \text{if } p \in P \setminus {}^\bullet t \\ M(p) - 1 & \text{if } p \in {}^\bullet t \\ 1 & \text{if } p = p^k_t \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \overline{\overline{M}}(p) = \begin{cases} M(p) & \text{if } p \in P \\ 1 & \text{if } p = d^k \\ 0 & \text{otherwise.} \end{cases}$$

Let $s_1 \longrightarrow_1 s_1'$ for some $s_1, s_1' \in S_1$. It can be easily seen that $f(s_1) \longrightarrow_2 f(s_1')$. On the other hand, let $M_2 \longrightarrow_2 M_2'$ and $M_2 = f(s_1)$ for some $s_1 \in S_1$ and $M_2, M_2' \in S_2$. Then there exists $s_1' \in S_1$ such that $M_2' = f(s_1')$ and $s_1 \longrightarrow_1 s_1'$. This implies that f is surjective and moreover f is trivially injective. Hence, $\widehat{T(N)_{M_1}}$ and $T(N')_{\psi(M_1)}$ are isomorphic unlabelled transition systems. □

Theorem 3. *Let N be a labelled Petri net, and M_1, M_2 a pair of markings in N and ϕ a μ-calculus formula. There is a polynomial time reduction producing an unlabelled and normed Petri net N', a pair of markings $\psi(M_1), \psi(M_2)$ in N' and a μ-calculus formula $\widehat{\phi}$ such that*

$$M_1 \sim_{T(N)} M_2 \quad \textit{if and only if} \quad \psi(M_1) \sim_{T(N')} \psi(M_2)$$

and

$$T(N), M_1 \models \phi \quad \textit{if and only if} \quad T(N'), \psi(M_1) \models \widehat{\phi}.$$

Proof. By Lemma 3 and Theorems 1 and 2. Normedness is by Remark 3. □

Since the bisimilarity checking problem and model checking problems with EF-logic and EG-logic are undecidable [Jan95,Esp97,EK95] for labelled Petri nets, we obtain the following undecidability results for unlabelled and normed Petri nets. In the case of model checking problems we use Remark 4 and the fact that undecidability of model checking with EG-logic can be proved by standard "weak" simulation of a 2-counter machine and we can easily ensure the validity of condition (3) for the Petri net simulating the 2-counter machine.

Corollary 2. *Bisimilarity checking problem for unlabelled and normed Petri nets is undecidable.*

Corollary 3. *Model checking problems with EF-logic and EG-logic for unlabelled and normed Petri nets are undecidable.*

Since the bisimilarity checking problem between a labelled Petri net and a finite-state system is EXPSPACE-hard (see comments e.g. in [May00]), we get also the following corollary.

Corollary 4. *Bisimilarity checking problem between an unlabelled and normed Petri net and a finite-state system is EXPSPACE-hard.*

4.2 Pushdown Systems

It is known that the bisimilarity checking problem for pushdown processes is decidable [Sén98] and PSPACE-hard [May00]. PSPACE-hard is also the bisimilarity checking problem between a pushdown process and a finite-state system [May00] — this problem is moreover in EXPTIME [JKM98].

Model checking pushdown processes with modal μ-calculus is decidable and EXPTIME-complete [Wal96]. This means that the model checking problem with EF-logic, EG-logic and CTL is also in EXPTIME. The model checking problems with these logics are PSPACE-hard — see e.g. [May98]. Moreover, model checking with EF-logic and CTL is known ([Wal00]) to be PSPACE-complete and EXPTIME-complete, respectively. The exact complexity of model checking with EG-logic is unknown, however, it seems to be EXPTIME-complete by modification of arguments from [Wal00].

Definition 9 (Pushdown system). *A* pushdown system Δ *is a tuple* $\Delta = (Q, \Gamma, \mathcal{A}ct, \longrightarrow_\Delta)$ *where* Q *is a finite set of* control states*,* Γ *is a finite* stack alphabet *such that* $Q \cap \Gamma = \emptyset$*,* $\mathcal{A}ct$ *is a finite* input alphabet*, and* $\longrightarrow_\Delta \subseteq Q \times \Gamma \times \mathcal{A}ct \times Q \times \Gamma^*$ *is a finite* $(|\longrightarrow_\Delta| < \infty)$ transition relation*, written* $pA \xrightarrow{a}_\Delta q\alpha$ *for* $(p, A, a, q, \alpha) \in \longrightarrow_\Delta$.

Definition 10 (Labelled transition system $T(\Delta)$).
Let $\Delta = (Q, \Gamma, \mathcal{A}ct, \longrightarrow_\Delta)$ *be a pushdown system. We define a corresponding labelled transition system* $T(\Delta)$ *as* $T(\Delta) = (S, \mathcal{A}ct, \longrightarrow)$ *where* $S = \{p\beta \mid p \in Q \ \wedge \ \beta \in \Gamma^*\}$ *and* $p\beta \xrightarrow{a} q\gamma$ *iff* $\beta = A\beta'$*,* $\gamma = \alpha\beta'$ *and* $pA \xrightarrow{a}_\Delta q\alpha$.

Our aim is to transform Δ into an unlabelled pushdown system such that bisimilarity and model checking are preserved. For technical convenience, we assume from now on that Γ contains a distinct "dummy" symbol Z such that $pZ \not\longrightarrow$ for any $p \in Q$. Then trivially

$$p_1\beta_1 \sim_{T(\Delta)} p_2\beta_2 \quad \text{if and only if} \quad p_1\beta_1 Z \sim_{T(\Delta)} p_2\beta_2 Z \tag{4}$$

$$T(\Delta), p_1\beta_1 \models \phi \quad \text{if and only if} \quad T(\Delta), p_1\beta_1 Z \models \phi \tag{5}$$

for any $p_1, p_2 \in Q$, $\beta_1, \beta_2 \in \Gamma^*$ and a μ-calculus formula ϕ. In particular, all reachable states from $p\beta Z$ are of the form $q\beta' Z$ where $p, q \in Q$ and $\beta, \beta' \in \Gamma^*$.

Definition 11 (Unlabelled pushdown system). *An* unlabelled pushdown system *is a pushdown system* $\Delta = (Q, \Gamma, \mathcal{A}ct, \longrightarrow_\Delta)$ *such that* $|\mathcal{A}ct| = 1$.

Remark 6. Whenever $|\mathcal{A}ct| = 1$, let us say $\mathcal{A}ct = \{a\}$, we omit $\mathcal{A}ct$ from the definition of the pushdown system Δ and instead of $pA \xrightarrow{a}_\Delta q\alpha$ we simply write $pA \longrightarrow_{\Delta'} q\alpha$ where $\Delta' = (Q, \Gamma, \longrightarrow_{\Delta'})$ and $\longrightarrow_{\Delta'} \subseteq Q \times \Gamma \times Q \times \Gamma^*$.

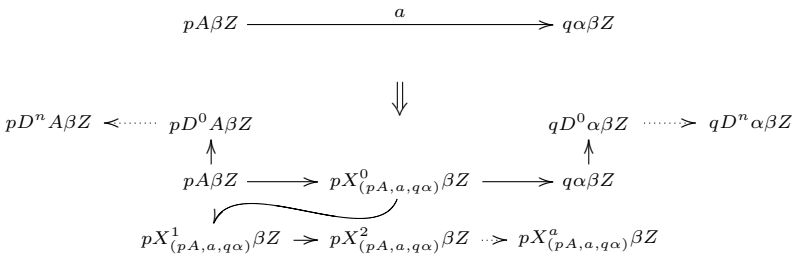


Fig. 3. Transformation of a transition $pA\beta Z \xrightarrow{a} q\alpha\beta Z$

Let $\Delta = (Q, \Gamma, \mathcal{A}ct, \longrightarrow_\Delta)$ be a pushdown system such that $Z \in \Gamma$ is the "dummy" stack symbol. W.l.o.g. assume that $\mathcal{A}ct = \{1, \ldots, n\}$ for some $n > 0$. We construct an unlabelled pushdown system $\Delta' = (Q, \Gamma', \longrightarrow_{\Delta'})$ where $\Gamma \subseteq \Gamma'$ such that $\widehat{T(\Delta)_{p_1\alpha_1 Z}}$ and $T(\Delta')_{p_1\alpha_1 Z}$ are isomorphic unlabelled transition systems for any $p_1 \in Q$ and $\alpha_1 \in \Gamma^*$. Again, see Definition 2 for the notation of transition systems restricted to reachable states from $p_1\alpha_1 Z$. The system Δ' is defined as follows:

$$\begin{aligned}
\Gamma' \quad &= \Gamma \cup \{X^k_{(pA,a,q\alpha)} \mid pA \xrightarrow{a}_\Delta q\alpha \ \wedge \ 0 \le k \le a\} \cup \{D^k \mid 0 \le k \le n\} \\
\longrightarrow_{\Delta'} &= \{(p, A, p, X^0_{(pA,a,q\alpha)}),\ (p, X^0_{(pA,a,q\alpha)}, q, \alpha) \mid pA \xrightarrow{a}_\Delta q\alpha\} \cup \\
&\quad \{(p, X^k_{(pA,a,q\alpha)}, p, X^{k+1}_{(pA,a,q\alpha)}) \mid pA \xrightarrow{a}_\Delta q\alpha \ \wedge \ 0 \le k < a\} \cup \\
&\quad \{(p, A, p, D^0 A) \mid p \in Q \ \wedge \ A \in \Gamma\} \cup \\
&\quad \{(p, D^k, p, D^{k+1}) \mid p \in Q \ \wedge \ 0 \le k < n\}.
\end{aligned}$$

Notice that in particular $pX^a_{(pA,a,q\alpha)}\beta Z \not\longrightarrow$ and $pD^n\beta Z \not\longrightarrow$ for any $\beta \in \Gamma'^*$. Graphical representation showing the transformation of $pA\beta Z \xrightarrow{a} q\alpha\beta Z$ where $\beta \in \Gamma^*$ and $pA \xrightarrow{a}_\Delta q\alpha$ can be seen in Figure 3.

Lemma 4. *Let $\Delta = (Q, \Gamma, \mathcal{A}ct, \longrightarrow_\Delta)$ be a pushdown system containing $Z \in \Gamma$. Let $\Delta' = (Q, \Gamma', \longrightarrow_{\Delta'})$ be the unlabelled pushdown system defined above. Then $\widehat{T(\Delta)_{p_1\alpha_1 Z}}$ and $T(\Delta')_{p_1\alpha_1 Z}$ are isomorphic unlabelled transition systems for any $p_1 \in Q$ and $\alpha_1 \in \Gamma^*$.*

Proof. Immediately from the construction. Notice that it is important that any reachable state in $T(\Delta')_{p_1\alpha_1 Z}$ ends with Z. In particular, from any state of the form $p\beta Z$ where $p \in Q$ and $\beta \in \Gamma^*$ (even if $\beta = \epsilon$) the following transition is possible in $T(\Delta')$: $p\beta Z \longrightarrow pD^0\beta Z$. □

Theorem 4. *Let Δ be a pushdown system, and $p_1\beta_1, p_2\beta_2$ a pair of states in $T(\Delta)$ and ϕ a μ-calculus formula. There is a polynomial time reduction producing an unlabelled and normed pushdown system Δ', a pair of states $\psi(p_1\beta_1), \psi(p_2\beta_2)$ in $T(\Delta')$ and a μ-calculus formula $\widehat{\phi}$ such that*

$$p_1\beta_1 \sim_{T(\Delta)} p_2\beta_2 \quad \textit{if and only if} \quad \psi(p_1\beta_1) \sim_{T(\Delta')} \psi(p_2\beta_2)$$

and

$$T(\Delta), p_1\beta_1 \models \phi \quad \textit{if and only if} \quad T(\Delta'), \psi(p_1\beta_1) \models \widehat{\phi}.$$

Proof. Directly from Lemma 4 together with (4) and (5) — producing the mapping ψ such that $\psi(p\beta) = p\beta Z$ for $p \in Q$ and $\beta \in \Gamma^*$ — and from Theorems 1 and 2. Normedness is because of Remark 3. □

Since the bisimilarity checking problem between a pushdown system and a finite-state system is PSPACE-hard [May00] (this is trivially also a lower bound for two pushdown systems), and because the model checking problems with CTL and Hennessy-Milner logic are EXPTIME-complete resp. PSPACE-complete [Wal00, May98], we obtain the following corollaries. In the case of CTL we use Remark 4 and the fact that we can easily ensure the validity of condition (3) similarly as in the proof of Corollary 1.

Corollary 5. *Bisimilarity checking problem between an unlabelled and normed pushdown system and a finite-state system (or another unlabelled and normed pushdown system) is PSPACE-hard.*

Corollary 6. *Model checking problems with CTL and Hennessy-Milner logic for unlabelled and normed pushdown systems are EXPTIME-complete and PSPACE-complete, respectively.*

The bisimilarity checking problem between a pushdown system and a finite-state system is in EXPTIME [JKM98] and PSPACE-hard [May00]. In order to establish its containment in e.g. PSPACE, it is enough to show it for unlabelled and normed pushdown systems.

Acknowledgements: I would like to thank Mogens Nielsen for his kind supervision and Daniel Polansky for his comments and suggestions.

References

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983.

[BE97] O. Burkart and J. Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science*, 62:138–159, June 1997. Columns: Concurrency.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[EK95] J. Esparza and A. Kiehn. On the model checking problem for branching time logics and basic parallel processes. In *International Conference on Computer-Aided Verification (CAV'95)*, volume 939 of *LNCS*, pages 353–366, 1995.

[Esp97] J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985.

[Jan95] P. Jancar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995.

[JKM98] P. Jancar, A. Kucera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. In *Proceedings of the Annual International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*. Springer-Verlag, 1998.

[JM95] P. Jancar and F. Moller. Checking regular properties of Petri nets. In *Proceedings of CONCUR'95*, volume 962 of *LNCS*, pages 348–362. Springer-Verlag, 1995.

[May98] R. Mayr. Strict lower bounds for model checking BPA. In *Proceedings of the MFCS'98 Workshop on Concurrency*, volume 18 of *ENTCS*. Springer-Verlag, 1998.

[May00] R. Mayr. On the complexity of bisimulation problems for pushdown automata. In *IFIP International Conference on Theoretical Computer Science (IFIP TCS'2000)*, volume 1872 of *LNCS*. Springer-Verlag, 2000.

[Mol96] F. Moller. Infinite results. In *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 195–216. Springer-Verlag, 1996.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.

[Sén98] G. Sénizergues. Decidability of bisimulation equivalence for equational graphs of finite out-degree. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science(FOCS-98)*, pages 120–129. IEEE Computer Society, 1998.

[Srb01] J. Srba. On the power of labels in transition systems. Technical Report RS-01-19, BRICS Research Series, 2001.

[Sti95] C. Stirling. Local model checking games. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.

[Tho93] W. Thomas. On the Ehrenfeucht-Fraïssé game in theoretical computer science (extended abstract). In *Proceedings of the 4th International Joint Conference CAAP/FASE, Theory and Practice of Software Development (TAPSOFT'93)*, volume 668 of *LNCS*, pages 559–568. Springer-Verlag, 1993.

[Vog92] W. Vogler. *Modular construction and partial order semantics of Petri nets*, volume 625 of *LNCS*. Springer-Verlag, 1992.

[Wal96] I. Walukiewicz. Pushdown processes: Games and model checking. In *International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 62–74, 1996. To appear in Information and Computation.

[Wal00] I. Walukiewicz. Model checking CTL properties of pushdown systems. In *Proceedings Foundations of Software Technology and Theoretical Computer Science (FSTTCS'00)*, volume 1974 of *LNCS*, pages 127–138. Springer-Verlag, 2000.

On Barbed Equivalences in π-Calculus

Davide Sangiorgi[1] and David Walker[2]

[1] INRIA Sophia-Antipolis, France davide.sangiorgi@inria.fr
[2] Oxford University Computing Laboratory, U.K. walker@comlab.ox.ac.uk

Abstract. This paper presents some new results on barbed equivalences for the π-calculus. The equivalences studied are barbed congruence and a variant of it called open barbed bisimilarity. The difference between the two is that in open barbed the quantification over contexts is inside the definition of the bisimulation and is therefore recursive. It is shown that if infinite sums are admitted to the π-calculus then it is possible to give a simple proof that barbed congruence and early congruence coincide on all processes, not just on image-finite processes. It is also shown that on the π-calculus, and on the extension of it with infinite sums, open barbed bisimilarity does not correspond to any known labelled bisimilarity. It coincides with a variant of open bisimilarity in which names that have been extruded are treated in a special way, similarly to how names are treated in early bisimilarity.

1 Introduction

This paper presents some new results on barbed equivalences for the π-calculus. The equivalences studied are *barbed congruence* [9] and a variant of it [5] here called *open barbed bisimilarity*. Both equivalences are obtained via kinds of bisimulation that use reduction and a notion of observation, and both are contextual in the sense that their definitions involve quantification over contexts. Most importantly, both relations are congruences, and therefore allow compositional reasoning about processes. The difference between the two is that in open barbed the quantification over contexts is inside the definition of the bisimulation and is therefore recursive.

Equivalences whose definitions involve just notions of reduction and observation and quantification over contexts are useful because they can be applied to a wide variety of calculi and languages. This is important because it is sometimes far from clear how to define appropriate equivalences by other means, for instance based on the actions that processes can perform according to some labelled transition rules. The main difficulty with definitions that involve quantification over contexts is that they are often awkward to work with directly. It is therefore important to look for more tractable characterizations of the equivalences.

The aims of the paper are to contrast the two barbed equivalences and to show that a simple proof of a characterization theorem for barbed congruence can be given that with the addition of infinite sums applies to all processes.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 292–304, 2001.
© Springer-Verlag Berlin Heidelberg 2001

State of the art. Barbed congruence has been used on a variety of concurrent calculi and languages (imperative, object-oriented, functional, etc.). On languages akin to the π-calculus, a number of results that characterize barbed congruence in terms of early congruence have been shown; examples are [3,7, 13]. Typically, these results are for image-finite processes only. The only results we are aware of that do not assume image-finiteness are in [11] and [4]. Their proofs are complex, however, and they cover only specific dialects of the π-calculus. In [11] the π-calculus is extended with infinite sums and with infinitely many mutually recursive definitions, and moreover each recursive definition can have an infinite number of name parameters. In [4] the calculus is a form of asynchronous π-calculus.

Open barbed bisimilarity has been applied to several variants of the π-calculus and to higher-order calculi; examples are [5,6,1]. The main advantage of open barbed bisimilarity over barbed congruence is that a characterization result can often be proved, using an appropriate form of labelled bisimilarity, that applies to *all* processes, not just image-finite processes. We are not aware of any characterization result for open barbed bisimilarity on the standard π-calculus, however. Such a result would be useful, beyond its significance for the π-calculus, because it would shed light on the robustness of open barbed bisimilarity, and in particular on the relationship between it and barbed congruence.

Contributions. The main contributions of this paper are as follows.

1. We show that if we admit infinite sums to the π-calculus, then we can give a simple proof that barbed congruence and early congruence coincide on all processes, not just on image-finite processes. (It remains an open question whether the equivalences coincide on the π-calculus.)
2. We show that, perhaps surprisingly, on the π-calculus (and on the extension of it with infinite sums) open barbed bisimilarity does not correspond to any known labelled bisimilarity. It coincides with a variant of *open bisimilarity* [12] in which names that have been extruded are treated in a special way, similarly to how names are treated in early bisimilarity.

The structure of the proof of (1) is similar to that in [13] for image-finite processes. As a corollary of (2), open barbed bisimilarity is different from barbed congruence on the π-calculus. Compared to [11], the result (1), besides having a much simpler proof, does not use infinitely many mutually recursive definitions. This is significant because infinitely many mutually recursive definitions cannot be encoded using replication; only finitely many mutually recursive definitions, each with a finite number of name parameters, can be so encoded; see [8].

The results in this paper do not tell us whether the new labelled bisimilarity introduced in (2) is interesting. Its mixture of features from open bisimilarity and early bisimilarity seems hard to justify observationally, however.

This paper treats the weak equivalences, which abstract from internal action. Analogous results for *strong* equivalences are easily obtained by simplifying the proofs. We believe that the results in the paper hold also for other dialects of the π-calculus that have the matching operator (which is fundamental to

the proofs of the characterization results), for instance the Asynchronous π-calculus. The calculus we work with in this paper has guarded sums. If we were to admit unguarded sums, as in CCS, then the clause for τ transitions in the characterization of open barbed bisimilarity would have to be modified along the lines of *dynamic bisimilarity* [10].

Other related work. A distinctive feature of open barbed bisimilarity is that it is both a bisimulation and a congruence. Montanari and Sassone [10] first used relations that by definition are both bisimulations and congruences, in CCS. These relations are defined like CCS bisimulation, but with an additional requirement similar to closure of an open barbed bisimulation under contexts; see Definition 2(4). On the π-calculus, the largest relation so obtained is different from (early) bisimilarity because bisimilarity is not a congruence. Honda and Yoshida [5] applied similar ideas in the setting of reduction-based bisimilarities, where τ transitions, or reductions, are the only relevant transitions of processes. The relation defined is essentially what we call *open barbed bisimilarity*, although the formulation and the calculi are rather different. In particular, a significant difference is that [5] distinguishes between names and variables, whereas this paper is about the π-calculus where there are only names.

2 Background

In this section we briefly recall some definitions and notations for the π-calculus.

We assume a countably infinite set N of *names*, ranged over by lower-case letters, $x, y, z, \ldots$. The *prefixes* are given by

$$\pi \;::=\; \overline{x}y \;\mid\; x(z) \;\mid\; \tau \;\mid\; [x=y]\pi \;.$$

Note that we take matches to be parts of prefixes. The *processes* and the *summations* of the π-calculus are given respectively by

$$\begin{array}{rcl} P & ::= & M \mid P \mid P' \mid \boldsymbol{\nu} z\, P \mid\; !P \\ M & ::= & \mathbf{0} \mid \pi.\, P \mid M + M' \;. \end{array}$$

We abbreviate $x(z).\,\mathbf{0}$ to x. A *context* is obtained when in some P given by the grammar above, the *hole* $[\cdot]$ replaces an occurrence of $\mathbf{0}$ that is not the left or right term in a sum $M + M'$. A *non-input context* is a context in which the hole does not occur underneath an input prefix.

In each of $x(z).\,P$ and $\boldsymbol{\nu} z\, P$, the displayed occurrence of z is binding with scope P. We write $\mathsf{fn}(E_1, E_2, \ldots)$ for the set of names that occur free in at least one of the entities $E_1, E_2, \ldots$. A *substitution* is a function σ on the set of names that has finite support, that is, is such that $\{x \mid x\sigma \neq x\}$ is finite. We write $E\sigma$ for the application of a substitution σ to an entity E.

The *actions* are given by

$$\alpha \;::=\; \overline{x}y \;\mid\; xy \;\mid\; \overline{x}(z) \;\mid\; \tau \;.$$

We write Act for the set of actions. We write $\mathsf{bn}(\alpha)$ for the set of names bound in α, which is $\{z\}$ if α is $\overline{x}(z)$ and $\emptyset$ otherwise, and $\mathsf{n}(\alpha)$ for the set of names that occur in α. The (*early*) *transition relations*, $\{\xrightarrow{\alpha} \mid \alpha \in Act\}$, are defined by the rules in Table 1. Elided from the table are four rules: the symmetric form SUM-R of SUM-L, which has $Q + P$ in place of $P + Q$, and the symmetric forms PAR-R, COMM-R, and CLOSE-R of PAR-L, COMM-L, and CLOSE-L, in which the roles of the left and right components are swapped. We adopt rules REP-ACT, REP-COMM, and REP-CLOSE so that the transition relations are image-finite. Other rules could be used, however [8]. We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\stackrel{\alpha}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ for $\alpha \in Act$. Further, $\stackrel{\widehat{\tau}}{\Longrightarrow}$ is $\Longrightarrow$, and $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ is $\stackrel{\alpha}{\Longrightarrow}$ for $\alpha \neq \tau$. We write $P \downarrow_x$ if P can perform an action of the form xy, and $P \downarrow_{\overline{x}}$ if P can perform an action of the form $\overline{x}y$ or $\overline{x}(z)$. Further, for μ a name x or a co-name $\overline{x}$, we write $P \Downarrow_\mu$ if there is Q such that $P \Longrightarrow Q$ and $Q \downarrow_\mu$.

We identify processes that are α-convertible. Moreover, in any discussion, we assume that the bound names of any processes or actions under consideration are chosen to be different from the names free in any other entities under consideration, such as processes, actions, substitutions, and sets of names. This convention is subject to the limitation that in considering a transition $P \xrightarrow{\overline{x}(z)} Q$, the name z that is bound in $\overline{x}(z)$ and in P can occur free in Q.

Definition 1 (Barbed bisimilarity and congruence).
A relation $\mathcal{S}$ is a *barbed bisimulation* if whenever $(P, Q) \in \mathcal{S}$,

1. $P \downarrow_\mu$ implies $Q \Downarrow_\mu$
2. $P \xrightarrow{\tau} P'$ implies $Q \Longrightarrow Q'$ for some Q' with $(P', Q') \in \mathcal{S}$
3. the variants of (1) and (2) with the roles of P and Q swapped.

P and Q are *barbed bisimilar*, $P \dot{\approx} Q$, if $(P, Q) \in \mathcal{S}$ for some barbed bisimulation $\mathcal{S}$. P and Q are *barbed congruent*, $P \cong^{\mathrm{c}} Q$, if $C[P] \dot{\approx} C[Q]$ for every context C. □

For two processes to be barbed congruent, the systems obtained by placing them into an arbitrary context must be barbed bisimilar. In open barbed bisimilarity, on the other hand, the context enclosing the processes being tested can be changed at any point during the bisimulation game: an open barbed bisimulation is required to be closed under contexts.

Definition 2 (Open barbed bisimilarity). A relation $\mathcal{S}$ is an *open barbed bisimulation* if whenever $(P, Q) \in \mathcal{S}$,

1. $P \downarrow_\mu$ implies $Q \Downarrow_\mu$
2. $P \xrightarrow{\tau} P'$ implies $Q \Longrightarrow Q'$ for some Q' with $(P', Q') \in \mathcal{S}$
3. the variants of (1) and (2) with the roles of P and Q swapped
4. $(C[P], C[Q]) \in \mathcal{S}$ for every context C.

P and Q are *open barbed bisimilar*, $P \dot{\approx}_{\mathrm{o}} Q$, if $(P, Q) \in \mathcal{S}$ for some open barbed bisimulation $\mathcal{S}$. □

Table 1. The transition rules

$$\textsc{Out}\ \frac{}{\overline{x}y.\,P \xrightarrow{\overline{x}y} P} \qquad \textsc{Inp}\ \frac{}{x(z).\,P \xrightarrow{xy} P\{y/z\}}$$

$$\textsc{Tau}\ \frac{}{\tau.\,P \xrightarrow{\tau} P} \qquad \textsc{Mat}\ \frac{\pi.\,P \xrightarrow{\alpha} P'}{[x=x]\pi.\,P \xrightarrow{\alpha} P'}$$

$$\textsc{Sum-l}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\textsc{Par-l}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$$

$$\textsc{Comm-l}\ \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Close-l}\ \frac{P \xrightarrow{\overline{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu} z\,(P' \mid Q')}\ z \notin \mathsf{fn}(Q)$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu} z\,P \xrightarrow{\alpha} \boldsymbol{\nu} z\,P'}\ z \notin \mathsf{n}(\alpha) \qquad \textsc{Open}\ \frac{P \xrightarrow{\overline{x}z} P'}{\boldsymbol{\nu} z\,P \xrightarrow{\overline{x}(z)} P'}\ z \neq x$$

$$\textsc{Rep-act}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\textsc{Rep-comm}\ \frac{P \xrightarrow{\overline{x}y} P' \quad P \xrightarrow{xy} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

$$\textsc{Rep-close}\ \frac{P \xrightarrow{\overline{x}(z)} P' \quad P \xrightarrow{xz} P''}{!P \xrightarrow{\tau} (\boldsymbol{\nu} z\,(P' \mid P'')) \mid !P}\ z \notin \mathsf{fn}(P)$$

Definition 3 (Early bisimilarity and congruence).
A relation $\mathcal{S}$ is an *early bisimulation* if whenever $(P, Q) \in \mathcal{S}$,

1. $P \xrightarrow{\alpha} P'$ implies $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ for some Q' with $(P', Q') \in \mathcal{S}$
2. the variant of (1) with the roles of P and Q swapped.

P and Q are *early bisimilar*, $P \approx Q$, if $(P, Q) \in \mathcal{S}$ for some early bisimulation $\mathcal{S}$. P and Q are *early congruent*, $P \approx^{\mathrm{c}} Q$, if $P\sigma \approx Q\sigma$ for every substitution σ. □

A process P is *image-finite* if for each derivative Q of P and each action α, there are $n \geq 0$ and $Q_1, \ldots, Q_n$ such that $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ implies $Q' = Q_i$ for some i. We recall the following result for the π-calculus; see [3,13] for proofs of closely related results.

Theorem 1 (Characterization Theorem). Suppose that P and Q are image-finite. Then $P \cong^{c} Q$ iff $P \approx^{c} Q$. □

3 Infinite Sums

In this section we show that if we admit sums in which the number of summands is infinite, then Theorem 1 can be extended to all processes of the enriched language, that is, the assumption of image-finiteness can be removed.

The only caveat for allowing infinite sums is that the possibility of α-converting names into fresh names must be maintained. To ensure this we assume that the set of names has cardinality ω_1, the first uncountable cardinal, and require sums to have only countably many summands. (Similar results can be obtained when ω_1 is replaced by an arbitrary uncountable regular cardinal κ and sums can have only fewer than κ summands.) Accordingly, we replace the form $M + M'$ in the grammar for summations by

$$\Sigma_{i \in I} M_i$$

where I is a countable set. We write $\pi_{\infty+}$ for the resulting calculus. The modifications required to the transition rules are straightforward.

Lemma 1. The set of processes of $\pi_{\infty+}$ has cardinality ω_1.

Proof This is an instance of a standard result about inductively defined sets. See for instance Section 1.3 of [2]. □

Lemma 2. For any process P of $\pi_{\infty+}$ and action α, the set $\{P' \mid P \stackrel{\alpha}{\Longrightarrow} P'\}$ is countable.

Proof The proof uses repeatedly the fact that a countable union of countable sets is countable. First one shows that for any P and α, $\{P' \mid P \stackrel{\alpha}{\longrightarrow} P'\}$ is countable. Then one shows by induction that for $n < \omega$, $\{P' \mid P \, (\stackrel{\tau}{\longrightarrow})^n \, P'\}$ is countable. From this it follows that $\{P' \mid P \Longrightarrow P'\}$ is countable, and hence that $\{P' \mid P \stackrel{\alpha}{\Longrightarrow} P'\}$ is countable. □

Definition 4 (Transfinite stratification of bisimilarity).

1. $\approx_0$ is the universal relation on $\pi_{\infty+}$.
2. For δ an ordinal, the relation $\approx_{\delta+1}$ is defined by: $P \approx_{\delta+1} Q$ if
   a) $P \stackrel{\alpha}{\longrightarrow} P'$ implies $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} \approx_{\delta} P'$

b) $Q \xrightarrow{\alpha} Q'$ implies $P \stackrel{\widehat{\alpha}}{\Longrightarrow} \approx_\delta Q'$.

3. For γ a limit ordinal, $P \approx_\gamma Q$ if $P \approx_\delta Q$ for all $\delta < \gamma$. □

Lemma 3. On $\pi_{\infty+}$, $P \approx Q$ iff $P \approx_\delta Q$ for every ordinal δ.

Proof First, by induction on δ we have that $P \approx Q$ implies $P \approx_\delta Q$ for every ordinal δ. For the converse first note that if $\delta < \delta'$ then $\approx_{\delta'}$ is included in $\approx_\delta$. Since by Lemma 1 the cardinality of the set of processes of $\pi_{\infty+}$ is ω_1, the universal relation $\approx_0$ has cardinality ω_1. Hence there is δ of cardinality ω_1 such that $\approx_{\delta+1}$ coincides with $\approx_\delta$. This is so because otherwise $\approx_{\delta+1}$ would be strictly included in $\approx_\delta$ for each $\delta < \omega_2$ (the second uncountable cardinal), contradicting that $\approx_0$ has cardinality ω_1. Suppose δ is the smallest ordinal such that $\approx_{\delta+1}$ coincides with $\approx_\delta$. Then $\approx_\delta$ is an early bisimulation, so $P \approx_\delta P'$ implies $P \approx P'$, and $\approx_{\delta'}$ coincides with $\approx_\delta$ for all $\delta' > \delta$. □

Theorem 2 (Characterization Theorem on $\pi_{\infty+}$). Suppose P and Q are processes of $\pi_{\infty+}$. Then $P \cong^{\mathrm{c}} Q$ iff $P \approx^{\mathrm{c}} Q$.

Proof We recall that processes P and Q are *barbed equivalent*, $P \cong Q$, if $C[P] \stackrel{\cdot}{\approx} C[Q]$ for every context C of the form $[\cdot] \mid R$. We show that $P \cong Q$ iff $P \approx Q$. The conclusion $P \cong^{\mathrm{c}} Q$ iff $P \approx^{\mathrm{c}} Q$ follows since $\cong^{\mathrm{c}}$ is the largest congruence included in $\cong$ and $\approx^{\mathrm{c}}$ is the largest congruence included in $\approx$.

That $P \approx Q$ implies $P \cong Q$ follows from the facts that $\approx$ is a barbed bisimulation and a non-input congruence (that is, is preserved by all non-input contexts), and that $\cong$ is the largest non-input congruence included in $\stackrel{\cdot}{\approx}$.

The main claim needed to show that $P \cong Q$ implies $P \approx Q$ is the following.

Claim. Suppose that $P \not\approx_\delta Q$ where δ is an ordinal. Then there is a summation M such that for any $\widetilde{z} \subseteq \mathsf{fn}(P, Q)$ and any fresh name s, one of the following holds:

1. $(\boldsymbol{\nu}\widetilde{z})(P' \mid (M+s)) \not\stackrel{\cdot}{\approx} (\boldsymbol{\nu}\widetilde{z})(Q \mid (M+s))$ for all P' such that $P \Longrightarrow P'$
2. $(\boldsymbol{\nu}\widetilde{z})\ (P \mid (M+s)) \not\stackrel{\cdot}{\approx} (\boldsymbol{\nu}\widetilde{z})\ (Q' \mid (M+s))$ for all Q' such that $Q \Longrightarrow Q'$.

Proof If δ is a limit ordinal and $P \not\approx_\delta Q$, then $P \not\approx_\gamma Q$ for some $\gamma < \delta$, and the result follows immediately by induction.

Suppose $\delta = \gamma + 1$. Then there are α and P' such that $P \xrightarrow{\alpha} P'$ but $P' \not\approx_\gamma Q'$ for all Q' such that $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ (or vice versa, when the argument is the same). By Lemma 2, $\{Q' \mid Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'\} = \{Q_i \mid i \in I\}$ for some countable set I. We prove that assertion (2) of the Claim holds (in the case when the roles of P and Q are swapped, one would prove assertion (1)).

Appealing to the induction hypothesis, for each $i \in I$ let M_i be a summation such that either (1) or (2) of the Claim holds for P', Q_i, and M_i. There are four cases, one for each form that α can take. We give the details only for the case when α is an input action. The other cases are similar, and for them we just indicate the main point in the construction.

Case 1 Suppose that α is xy. Let s_i $(i \in I)$ and s' be fresh names, and set

$$M \stackrel{\text{def}}{=} \overline{x}y.\,(s' + \Sigma_{i\in I}\,\tau.\,(M_i + s_i))\ .$$

Suppose that $\widetilde{z} \subseteq \mathsf{fn}(P,Q)$ and s is fresh, and let Q' be any process such that $Q \Longrightarrow Q'$. Let $A \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})(P \mid (M+s))$ and $B \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})(Q' \mid (M+s))$, and suppose, for a contradiction, that $A \dot{\approx} B$. We have

$$A \stackrel{\tau}{\longrightarrow} A' \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})\,(P' \mid (s' + \Sigma_{i\in I}\,\tau.\,(M_i + s_i)))$$

and $A' \Downarrow_{s'}$ but not $A' \Downarrow_s$. Since $A \dot{\approx} B$ there is B' such that $B \Longrightarrow B' \dot{\approx} A'$. In particular it must be that $B' \Downarrow_{s'}$ but not $B' \Downarrow_s$. The only way this is possible is if $I \neq \emptyset$ and

$$B' \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})\,(Q_j \mid (s' + \Sigma_{i\in I}\,\tau.\,(M_i + s_i)))$$

for some $j \in I$ (a derivative of Q' under $\stackrel{\alpha}{\Longrightarrow}$ is also a derivative of Q). Now either (1) or (2) of the Claim holds for P', Q_j, and M_j. Suppose that (2) holds. We have

$$A' \stackrel{\tau}{\longrightarrow} A''_j \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})\,(P' \mid (M_j + s_j))$$

and $A''_j \Downarrow_{s_j}$ but not $A''_j \Downarrow_{s'}$. Then $B' \Longrightarrow B''_j$ with $B''_j \dot{\approx} A''_j$, and we must have

$$B''_j \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})\,(Q'_j \mid (M_j + s_j))$$

for some Q'_j such that $Q_j \Longrightarrow Q'_j$. But $A''_j \dot{\approx} B''_j$ contradicts that (2) of the Claim holds for P', Q_j, and M_j.
Dually, if (1) of the Claim holds for P', Q_j, and M_j, then we obtain a contradiction by considering how A' can match the transition

$$B' \stackrel{\tau}{\longrightarrow} B''_j \stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{z})\,(Q_j \mid (M_j + s_j))\ .$$

Case 2 Suppose that α is $\overline{x}y$. Let s_i $(i \in I)$ and s' and w be fresh names, and set

$$M \stackrel{\text{def}}{=} x(w).\,(s' + \Sigma_{i\in I}\,[w=y]\tau.\,(M_i + s_i))\ .$$

Case 3 Suppose that α is $\overline{x}(z)$. Suppose that $\mathsf{fn}(P,Q) = \{a_1,\ldots,a_k\}$. Let s_i $(i \in I)$ and t and s' and w be fresh names, and set

$$M \stackrel{\text{def}}{=} x(w).\,(s' + \Sigma_{h=1}^{k}[w=a_h]t + \Sigma_{i\in I}\,\tau.\,(M_i + s_i))\ .$$

Case 4 Suppose that α is τ. Let s_i $(i \in I)$ be fresh names, and set

$$M \stackrel{\text{def}}{=} \Sigma_{i\in I}\,\tau.\,(M_i + s_i)\ .$$

This completes the proof of the Claim. □

To complete the proof of the theorem, suppose that $P \not\approx Q$. Then by Lemma 3, $P \not\approx_\delta Q$ for some δ. Then let M be as given by the Claim for P and Q, let s be fresh, and set $C \stackrel{\text{def}}{=} [\cdot] \mid (M + s)$. Then $C[P] \dot{\not\approx} C[Q]$, and so $P \not\cong Q$. □

4 Characterization of Open Barbed Bisimilarity

In this section we show a characterization theorem for open barbed bisimilarity on the full π-calculus. The theorem holds both for the π-calculus and for its extension with infinite sums, $\pi_{\infty+}$. The labelled bisimilarity that characterizes open barbed bisimilarity is similar to open bisimilarity; indeed on the subcalculus without restriction it coincides with open bisimilarity. On the full calculus, however, the relations treat extruded names differently. To highlight the difference, we recall the definition of open bisimilarity.

Open bisimilarity is usually defined using the late transition relations. One of the significant features of open bisimilarity is that when comparing two processes, it suffices to consider only input actions and bound-output actions whose object is a single fresh name. This can be important in reducing the amount of work needed to determine whether processes are equivalent. Here, however, it is convenient to cast the definition using the early transition relations.

A *distinction* is a finite symmetric and irreflexive relation on names. A substitution σ *respects* a distinction D if $(x, y) \in D$ implies $x\sigma \neq y\sigma$. Given sets of names Y and Z, we write $Y \otimes Z$ for the distinction that contains all pairs (y, z) and (z, y) such that $y \in Y$, $z \in Z$, and $y \neq z$.

Definition 5 (Open bisimilarity). An *open bisimulation* is a family of relations $\{\mathcal{S}_D \mid D \text{ a distinction}\}$ such that whenever $(P, Q) \in \mathcal{S}_D$,

1. if $P \xrightarrow{\alpha} P'$ and α is not a bound output, then $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ for some Q' with $(P', Q') \in \mathcal{S}_D$
2. if $P \xrightarrow{\overline{x}(z)} P'$ then $Q \stackrel{\overline{x}(z)}{\Longrightarrow} Q'$ for some Q' with $(P', Q') \in \mathcal{S}_{D'}$ where $D' = D \cup (\{z\} \otimes \mathsf{fn}(P, Q))$
3. the variants of (1) and (2) with the roles of P and Q swapped
4. $(P\sigma, Q\sigma) \in \mathcal{S}_{D\sigma}$ for every σ that respects D.

We write $\{\approx_{\mathrm{o}}^{D} \mid D \text{ a distinction}\}$ for the pointwise union of all open bisimulations, and refer to $\approx_{\mathrm{o}}^{D}$ as *open D-bisimilarity*. In particular, we write $\approx_{\mathrm{o}}$ for $\approx_{\mathrm{o}}^{\emptyset}$, and refer to it as *open bisimilarity*. □

The definition of quasi-open bisimilarity involves a family of relations indexed by finite sets of names, rather than by distinctions. We need the analogue of the notion that a substitution respects a distinction. Suppose $\widetilde{z}$ is a finite set of names. A substitution σ *respects* $\widetilde{z}$ if whenever $z \in \widetilde{z}$ and $y \neq z$, then $y\sigma \neq z\sigma$.

Definition 6 (Quasi-open bisimilarity). A *quasi-open bisimulation* is a family of relations $\{\mathcal{S}^{\widetilde{z}} \mid \widetilde{z} \subseteq \mathsf{N} \text{ finite}\}$ such that whenever $(P, Q) \in \mathcal{S}^{\widetilde{z}}$,

1. if $P \xrightarrow{\alpha} P'$ and α is not a bound output, then $Q \stackrel{\widehat{\alpha}}{\Longrightarrow} Q'$ for some Q' with $(P', Q') \in \mathcal{S}^{\widetilde{z}}$
2. if $P \xrightarrow{\overline{x}(z)} P'$ then $Q \stackrel{\overline{x}(z)}{\Longrightarrow} Q'$ for some Q' with $(P', Q') \in \mathcal{S}^{\widetilde{z} \cup \{z\}}$
3. the variants of (1) and (2) with the roles of P and Q swapped

4. $(P\sigma, Q\sigma) \in \mathcal{S}^{\widetilde{z}\sigma}$ for every σ that respects $\widetilde{z}$.

We write $\{\approx_{\mathrm{q}}^{\widetilde{z}} \mid \widetilde{z} \subseteq \mathsf{N} \text{ finite}\}$ for the pointwise union of all quasi-open bisimulations, and refer to $\approx_{\mathrm{q}}^{\widetilde{z}}$ as *quasi-open $\widetilde{z}$-bisimilarity*. In particular, we write $\approx_{\mathrm{q}}$ for $\approx_{\mathrm{q}}^{\emptyset}$, and refer to it as *quasi-open bisimilarity*. □

In open bisimilarity, when a name z is sent in a bound-output action, the distinction is enlarged to ensure that z is never identified with any name that is free in the processes that send it. In quasi-open bisimilarity, in contrast, at no point after the scope of z is extruded can a substitution be applied that identifies z with *any* other name. Roughly, quasi-open bisimilarity treats names that are extruded in the same way that early bisimilarity treats all names.

Lemma 4. Quasi-open bisimilarity, $\approx_{\mathrm{q}}$, is a congruence.

Proof (Outline) The proof involves showing that $\approx_{\mathrm{q}}$ is preserved by each operator of the calculus. This is done by exhibiting appropriate quasi-open bisimulations. Due to the special treatment of extruded names, the case of composition needs special care. □

Quasi-open bisimilarity, like open bisimilarity, is more discriminating than early congruence: the union over $\widetilde{z}$ of the relations $\approx_{\mathrm{q}}^{\widetilde{z}}$ is an early bisimulation; and for instance $P \approx^{\mathrm{c}} Q$ but $P \not\approx_{\mathrm{q}} Q$ where

$$P \stackrel{\mathrm{def}}{=} x.\, x + x + x.\, [y = z]x$$
$$Q \stackrel{\mathrm{def}}{=} x.\, x + x\ .$$

The reason why $P \not\approx_{\mathrm{q}} Q$ is that $[y = z]x \not\approx_{\mathrm{q}} x$ and $[y = z]x \not\approx_{\mathrm{q}} \mathbf{0}$, as can be seen by considering the effects of applying the identity substitution and the substitution $\{y/z\}$ respectively.

Moreover, quasi-open bisimilarity is strictly weaker than open bisimilarity.

Lemma 5. $\approx_{\mathrm{o}}$ is strictly included in $\approx_{\mathrm{q}}$.

Proof For $\widetilde{z}$ a finite set of names define $\mathcal{S}^{\widetilde{z}}$ by setting $(P, Q) \in \mathcal{S}^{\widetilde{z}}$ if there is a distinction D such that $P \approx_{\mathrm{o}}^{D} Q$ and for each pair $(x, y) \in D$, either $x \in \widetilde{z}$ or $y \in \widetilde{z}$. Then $\{\mathcal{S}^{\widetilde{z}} \mid \widetilde{z} \subseteq \mathsf{N} \text{ finite}\}$ can be shown to be a quasi-open bisimulation. So in particular, if $P \approx_{\mathrm{o}} Q$ then $(P, Q) \in \mathcal{S}^{\emptyset}$ and so $P \approx_{\mathrm{q}} Q$.

To show that the inclusion is strict we can take

$$P \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} z\, \overline{x}z.\, (x(w) + x(w).\, z + x(w).\, [w = z]z)$$
$$Q \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} z\, \overline{x}z.\, (x(w) + x(w).\, z)\ .$$

Then $P \xrightarrow{\overline{x}(z)} \xrightarrow{xw} P' \stackrel{\mathrm{def}}{=} [w = z]z$ and, where $D = \{(x, z), (z, x)\}$, we have $P' \not\approx_{\mathrm{o}}^{D} \mathbf{0}$ because $\{z/w\}$ respects D, while $P' \not\approx_{\mathrm{o}}^{D} z$ because the identity respects D, so $P \not\approx_{\mathrm{o}} Q$. On the other hand, $P' \approx_{\mathrm{q}}^{\{z\}} \mathbf{0}$ because $\{z/w\}$ does not respect $\{z\}$, and so $P \approx_{\mathrm{q}} Q$. □

To prove the characterization result we need that open barbed bisimilarity is preserved by arbitrary substitution.

Lemma 6. $P \dot{\approx}_{\mathrm{o}} Q$ implies $P\sigma \dot{\approx}_{\mathrm{o}} Q\sigma$, for all σ.

Proof Suppose $\sigma = \{y_1 \cdots y_n / x_1 \ldots x_n\}$ where $\{x_1, \ldots, x_n\}$ is the support of σ. Let a be a fresh name and set

$$P_1 \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} a\, (\overline{a}y_1. \ldots . \overline{a}y_n \mid a(x_1). \ldots . a(x_n).\, P)$$
$$Q_1 \stackrel{\mathrm{def}}{=} \boldsymbol{\nu} a\, (\overline{a}y_1. \ldots . \overline{a}y_n \mid a(x_1). \ldots . a(x_n).\, Q)\ .$$

By clause (4) of Definition 2, $P_1 \dot{\approx}_{\mathrm{o}} Q_1$. Using some simple properties of $\dot{\approx}_{\mathrm{o}}$, we have

$$P_1 \dot{\approx}_{\mathrm{o}} \underbrace{\tau. \ldots . \tau.}_{n} P\sigma \dot{\approx}_{\mathrm{o}} P\sigma\ ,$$

and similarly, $Q_1 \dot{\approx}_{\mathrm{o}} Q\sigma$. Hence by transitivity, $P\sigma \dot{\approx}_{\mathrm{o}} Q\sigma$. □

The following result holds for the π-calculus and for its extension with infinite sums.

Theorem 3 (Characterization Theorem for open barbed bisimilarity).

For any processes P and Q, $P \approx_{\mathrm{q}} Q$ iff $P \dot{\approx}_{\mathrm{o}} Q$.

Proof The implication from left to right holds because $\approx_{\mathrm{q}}$ is an open barbed bisimulation. This follows from the definitions and the fact that $\approx_{\mathrm{q}}$ is a congruence, by Lemma 4.

For the other implication, we define a family of relations $\{\mathcal{R}^{\widetilde{x}} \mid \widetilde{x} \subseteq \mathsf{N} \text{ finite}\}$ such that $\dot{\approx}_{\mathrm{o}}$ is included in $\mathcal{R}^{\emptyset}$, and show that it is a quasi-open bisimulation.

Suppose $\widetilde{x} = \{x_i \mid i \in I\}$. Set $(P, Q) \in \mathcal{R}^{\widetilde{x}}$ if there is $\widetilde{a} = \{a_i \mid i \in I\}$ (with $a_i \neq a_j$ for $i \neq j$) such that $\widetilde{a} \cap \mathsf{fn}(P, Q, \widetilde{x}) = \emptyset$ and

$$\boldsymbol{\nu}\widetilde{x}\, (\Pi_{i\in I} !\overline{a_i}x_i \mid P) \dot{\approx}_{\mathrm{o}} \boldsymbol{\nu}\widetilde{x}\, (\Pi_{i\in I} !\overline{a_i}x_i \mid Q)\ . \tag{1}$$

We consider clauses (4), (1), and (2) in Definition 6; the argument for clause (3) is similar to those for (1) and (2).

Suppose $(P, Q) \in \mathcal{R}^{\widetilde{x}}$. Let $P_1 \stackrel{\mathrm{def}}{=} \boldsymbol{\nu}\widetilde{x}(\Pi_{i\in I} !\overline{a_i}x_i \mid P)$ and $Q_1 \stackrel{\mathrm{def}}{=} \boldsymbol{\nu}\widetilde{x}(\Pi_{i\in I} !\overline{a_i}x_i \mid Q)$ be as in equation (1).

Clause (4) Suppose σ respects $\widetilde{x}$. Choose $\widetilde{b} = \{b_i \mid i \in I\}$ such that $b_i \neq b_j$ for $i \neq j$ and $\widetilde{b} \cap \mathsf{fn}(P, Q, \widetilde{x}, P\sigma, Q\sigma, \widetilde{x}\sigma) = \emptyset$. Let $\theta = \{\widetilde{b}/\widetilde{a}\}$. Then by applying Lemma 6 to equation (1) with the substitution θ,

$$\boldsymbol{\nu}\widetilde{x}\, (\Pi_{i\in I} !\overline{b_i}x_i \mid P) \dot{\approx}_{\mathrm{o}} \boldsymbol{\nu}\widetilde{x}\, (\Pi_{i\in I} !\overline{b_i}x_i \mid Q)\ . \tag{2}$$

Let ρ be the substitution that agrees with σ except that $b_i\rho = b_i$ for each i. Then by applying Lemma 6 to equation (2) with the substitution ρ, where $\widetilde{y} = \widetilde{x}\sigma$ we have

$$\boldsymbol{\nu}\widetilde{y}\, (\Pi_{i\in I} !\overline{b_i}y_i \mid P\sigma) \dot{\approx}_{\mathrm{o}} \boldsymbol{\nu}\widetilde{y}\, (\Pi_{i\in I} !\overline{b_i}y_i \mid Q\sigma)\ .$$

Hence $(P\sigma, Q\sigma) \in \mathcal{R}^{\widetilde{y}}$, as required.

Clause (1) We give just the argument for input actions. Those for free-output and τ actions are of a similar nature.

Suppose $P \xrightarrow{x\,z} P'$. We have to find Q' such that $Q \stackrel{x\,z}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}^{\widetilde{x}}$. There are four cases, determined by whether or not $x \in \widetilde{x}$ and whether or not $z \in \widetilde{x}$. We give just one case. The arguments for the others are variations of it.

Suppose $x \in \widetilde{x}$, say $x = x_i$, and $z \notin \widetilde{x}$. Let s, s', and u be fresh names, and consider the context

$$C \stackrel{\text{def}}{=} \Big(s + a_i(u).(s' + \overline{u}z)\Big) \mid [\cdot]\ .$$

We have

$$C[P_1] \Longrightarrow P_2 \stackrel{\text{def}}{=} \mathbf{0} \mid \boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid P')$$

and it holds that not $P_2 \Downarrow_s$ and not $P_2 \Downarrow_{s'}$. Since $P_1 \mathrel{\dot{\approx}_{\mathrm{o}}} Q_1$, there is Q_2 such that $C[Q_1] \Longrightarrow Q_2 \mathrel{\dot{\approx}_{\mathrm{o}}} P_2$, so in particular it holds that not $Q_2 \Downarrow_s$ and not $Q_2 \Downarrow_{s'}$. The only possibility is that

$$Q_2 = \mathbf{0} \mid \boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid Q')$$

for some Q' such that $Q \stackrel{x\,z}{\Longrightarrow} Q'$. Since $P_2 \mathrel{\dot{\approx}_{\mathrm{o}}} Q_2$ we have

$$\boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid P') \mathrel{\dot{\approx}_{\mathrm{o}}} \boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid Q')$$

and so $(P', Q') \in \mathcal{R}^{\widetilde{x}}$.

Clause (2) Suppose $P \xrightarrow{\overline{x}(z)} P'$, where z is fresh. We have to find Q' such that $Q \stackrel{\overline{x}(z)}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}^{\widetilde{x}\cup\{z\}}$. There are two subcases. We give one; the argument for the other is a variation of it.

Suppose $x \in \widetilde{x}$, say $x = x_i$. Let s, s', t, u, and a be fresh names, and consider the context

$$C \stackrel{\text{def}}{=} \Big(s + a_i(u).(s' + u(z).(\,!\overline{a}z \mid \Pi_{c\in \mathsf{fn}(P,Q)}[z = c]t))\Big) \mid [\cdot]\ .$$

We have

$$C[P_1] \Longrightarrow P_2 \stackrel{\text{def}}{=} \boldsymbol{\nu}z\,\Big(\,!\overline{a}z \mid \Pi_{c\in \mathsf{fn}(P,Q)}[z = c]t \mid \boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid P')\Big)$$

and it holds that not $P_2 \Downarrow_s$ and not $P_2 \Downarrow_{s'}$ and not $P_2 \Downarrow t$. Since $P_1 \mathrel{\dot{\approx}_{\mathrm{o}}} Q_1$, there is Q_2 such that $C[Q_1] \Longrightarrow Q_2 \mathrel{\dot{\approx}_{\mathrm{o}}} P_2$, so in particular it holds that not $Q_2 \Downarrow_s$ and not $Q_2 \Downarrow_{s'}$ and not $Q_2 \Downarrow t$. The only possibility is that

$$Q_2 = \boldsymbol{\nu}z\,\Big(\,!\overline{a}z \mid \Pi_{c\in \mathsf{fn}(P,Q)}[z = c]t \mid \boldsymbol{\nu}\widetilde{x}\,(\Pi_{i\in I}!\overline{a_i}x_i \mid Q')\Big)$$

for some Q' such that $Q \stackrel{\overline{x}(z)}{\Longrightarrow} Q'$. Since $P_2 \mathrel{\dot{\approx}_{\mathrm{o}}} Q_2$ we have, using axioms of structural congruence and the property

$$\boldsymbol{\nu}y\,([y = b]M \mid R) \mathrel{\dot{\approx}_{\mathrm{o}}} \boldsymbol{\nu}y\,R \qquad \text{if } y \neq b,$$

that

$$P_2 \dot{\approx}_{\mathrm{o}} \boldsymbol{\nu} z\, \boldsymbol{\nu} \widetilde{x} \left(!\overline{a}z \mid \Pi_{i \in I} !\overline{a_i} x_i \mid P' \right)$$

and

$$Q_2 \dot{\approx}_{\mathrm{o}} \boldsymbol{\nu} z\, \boldsymbol{\nu} \widetilde{x} \left(!\overline{a}z \mid \Pi_{i \in I} !\overline{a_i} x_i \mid Q' \right) .$$

Hence $(P', Q') \in \mathcal{R}^{\widetilde{x} \cup \{z\}}$, as required.

This completes the outline of the proof of the theorem. □

References

1. M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *28th Annual Symposium on Principles of Programming Languages*. ACM, 2001.
2. P. Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*. North Holland, 1977.
3. R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
4. C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *ICALP'98: Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
5. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
6. A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of Concurrent ML with local names. In *15th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2000.
7. M. Merro. *Locality in the π-calculus and Applications to Object-Oriented Languages*. PhD thesis, Ecole des Mines de Paris, 2000.
8. R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993.
9. R. Milner and D. Sangiorgi. Barbed bisimulation. In *ICALP'92: Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
10. U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, XVI(2):171–199, 1992.
11. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.
12. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996.
13. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

CCS with Priority Guards

Iain Phillips*

Department of Computing, Imperial College, London
iccp@doc.ic.ac.uk

Abstract. It has long been recognised that standard process algebra has difficulty dealing with actions of different priority, such as for instance an interrupt action of high priority. Various solutions have been proposed. We introduce a new approach, involving the addition of "priority guards" to Milner's process calculus CCS. In our approach, priority is *unstratified*, meaning that actions are not assigned fixed levels, so that the same action can have different priority depending where it appears in a program. Unlike in other unstratified accounts of priority in CCS (such as that of Camilleri and Winskel), we treat inputs and outputs symmetrically. We introduce the new calculus, give examples, develop its theory (including bisimulation and equational laws), and compare it with existing approaches. We show that priority adds expressiveness to both CCS and the π-calculus.

1 Introduction

It has long been recognised that standard process algebra [13,9,2] has difficulty dealing with actions of different priority, such as for instance an interrupt action of high priority. Various authors have suggested how to add priority to process languages such as ACP [1,10], CCS [4,3] and CSP [8,7]. We introduce a new approach, involving the addition of "priority guards" to the summation operator of Milner's process calculus CCS. In our approach, priority is *unstratified*, meaning that actions are not assigned fixed levels, so that the same action can have different priority depending where it appears in a program. We shall see that existing accounts of priority in CCS are either stratified [4], or else they impose a distinction between outputs and inputs, whereby prioritised choice is only made on inputs [3,5]. This goes against the spirit of CCS, where inputs and outputs are treated symmetrically, and we contend that it is unnecessary. We introduce the new calculus, give examples, develop its theory (including bisimulation and equational laws), and compare it with existing approaches. We show that priority adds expressiveness to both CCS and the π-calculus.

We start with the idea of priority. We assume some familiarity with CCS notation [13]. Consider the CCS process $a + b$. The actions a and b have equal status. Which of them engages in communication depends on whether the environment is offering the complementary actions $\bar{a}$ or $\bar{b}$. By "environment" we

* Partially funded by EPSRC grant GR/K54663

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 305–320, 2001.
© Springer-Verlag Berlin Heidelberg 2001

mean whatever processes may be placed in parallel with $a + b$. We would like some means to favour a over b, say, so that if the environment offers both, then only a can happen. This would be useful if, for instance, a was an interrupt action. We need something more sophisticated than simply removing b altogether, since, if a cannot communicate, it should not stop b from doing so. This brief analysis points to two features of priority: (1) Priority removes ("preempts") certain possibilities that would have existed without priority. Thus if a can communicate then b is preempted. (2) Reasoning about priority in CCS has to be parametrised by the environment.

We now explain the basic idea of priority guards. Let P be a process, let a be an action, and let U be some set of actions. Then we can form a new process $U : a.P$, which behaves like $a.P$, except that the initial action a is conditional on the environment not offering actions in $\bar{U}$, the CCS "complement" of U. We call U a *priority guard* in $U : a.P$. All actions in U have priority over a at this point in the computation. We call our calculus CPG (for CCS with Priority Guards).

As a simple example, if we have a CCS process $a.P + b.Q$ and we wish to give a priority over b in the choice, we add a priority guard to get $a.P + a : b.Q$ (we omit the set braces around a). Priority is specific to this choice, since the guard affects only the initial b, and not any further occurrences of b there may be in Q.

Let us see how this example is handled in two existing approaches to priority. Cleaveland and Hennessy [4] add new higher priority actions to CCS. They would write our example as $\underline{a}.P + b.Q$ (high priority actions are underlined). In their stratified calculus, actions have fixed priority levels, and only actions at the same priority level can communicate. In this paper we are interested in an unstratified approach, and so our starting point of reference is Camilleri and Winskel's priority choice operator [3]. In their notation the example becomes $a.P +\!\!\rangle b.Q$. They make the priority of a over b specific to the particular choice, so that b might have priority over a elsewhere in the same program. We shall compare our approach with these two existing ones in Sect. 2.

A striking by-product of adding priority guards to CCS is that we can encode mixed input and output guarded summation using priority guards and restricted parallel composition. As a simple example, $a.P + \bar{b}.Q$ can be encoded with priority guards as $\mathrm{new}\, c(c : a.(P|\bar{c})|c : \bar{b}.(Q|\bar{c}))$ (where c is a fresh action). This expresses in a natural way the preemptive nature of $+$, whereby pursuing one option precludes the others. The same effect can be achieved in Camilleri and Winskel's calculus (but only for input guards): $a.P + b.Q$ can be encoded as

$$\mathrm{new}\, c\, ((c +\!\!\rangle a.(P|\bar{c})|(c +\!\!\rangle b.(Q|\bar{c})))$$

but of course here we are exchanging one form of choice for another. We shall return to this encoding of summation in Sect. 6.

To end this section, we give an example, involving handling of hidden actions and the scoping of priority. We wish to program a simple interrupt. Let P be a system which consists of two processes A, B in parallel which perform actions a, b respectively, while communicating internally to keep in step with each other.

P also has an interrupt process I which shuts down A and B when the interrupt signal int is received.

$$P \stackrel{\text{df}}{=} \text{new } \mathsf{mid}, \mathsf{int}_A, \mathsf{int}_B\ (A|B|I)\ A \stackrel{\text{df}}{=} \mathsf{int}_A : a.\overline{\mathsf{mid}}.A + \mathsf{int}_A$$
$$I \stackrel{\text{df}}{=} \mathsf{int}.(\overline{\mathsf{int}}_A.\overline{\mathsf{int}}_B + \overline{\mathsf{int}}_B.\overline{\mathsf{int}}_A) \quad B \stackrel{\text{df}}{=} \mathsf{int}_B : b.\mathsf{mid}.B + \mathsf{int}_B$$

Without the priority guards in A and B, P could receive an int and yet A and B could continue with a and b. Actions int_A, int_B have priority over a, b, respectively. This only applies within the scope of the restriction. We can apply the usual techniques of CCS (including removing τ actions) and get

$$P = a.P_1 + b.P_2 + \mathsf{int} \quad P_1 = b.P + \mathsf{int} \quad P_2 = a.P + \mathsf{int}$$

which is what we wanted. We consider this example more precisely in Sect. 8.

Notice that in the system as a whole, once the high-priority interrupt action is restricted, we have regained a standard CCS process without priority. Thus priority can be encapsulated locally, which we regard as an important feature when programming larger systems, where different priority frameworks may be in use in different subsystems.

The rest of the paper is organised as follows: First we compare our approach with related work (Sect. 2). Next we define the language of processes (Sect. 3). Then we look at reactions (Sect. 4) and labelled transitions (Sect. 5). We then look at bisimulation and equational theories for both the strong (Sect. 6) and weak cases (Sect. 7). We then return to our interrupt example (Sect. 8), and look at the extra expressiveness afforded by priority guards (Sect. 9). The paper is completed with some brief conclusions.

2 Comparison with Related Work

We refer the reader to [5] for discussion of the many approaches taken by other authors to priority. Here we restrict ourselves to comparison of our work with that of Camilleri and Winskel [3] (referred to as CW for short) and Cleaveland, Lüttgen and Natarajan [5] (CLN for short).

2.1 Camilleri and Winskel (CW)

As we have seen, CW's CCS with a prioritised choice operator $P +\!\rangle Q$ allows priority to be decided in a way which is specific to each choice in a system. The idea of a priority choice between processes is interesting and natural. The authors present an operational semantics via a labelled transition relation, and define a bisimulation-based equivalence. They also give an axiomatisation of this equivalence which is complete for finite processes (i.e. those not using recursion). They do not show how to hide the τ-actions resulting from communications (though this is treated in [11]).

As we saw in the Introduction, reasoning about priority has to be parametrised on the environment. The CW transition relation is parametrised

on a set of output actions R. Thus $\vdash_R P \xrightarrow{\alpha} P'$ means that, in an environment which is ready to perform precisely the actions R, the process P can perform an action α to become P'. For example, $\vdash_R a{+\!\rangle}b \xrightarrow{a} 0$ (any R), while $\vdash_R a{+\!\rangle}b \xrightarrow{b} 0$ provided $\bar{a} \notin R$.

We have borrowed the idea of parametrisation on the environment for our labelled transition system for CPG. For us $P \xrightarrow{\alpha}_U P'$ means that, in an environment which offers no action in the set $\bar{U}$, process P can perform α to become P'. Our most basic rule is essentially $U : a.P \xrightarrow{a}_U P$, provided $a \notin U$.

Note that the CW syntax shows the environment only in the prioritised choice $a.P{+\!\rangle}b.Q$, and does this implicitly, in that $b.Q$'s "environment" is $a.P$, while $a.P$ says nothing about the actual environment. In CPG the environment is represented in the syntax directly.

There is a difference in expressiveness between CPG and CW's calculus, in that the latter cannot express cycles of priority, whereas we can in CPG. CW consider the paradoxical example new $a, b\ ((a{+\!\rangle}\bar{b})|(b{+\!\rangle}\bar{a}))$. The problem is that there is a circularity, with a having priority over b, as well as vice versa. Can the system act? They decide to sidestep this question by breaking the symmetry in CCS between inputs and outputs, and only allowing prioritised choice on input actions. We feel that this complicates the syntax and operational semantics, and should not be necessary. There seems to be no essential reason for CW not to allow the system with circular priorities, since their environmental transition relation should be able to handle it. In our approach the example is admitted, and results in a deadlock, which would seem to be in keeping with CW's approach. We consider this example again at the end of Sect. 5.

Another reason why CW disallow priority choice on output actions is to assist in obtaining the normal form they use for proving the completeness of their equational laws for finite processes. However this normal form is still quite complicated (consisting of a sum of priority sums of sums). In our calculus CPG we have only one form of choice, and so completeness is technically simpler.

2.2 Cleaveland, Lüttgen, and Natarajan (CLN)

In CLN's basic approach [5], which is derived from earlier work of Cleaveland and Hennessy [4], actions have priority levels. Mostly they consider just two levels—ordinary actions and higher priority, underlined actions. Only actions at the same level of priority can communicate, which is really quite restrictive when one considers that two actions which are intended to communicate may have quite different priorities within their respective subsystems. Silent actions resulting from communication have preemptive power over all actions of lower priority. The authors present both strong and weak bisimulation-based equivalences (drawing on [15]), and axiomatise these for finite processes.

In our unstratified calculus CPG, by contrast, actions do not have priority levels—each priority guard operates independently, in the spirit of [3].

We referred in the Introduction to the desirability of encapsulating priorities locally. This encapsulation is present in Camilleri and Winskel's calculus (and in

our own), but not in Cleaveland and Hennessy's, since a high priority $\underline{\tau}$ is treated differently from a standard τ. However, the development in [5] goes beyond the basic Cleaveland and Hennessy calculus to consider distributed priorities, where preemption is decided locally rather than globally. Consideration is also given to extending the distributed priority calculus to allow communication between actions at different levels. The authors identify a problem with associativity of parallel composition. Consider the system

$$(a + \underline{b})|(\bar{b} + \underline{c})|\underline{\bar{c}}$$

where communication is allowed between complementary actions at different levels. If this associates to the left, then a is preempted by b; however if it associates to the right then b is preempted by c, and so a is not preempted. A similar problem is encountered when extending the distributed calculus to allow more than two levels. CLN's proposed solution is to follow CW by only allowing priorities to be resolved between *input* actions, while treating all output actions as having equal priority. We have already mentioned our reservations about this. Nevertheless the distinction between inputs and outputs gives a workable "mixed-level" calculus (distributed, multi-level, with communication between different levels). It is particularly nice that CLN show that the CW calculus can be translated faithfully and naturally into this mixed-level calculus.

It is striking that both CW and the mixed-level calculus of CLN adopt the same syntactic restriction on inputs and outputs, and also that only *strong* equivalence (τ actions not hidden) is presented for the mixed-level calculus. We shall present a weak equivalence for CPG.

3 The Language CPG

We shall denote our augmentation of CCS with priority guards by *CPG* (CCS with Priority Guards). First we define the actions of CPG. In standard CCS [13, Part I] there is a set of *names* $\mathcal{N}$ and a disjoint set of *co-names* $\bar{\mathcal{N}}$, together with a single silent action τ. To these standard names $\mathcal{N}$ we shall add a new disjoint set of names $\mathcal{U}$ and a dual set $\bar{\mathcal{U}}$. These are the actions which can be used in priority guards; they can also be used in the standard way. They need to be kept separate from standard actions, since we have to be careful with them in reasoning compositionally about processes.

To see why we take this approach, consider the law $P = \tau.P$, which is valid for CCS processes.[1] In CPG, if a can be a priority guard then $a \neq \tau.a$ since there is a context in which the two sides behave differently. Indeed, $a|\bar{a}:b$ cannot perform b (since, as we shall see, b is preempted by the offer of a), whereas $\tau.a|\bar{a}:b$ can perform b initially, as a is not offered until τ has occurred. However if we know that a is a standard name then we do have $a = \tau.a$. So we can retain CCS reasoning when processes only involve standard names.

[1] We are following the formulation of CCS in [13] rather than that of [12]. Processes such as $P + (Q|R)$ are not allowed, only guarded choices $\sum \alpha_i.P_i$.

We define $\mathsf{Std} = \mathcal{N} \cup \bar{\mathcal{N}}$, $\mathsf{Pri} = \mathcal{U} \cup \bar{\mathcal{U}}$, $\mathsf{Vis} = \mathsf{Std} \cup \mathsf{Pri}$ and $\mathsf{Act} = \mathsf{Vis} \cup \{\tau\}$. We let $u, v, \ldots$ range over Pri, $a, b, \ldots$ over Vis and $\alpha, \beta, \ldots$ over Act. Also $S, T, \ldots$ range over finite subsets of Vis, and $U, V \ldots$ over finite subsets of Pri. If $S \subseteq \mathsf{Vis}$, let $\bar{S}$ denote $\{\bar{a} : a \in S\}$, where if $\bar{a} \in \bar{\mathcal{N}} \cup \bar{\mathcal{U}}$ then $\bar{\bar{a}} = a$.

Now we define processes:

Definition 1. *(cf [13, Definition 4.1]) $\mathcal{P}$ is the smallest set such that whenever P, P_i are processes then $\mathcal{P}$ contains*

1. $\sum_{i \in I} S_i : \alpha_i.P_i$ (guarded summation: I finite, each S_i finite)
2. $P_1 | P_2$ (parallel composition)
3. new a P (restriction)
4. $A\langle a_1, .., a_n \rangle$ (identifier)

$\mathcal{P}$ is ranged over by $P, Q, R, \ldots$ We let $M, N, \ldots$ range over (guarded) summations. We assume that each identifier $A\langle b_1, .., b_n \rangle$ comes with a defining equation $A(a_1, .., a_n) \stackrel{\text{df}}{=} P$, where P is a process whose free names are drawn from $a_1, .., a_n$. We will tend to abbreviate $a_1, .., a_n$ by $\overrightarrow{a}$. We write the empty guarded summation as 0 and abbreviate $S : \alpha.0$ by $S : \alpha$. It is assumed that the order in a summation is immaterial. We abbreviate $\emptyset : \alpha$ by α. Definition 1 is much as in standard CCS except for the priority guards S_i. The meaning of the priority guard $S : \alpha$ is that α can only be performed if the environment does not offer any action in $\bar{S} \cap \mathsf{Pri}$. Clearly, any names in $S - \mathsf{Pri}$ have no effect as guards, and can be eliminated without changing the behaviour of a process. We allow them to occur in the syntax, since otherwise we could not freely instantiate the parameters in an identifier. We write $u : \alpha$ instead of $\{u\} : \alpha$. Restriction is a variable-binding operator, and we write $\mathsf{fn}(P)$ for the free names of P.

Two sublanguages of CPG are of interest:

Definition 2. *Let $\mathcal{P}_{\mathsf{Std}}$ be the sublanguage of* standard *processes generated as in Definition 1 except that all names are drawn from* Std *(i.e. we effectively take $\mathcal{U} = \emptyset$ and $S_i = \emptyset$ in clause (1)). Let $\mathcal{P}_{\mathsf{Ug}}$ be the sublanguage of* unguarded *processes generated as in Definition 1 except that all priority guards are empty (i.e. $S_i = \emptyset$ in clause (1)).*

Clearly $\mathcal{P}_{\mathsf{Std}} \subseteq \mathcal{P}_{\mathsf{Ug}} \subseteq \mathcal{P}$. Note that $\mathcal{P}_{\mathsf{Std}}$ is effectively standard CCS. The unguarded processes $\mathcal{P}_{\mathsf{Ug}}$ differ from $\mathcal{P}_{\mathsf{Std}}$ in that they may contain names in Pri. Such processes cause no problems for strong equivalence (Proposition 4), but care is needed with weak equivalence (Sect. 7), since e.g. u and $\tau.u$ ($u \in \mathsf{Pri}$) are not weakly equivalent, as remarked above.

4 Offers and Reaction

Structural congruence is the most basic equivalence on processes, which facilitates reaction by bringing the subprocesses which are to react with each other into juxtaposition. It is defined as for CCS:

Definition 3. *(cf [13, Definition 4.7])* Structural congruence*, written* $\equiv$*, is the congruence on* $\mathcal{P}$ *generated by the following equations:*

1. Change of bound names (alpha-conversion)
2. $P|0 \equiv P$, $P|Q \equiv Q|P$, $P|(Q|R) \equiv (P|Q)|R$
3. new $a\ (P|Q) \equiv P|\text{new } a\ Q$ if $a \notin \mathsf{fn}(P)$;
   new $a\ 0 \equiv 0$, new a new $b\ P \equiv$ new b new $a\ P$
4. $A\langle \overrightarrow{b} \rangle \equiv \{\overrightarrow{b}/\overrightarrow{a}\}P$ if $A(\overrightarrow{a}) \stackrel{\text{df}}{=} P$

Recall that a guarded action $S:a$ is conditional on other processes in the environment not offering actions in $\bar{S} \cap \mathsf{Pri}$. Before defining reaction we define for each process P the set $\mathsf{off}(P) \subseteq \mathsf{Pri}$ of "higher priority" actions "offered" by P.

Definition 4. *By induction on* $P \in \mathcal{P}$*:*

1. $\mathsf{off}(\sum_{i\in I} S_i{:}\alpha_i.P_i) = \{\alpha_i : i \in I, \alpha_i \in \mathsf{Pri}, \alpha_i \notin S_i\}$
2. $\mathsf{off}(P_1|P_2) = \mathsf{off}(P_1) \cup \mathsf{off}(P_2)$
3. $\mathsf{off}(\text{new } a\ P) = \mathsf{off}(P) - \{a, \bar{a}\}$
4. $\mathsf{off}(A\langle \overrightarrow{b} \rangle) = \mathsf{off}(\{\overrightarrow{b}/\overrightarrow{a}\}P)$ if $A(\overrightarrow{a}) \stackrel{\text{df}}{=} P$

In item 1 the reason that we insist $\alpha_i \notin S_i$ is that we want to equate a process such as $u:u$ with 0, since $u:u$ can never engage in a reaction. Note that if $P \in \mathcal{P}_{\mathsf{Std}}$ then $\mathsf{off}(P) = \emptyset$.

In CPG, a reaction can be conditional on offers from the environment. Consider $u : b|\bar{b}$. This can react by communication on b, $\bar{b}$. However b is guarded by u, and so the reaction is conditional on the environment not offering $\bar{u}$. We reflect this by letting reaction be parametrised on sets of actions $U \subseteq \mathsf{Pri}$. The intended meaning of $P \to_U P'$ is that P can react on its own, as long as the environment does not offer $\bar{u}$ for any $u \in U$ (in our parlance, "eschews" U).

Definition 5. *Let* $P \in \mathcal{P}$ *and let* $S \subseteq \mathsf{Act}$ *be finite.* P eschews S *(written* $P\ \mathsf{eschews}\ S$*) iff* $\mathsf{off}(P) \cap \bar{S} = \emptyset$.

Definition 6. *(cf [13, Definition 4.13]) The* reaction relation *on* $\mathcal{P}$ *is the smallest relation* $\to$ *on* $\mathcal{P} \times \wp(\mathsf{Pri}) \times \mathcal{P}$ *generated by the following rules:*

$$S{:}\tau.P + M \to_{S\cap\mathsf{Pri}} P$$

$$\frac{S{:}a.P + M\ \mathsf{eschews}\ T \quad T{:}\bar{a}.Q + N\ \mathsf{eschews}\ S}{(S{:}a.P + M)|(T{:}\bar{a}.Q + N) \to_{(S\cup T)\cap\mathsf{Pri}} P|Q} \qquad \frac{P \to_U P' \quad Q\ \mathsf{eschews}\ U}{P|Q \to_U P'|Q}$$

$$\frac{P \to_U P'}{\text{new } a\ P \to_{U-\{a,\bar{a}\}} \text{new } a\ P'} \qquad \frac{P \to_U P' \quad P \equiv Q \quad P' \equiv Q'}{Q \to_U Q'}$$

We abbreviate $P \to_\emptyset P'$ *by* $P \to P'$.

The second clause of Definition 6 is the most important. In order for an action a to react with a complementary $\bar{a}$, the two sides must not preempt each other (i.e. they must eschew each other's guards). Furthermore the reaction remains conditional on the environment eschewing the union of their guards. The restriction rule shows how this conditionality can then be removed by scoping. Notice that if we restrict attention to the unguarded processes $\mathcal{P}_{\mathsf{Ug}}$ (i.e. we let $U = \emptyset$) we recover the usual CCS reaction relation. So the new reaction relation is conservative over the old.

5 Labelled Transitions

As in standard CCS, we wish to define a transition relation on processes $P \xrightarrow{\alpha} P'$ meaning that P can perform action α and become P'. As we did with reaction, we refine the transition relation so that it is parametrised on sets of actions $U \subseteq \mathsf{Pri}$. The intended meaning of $P \xrightarrow{\alpha}_U P'$ is that P can perform α as long as the environment eschews U. Our definition is inspired by the transition relation in [3], which is parametrised on what set of output actions the environment is ready to perform.

Definition 7. *(cf [13, Definition 5.1]) The* transition relation *on $\mathcal{P}$ is the smallest relation $\rightarrow$ on $\mathcal{P} \times Act \times \wp(\mathsf{Pri}) \times \mathcal{P}$ generated by the following rules:*

$$\text{(sum)} \qquad M + S{:}\alpha.P + N \xrightarrow{\alpha}_{S \cap \mathsf{Pri}} P \quad \text{if } \alpha \notin S \cap \mathsf{Pri}$$

$$\text{(react)} \qquad \frac{P_1 \xrightarrow{a}_{U_1} P_1' \quad P_2 \xrightarrow{\bar{a}}_{U_2} P_2' \quad P_1 \,\mathsf{eschews}\, U_2 \quad P_2 \,\mathsf{eschews}\, U_1}{P_1|P_2 \xrightarrow{\tau}_{U_1 \cup U_2} P_1'|P_2'}$$

$$\text{(par)} \qquad \frac{P_1 \xrightarrow{\alpha}_U P_1' \quad P_2 \,\mathsf{eschews}\, U}{P_1|P_2 \xrightarrow{\alpha}_U P_1'|P_2} \qquad \frac{P_2 \xrightarrow{\alpha}_U P_2' \quad P_1 \,\mathsf{eschews}\, U}{P_1|P_2 \xrightarrow{\alpha}_U P_1|P_2'}$$

$$\text{(res)} \qquad \frac{P \xrightarrow{\alpha}_U P'}{\mathrm{new}\, a\, P \xrightarrow{\alpha}_{U - \{a,\bar{a}\}} \mathrm{new}\, a\, P'} \quad \text{if } \alpha \notin \{a, \bar{a}\}$$

$$\text{(ident)} \qquad \frac{\{\overrightarrow{b}/\overrightarrow{a}\}P \xrightarrow{\alpha}_U P'}{A\langle \overrightarrow{b} \rangle \xrightarrow{\alpha}_U P'} \quad \text{if } A(\overrightarrow{a}) \stackrel{\mathrm{df}}{=} P$$

We abbreviate $P \xrightarrow{\alpha}_{\emptyset} P'$ by $P \xrightarrow{\alpha} P'$ and $\exists P'.P \xrightarrow{\alpha}_U P'$ by $P \xrightarrow{\alpha}_U$.

Proposition 1. *If $P \xrightarrow{\alpha}_U P'$ then $\alpha \notin U$ and U is finite. Moreover,*

$$\{u \in \mathsf{Pri} : \exists U.P \xrightarrow{u}_U\} \subseteq \mathsf{off}(P) \ .$$

To see that $\mathsf{off}(P)$ can be unequal to $\{u \in \mathsf{Pri} : \exists U.P \xrightarrow{u}_U\}$, consider $u{:}v.0|\bar{u}.0$. We see that $\mathsf{off}(u{:}v|\bar{u}) = \{v, \bar{u}\}$, but $u{:}v|\bar{u}$ cannot perform v.

As with reaction, note that if we restrict attention to the unguarded processes $\mathcal{P}_{\mathsf{Ug}}$ (i.e. we let $U = \emptyset$) we recover the usual CCS transition relation. So the new transition relation is conservative over the old. In applications we envisage that

the standard CCS transition relation can be used most of the time. The CPG transition relation will only be needed in those subsystems which use priority.

As an illustration of the design choices embodied in our definitions, consider the circular example of Camilleri & Winskel (Subsect. 2.1):

$$P \stackrel{\text{df}}{=} u.a + u:\bar{v} \quad Q \stackrel{\text{df}}{=} v.b + v:\bar{u} \quad R \stackrel{\text{df}}{=} \text{new } u, v\ (P|Q)$$

In P action u has priority over $\bar{v}$, while in Q action v has priority over $\bar{u}$. We have $P \stackrel{u}{\to} a$, $Q \stackrel{\bar{u}}{\to}_v 0$. For a u communication to happen, by rule (react) we need $\bar{v} \notin \mathsf{off}(P)$, but $\mathsf{off}(P) = \{u, \bar{v}\}$, so that the u communication cannot happen. Similarly the v communication cannot happen, and so R is strongly equivalent to 0 (strong offer equivalence is defined in the next section).

6 Strong Offer Bisimulation

Similarly to standard CCS, we define process equivalences based on strong and weak bisimulation. We consider strong bisimulation in this section and weak bisimulation (i.e. with hiding of silent actions) in the next.

The intuition behind our notion of bisimulation is that for processes to be equivalent they must make the same offers, and for a process Q to simulate a process P, Q must be able to do whatever P can, though possibly constrained by fewer or smaller priority guards. For instance, we would expect the processes $a + u:a$ and a to be equivalent, since the priority guarded $u:a$ is simulated by a.

Definition 8. *(cf [13]) A symmetric relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ is a* strong offer bisimulation *if $\mathcal{S}(P, Q)$ implies both that $\mathsf{off}(P) = \mathsf{off}(Q)$ and that for all $\alpha \in \mathsf{Act}$,*

if $P \stackrel{\alpha}{\to}_U P'$ then for some Q' and $V \subseteq U$, we have $Q \stackrel{\alpha}{\to}_V Q'$ and $\mathcal{S}(P', Q')$

Definition 9. *Processes P and Q are* strongly offer equivalent*, written $P \stackrel{\mathsf{off}}{\sim} Q$, iff there is some strong offer bisimulation $\mathcal{S}$ such that $\mathcal{S}(P, Q)$.*

Proposition 2. *(cf [13, Prop 5.2]) $\equiv$ is a strong offer bisimulation. Hence $\equiv$ implies $\stackrel{\mathsf{off}}{\sim}$.* □

Proposition 3. *(cf [13, Theorem 5.6]) $P \stackrel{\tau}{\to}_U \equiv P'$ iff $P \to_U P'$* □

Theorem 1. *(cf [13, Proposition 5.29]) Strong offer equivalence is a congruence, i.e. if $P \stackrel{\mathsf{off}}{\sim} Q$ then*

1. $S:\alpha.P + M \stackrel{\mathsf{off}}{\sim} S:\alpha.Q + M$
2. new a $P \stackrel{\mathsf{off}}{\sim}$ new a Q
3. $P|R \stackrel{\mathsf{off}}{\sim} Q|R$
4. $R|P \stackrel{\mathsf{off}}{\sim} R|Q$

□

Note that if $P, Q \in \mathcal{P}_{\mathsf{Ug}}$ then $P \overset{\mathsf{off}}{\sim} Q$ iff $P \sim Q$, where $P \sim Q$ denotes that P and Q are strongly equivalent in the usual sense of [13]. So $\overset{\mathsf{off}}{\sim}$ is conservative over $\sim$. In fact we can say more:

Proposition 4. *Let $P, Q \in \mathcal{P}_{\mathsf{Ug}}$. If $P \sim Q$ then $C[P] \overset{\mathsf{off}}{\sim} C[Q]$, for any context $C[\cdot]$.* □

So we can reuse all the known equivalences between CCS processes when working with CPG processes.

Proposition 5. *(cf [13, Proposition 5.21]) For all $P \in \mathcal{P}$,*

$$P \overset{\mathsf{off}}{\sim} \sum\{U : \alpha.Q : P \xrightarrow{\alpha}_U Q\}$$

Proposition 6. *The following laws hold:*

$$M + S : \alpha.P \overset{\mathsf{off}}{\sim} M + (S \cap \mathsf{Pri}) : \alpha.P \tag{1}$$

$$M + U : \alpha.P \overset{\mathsf{off}}{\sim} M \qquad \text{if } \alpha \in U \subseteq \mathsf{Pri} \tag{2}$$

$$M + U : \alpha.P + (U \cup V) : \alpha.P \overset{\mathsf{off}}{\sim} M + U : \alpha.P \tag{3}$$

$$\begin{aligned}(\sum U_i : \alpha_i.P_i) \mid (\sum V_j : \beta_j.Q_j) \\ \overset{\mathsf{off}}{\sim} \sum\{U_i : \alpha_i.(P_i | (\sum V_j : \beta_j.Q_j)) : \forall j.\beta_j \notin \bar{U}_i\} \\ + \sum\{V_j : \beta_j.((\sum U_i : \alpha_i.P_i) | Q_j)) : \forall i.\alpha_i \notin \bar{V}_j\} \\ + \sum\{(U_i \cup V_j) : \tau.P_i | Q_j : \alpha_i = \bar{\beta}_j \in \mathsf{Vis}, \forall i', j'.\alpha_{i'} \notin \bar{V}_j, \beta_{j'} \notin \bar{U}_i\}\end{aligned} \tag{4}$$

$$\text{new } a\, (\sum U_i : \alpha_i.P_i) \overset{\mathsf{off}}{\sim} \sum((U_i - \{a, \bar{a}\}) : \alpha_i.\text{new } a\, P_i : \alpha_i \neq a, \bar{a}\} \tag{5}$$

Definition 10. *Let $\mathcal{A}_S$ be the following set of axioms: the axioms of structural congruence $\equiv$ (Definition 3) together with the five laws of Proposition 6.*

Theorem 2. *The set of axioms $\mathcal{A}_S$ is complete for $\overset{\mathsf{off}}{\sim}$ on finite CPG processes (a CPG process is* finite *if it contains no identifiers).* □

As mentioned in the Introduction, we can encode mixed input and output guarded summation using priority guards and restricted parallel composition.

Proposition 7. *Suppose that* $\{\alpha_i : i \in I\}$ *are actions which cannot react with each other, i.e. there do not exist* $i, j \in I$ *and* $a \in \mathsf{Vis}$ *such that* $\alpha_i = a$ *and* $\alpha_j = \bar{a}$*. Then*

$$\sum S_i : \alpha_i.P_i \overset{\mathsf{off}}{\sim} \text{new } u \, (\prod (S_i \cup \{u\} : \alpha_i(P_i|\bar{u})))$$

where $u \in \mathsf{Pri}$ *is some fresh name not occurring in* $\sum S_i : \alpha_i.P_i$ *and* $\prod$ *denotes parallel composition.* □

The non-reaction condition in Proposition 7 is needed, since otherwise the right-hand side would have extra unwanted reactions. The condition is not unduly restrictive, since if we have a system where the same channel a is used to pass messages both to and from a process, we can simply separate a out into two separate channels, one for each direction.

7 Weak Offer Bisimulation

We now investigate weak bisimulation, where reactions are hidden.

Definition 11. $P \Rightarrow_U P'$ *iff* $P \overset{\mathrm{id}}{=} P'$ *or* $\exists U_1, \ldots, U_n . P \rightarrow_{U_1} \cdots \rightarrow_{U_n} P'$ *with* $U = U_1 \cup \cdots \cup U_n$ *(*$n \geq 1$*). We abbreviate* $P \Rightarrow_\emptyset P'$ *by* $P \Rightarrow P'$*.*

$P \overset{\alpha}{\Rightarrow}_U P'$ *iff* $\exists U', U'' . P \Rightarrow_{U'} P'' \overset{\alpha}{\rightarrow}_{U''} P''' \Rightarrow P'$ *with* $U = U' \cup U''$ *and* $\mathsf{off}(P'') \subseteq \mathsf{off}(P)$*.*

Here $P \overset{\mathrm{id}}{=} P'$ means that P and P' are identically equal. So $P \Rightarrow_U P'$ allows zero or more internal transitions with guards included in U. The condition $\mathsf{off}(P'') \subseteq \mathsf{off}(P)$ is needed to obtain a weak equivalence which is a congruence. The reason why we allow priority guards before performing a visible action, but not after, is as follows: For Q to simulate $P \overset{a}{\rightarrow}_U P'$, Q must expect an environment offering $\bar{U}$ up to and including performing a. After this, the environment has changed, and might be offering anything. So Q can perform further reactions to reach Q' simulating P', but these reactions must not be subject to any priority guards.

Definition 12. *A symmetric relation* $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ *is a* weak offer bisimulation *if* $\mathcal{S}(P, Q)$ *implies both that* $\mathsf{off}(P) = \mathsf{off}(Q)$ *and that:*

if $P \rightarrow_U P'$ *then for some* Q' *and* $U' \subseteq U$*, we have* $Q \Rightarrow_{U'} Q'$ *and* $\mathcal{S}(P', Q')$*, and for all* $a \in \mathsf{Vis}$*,*

if $P \overset{a}{\rightarrow}_U P'$ *then for some* Q' *and* $U' \subseteq U$*, we have* $Q \overset{a}{\Rightarrow}_{U'} Q'$ *and* $\mathcal{S}(P', Q')$*.*

On the sublanguage $\mathcal{P}_{\mathsf{Std}}$ (which corresponds to CCS) weak offer bisimulation is the same as for CCS [13, Proposition 6.3].

Definition 13. *Processes* P *and* Q *are* weakly offer equivalent*, written* $P \overset{\mathsf{off}}{\approx} Q$*, iff there is some weak offer bisimulation* $\mathcal{S}$ *such that* $\mathcal{S}(P, Q)$*.*

Proposition 8. *For any P, Q, if $P \overset{\mathsf{off}}{\sim} Q$ then $P \overset{\mathsf{off}}{\approx} Q$.* □

Theorem 3. *(cf [13, Proposition 6.17]) $\overset{\mathsf{off}}{\approx}$ is a congruence.* □

So we have a congruence which conservatively extends CCS.

We now turn to the equational theory of weak offer equivalence. In CCS we have the law $P \approx \tau.P$ [13, Theorem 6.15]. However in CPG, $u \overset{\mathsf{off}}{\not\approx} \tau.u$. This is because $\mathsf{off}(u) = \{u\}$ whereas $\mathsf{off}(\tau.u) = \emptyset$. However the usual CCS equivalence laws will still hold for the standard processes $\mathcal{P}_{\mathsf{Std}}$ (recall that for $P \in \mathcal{P}_{\mathsf{Std}}$, $\mathsf{off}(P) = \emptyset$).

Proposition 9. *(cf [13, Theorem 6.15]) The following laws hold:*

$$\tau.P \overset{\mathsf{off}}{\approx} P \quad \text{if } \mathsf{off}(P) = \emptyset \tag{6}$$

$$M + N + \tau.N \overset{\mathsf{off}}{\approx} M + \tau.N \quad \text{if } \mathsf{off}(N) \subseteq \mathsf{off}(M) \tag{7}$$

$$M + \alpha.P + \alpha.(\tau.P + N) \overset{\mathsf{off}}{\approx} M + \alpha.(\tau.P + N) \tag{8}$$

We stated (6), (7) and (8) because in many situations it is convenient to use conventional CCS reasoning. The next result gives the "intrinsic" τ-laws of CPG:

Proposition 10. *The following four laws hold:*

$$M + U : \tau.M \overset{\mathsf{off}}{\approx} M \tag{9}$$

$$M + U : \tau.(N + V : \tau.P) \overset{\mathsf{off}}{\approx} M + U : \tau.(N + V : \tau.P) + (U \cup V) : \tau.P \tag{10}$$

If $\mathsf{off}(N + V : a.P) \subseteq \mathsf{off}(M)$:

$$M + U : \tau.(N + V : a.P) \overset{\mathsf{off}}{\approx} M + U : \tau.(N + V : a.P) + (U \cup V) : a.P \tag{11}$$

$$M + U : \alpha.P + U : \alpha.(\tau.P + N) \overset{\mathsf{off}}{\approx} M + U : \alpha.(\tau.P + N) \tag{12}$$

We can derive (7) from (10) and (11). Also we can derive:

$$\tau.M \overset{\mathsf{off}}{\approx} M \quad \text{if } \mathsf{off}(M) = \emptyset \tag{13}$$

from (9), (10), (11). Recall that every process is strongly equivalent to a summation (Proposition 5), and so (13) is effectively as strong as (6).

Definition 14. *Let $\mathcal{A}_W$ be the axioms $\mathcal{A}_S$ (Definition 10) together with (9), (10), (11) and (12).*

Theorem 4. *The axioms $\mathcal{A}_W$ are complete for $\overset{\mathsf{off}}{\approx}$ on finite processes.* □

Proposition 11. *(cf [13, Theorem 6.19]) Unique solution of equations. Let $\overrightarrow{X}$ be a (possibly infinite) sequence of process variables X_i. Up to $\overset{\mathsf{off}}{\approx}$, there is a unique sequence $\overrightarrow{P}$ of processes which satisfy the formal equations:*

$$X_i \overset{\mathsf{off}}{\approx} \sum_j U_{ij} : a_{ij}.X_{k(ij)}$$

(notice that τs are not allowed). □

8 Example

We now revisit the interrupt example from Sect. 1. Recall that we had:

$$P \overset{\mathrm{df}}{=} \mathrm{new}\ \mathsf{mid}, \mathsf{int}_A, \mathsf{int}_B\ (A|B|I)\ A \overset{\mathrm{df}}{=} \mathsf{int}_A : a.\overline{\mathsf{mid}}.A + \mathsf{int}_A$$
$$I \overset{\mathrm{df}}{=} \mathsf{int}.(\overline{\mathsf{int}}_A.\overline{\mathsf{int}}_B + \overline{\mathsf{int}}_B.\overline{\mathsf{int}}_A) \quad B \overset{\mathrm{df}}{=} \mathsf{int}_B : b.\mathsf{mid}.B + \mathsf{int}_B$$

We want to show $P \overset{\mathsf{off}}{\approx} Q$, where

$$Q \overset{\mathrm{df}}{=} a.Q_1 + b.Q_2 + \mathsf{int} \quad Q_1 \overset{\mathrm{df}}{=} b.Q + \mathsf{int} \quad Q_2 \overset{\mathrm{df}}{=} a.Q + \mathsf{int}$$

Clearly $\mathsf{int}_A, \mathsf{int}_B \in \mathsf{Pri}$. We take $a, b, \mathsf{mid}, \mathsf{int} \in \mathsf{Std}$. This means that $Q \in \mathcal{P}_{\mathsf{Std}}$. We can use Laws (4), (5) to get:

$$P \overset{\mathsf{off}}{\sim} a.P_1 + b.P_2 + \mathsf{int}.P_3$$

$$P_1 \overset{\mathsf{off}}{\sim} b.P_4 + \mathsf{int}.\tau \quad P_2 \overset{\mathsf{off}}{\sim} a.P_4 + \mathsf{int}.\tau \quad P_3 \overset{\mathsf{off}}{\sim} \tau.\tau + \tau.\tau \quad P_4 \overset{\mathsf{off}}{\sim} \tau.P + \mathsf{int}.\tau.P_3$$

where P_1, P_2, P_3, P_4 are various states of P. We can use law (6) to get:

$$P_1 \overset{\mathsf{off}}{\approx} b.P_4 + \mathsf{int} \quad P_2 \overset{\mathsf{off}}{\approx} a.P_4 + \mathsf{int} \quad P_3 \overset{\mathsf{off}}{\approx} 0 \quad P_4 \overset{\mathsf{off}}{\approx} \tau.P + \mathsf{int}$$

Then we use law (7) to get $\tau.P + \mathsf{int} \overset{\mathsf{off}}{\approx} \tau.P$. Notice that this needs $\mathsf{off}(P) = \emptyset$, i.e. $a, b, \mathsf{int} \notin \mathsf{Pri}$. Finally:

$$P \overset{\mathsf{off}}{\approx} a.P_1 + b.P_2 + \mathsf{int} \quad P_1 \overset{\mathsf{off}}{\approx} b.P + \mathsf{int} \quad P_2 \overset{\mathsf{off}}{\approx} a.P + \mathsf{int}$$

By Proposition 11 we get $P \overset{\mathsf{off}}{\approx} Q$ as we wanted.

Our reasoning was presented equationally, but could equally well have been done using bisimulation. We first unfolded the behaviour of P. Since all prioritised actions were restricted, the system P had no priorities as far as the environment was concerned. We could therefore remove silent actions and simplify using standard techniques of CCS.

In the Introduction we used plain equality (=) when talking about equivalence between CPG processes. This is to be interpreted as $\overset{\mathsf{off}}{\approx}$.

9 Expressiveness

In this section we show that priorities add expressive power to both CCS and the π-calculus [14,13]. As far as we are aware, this has not been previously shown for any notion of priority in process algebra. We use the work of Ene and Muntian [6], which was inspired by that of Palamidessi [16].

Definition 15. *[16] An* encoding $[\![\cdot]\!]$ *is a compositional mapping from (the processes of) one calculus to another. It is called* uniform *if* $[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$ *and, for any renaming* σ, $[\![\sigma(P)]\!] = \sigma([\![P]\!])$. *A semantics is* reasonable *if it distinguishes two processes* P *and* Q *whenever in some computation of* P *the actions on certain intended channels are different from those of any computation of* Q.

Definition 16. *(slight modification of [16, Definition 3.1]) A process*

$$P = P_1|\cdots|P_n$$

is an electoral system *if every computation of* P *can be extended (if necessary) to a computation which declares a leader* i *by outputting* $\bar{o}_i$, *and where no computation declares two different leaders.*

The intuition behind the following theorem is that priorities give us something of the power of one-many (broadcast) communication, in that a single process can simultaneously interrupt several other processes. By contrast, π-calculus communication is always one-one.

Theorem 5. *There is no uniform encoding of CPG into the* π*-calculus preserving a reasonable semantics.*

Proof. (Sketch) We follow the proof of Ene and Muntian's result that the broadcast π-calculus cannot be encoded in the π-calculus [6]. The network of CPG processes $P_1|\cdots|P_n$ is an electoral system, where $P_i \stackrel{\text{df}}{=} u\!:\!a.(\bar{u}|\bar{o}_i)|\bar{a}$. If P_i manages to communicate on a then P_i declares itself the leader. No other process can now do this, since P_i is preventing all the other processes from performing a by offering $\bar{u}$.

The rest of the proof is as in [6]. Suppose that we have an encoding $[\![\cdot]\!]$ of CPG into the π-calculus. Let $\sigma(o_i) = o_{m+i}$, with σ the identity otherwise. Consider $P_1|\cdots|P_{m+n}$. This is an electoral system and so the encoding $[\![P_1|\cdots|P_{m+n}]\!]$ must be also. But

$$\begin{aligned}[\![P_1|\cdots|P_{m+n}]\!] &= [\![(P_1|\cdots|P_m)|\sigma(P_1|\cdots|P_n)]\!] \\ &= [\![P_1|\cdots|P_m]\!]|[\![\sigma(P_1|\cdots|P_n)]\!]\end{aligned}$$

So we have two electoral systems of m and n processes respectively, which can be run independently in the π-calculus to produce two winners. Contradiction. □

Since CCS can be encoded in π-calculus, it follows that CPG has greater expressive power than CCS. It also follows that we can add expressive power to the π-calculus by adding priority guards.

Theorem 6. *There is no uniform encoding of the π-calculus into CPG preserving a reasonable semantics.*

Proof. Much as in [16], where it is shown for CCS rather than CPG. □

The results of this section apply equally to Camilleri-Winskel and Cleaveland-Hennessy-style priority.

10 Conclusions

We have introduced priority guards into CCS to form the language CPG. We have defined both strong and weak bisimulation equivalences and seen that they are conservative over the CCS equivalences, and that they are congruences. We have given complete equational laws for finite CPG in both the strong and weak cases. Conservation over CCS has the consequence that in verifying CPG systems we can often use standard CCS reasoning, as long as we take some care with actions in the set of prioritised actions Pri.

CPG overcomes the asymmetry between inputs and outputs present both in Camilleri and Winskel's calculus and in the corresponding calculus of Cleaveland, Lüttgen and Natarajan.

Finally, we have seen that priority guards add expressiveness to both CCS and the π-calculus.

We wish to thank Philippa Gardner, Rajagopal Nagarajan, Catuscia Palamidessi, Andrew Phillips, Irek Ulidowski, Maria Grazia Vigliotti, Nobuko Yoshida and the anonymous referees for helpful discussions and suggestions.

References

1. J.C.M. Baeten, J. Bergstra, and J.-W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, 9:127–168, 1986.
2. J. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.
3. J. Camilleri and G. Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, 1995.
4. R. Cleaveland and M.C.B. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, 1990.
5. R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001.
6. C. Ene and T. Muntian. Expressiveness of point-to-point versus broadcast communications. In *FCT '99*, volume 1684 of *Lecture Notes in Computer Science*, pages 258–268. Springer-Verlag, 1999.
7. C.J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, 1993.
8. H. Hansson and F. Orava. A process calculus with incomparable priorities. In *Proceedings of the North American Process Algebra Workshop*, pages 43–64, Stony Brook, New York, 1992. Springer-Verlag Workshops in Computer Science.

9. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
10. A. Jeffrey. A typed, prioritized process algebra. Technical Report 13/93, Dept. of Computer Science, University of Sussex, 1993.
11. C.-T. Jensen. *Prioritized and Independent Actions in Distributed Computer Systems.* PhD thesis, Aarhus University, 1994.
12. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
13. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
15. V. Natarajan, L. Christoff, I. Christoff, and R. Cleaveland. Priorities and abstraction in process algebra. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 14th Conference*, volume 880 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 1994.
16. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of the 25th Annual Symposium on Principles of Programming Languages, POPL '97*, pages 256–265. ACM, 1997.

A Testing Theory for Generally Distributed Stochastic Processes*

(Extended Abstract)

Natalia López and Manuel Núñez

Dpt. Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{natalia,mn}@sip.ucm.es

Abstract. In this paper we present a testing theory for stochastic processes. This theory is developed to deal with processes which probability distributions are not restricted to be exponential. In order to define this testing semantics, we compute the probability with which a process passes a test before an amount of time has elapsed. Two processes will be equivalent if they return the same probabilities for any test T and any time t. The key idea consists in *joining* all the random variables associated with the computations that the composition of process and test may perform. The combination of the values that this random variable takes and the probabilities of executing the actions belonging to the computation will give us the desired probabilities. Finally, we relate our stochastic testing semantics with other notions of testing.

1 Introduction

Process algebras [Hoa85,Hen88,Mil89,BW90] have become an important theoretical formalism to analyze distributed and concurrent systems. The first proposals were not powerful enough to describe some features of real systems. Due to that fact, process algebras have been extended with information to describe quantitative and qualitative features. Therefore, several timed (e.g. [RR88,BB93, NS94]), probabilistic (e.g. [LS91,GSS95,NdFL95,NdF95,CDSY99]), and timed-probabilistic (e.g. [Han91,Low95,GLNP97]) extensions of process algebras have appeared. However, these extensions are not enough to describe faithfully some systems. There exist systems where the probability to perform an action varies as time passes. So, during the last years a new extension has appeared: *Stochastic process algebras* [GHR93,ABC$^+$94,Hil96,BG98,HS00,HHK01]. These process algebras provide information about the probability to execute actions before an amount of time elapses. These probabilities are given by probability distribution functions. Except some of them ([BBG98,DKB98,HS00,BG01]), the majority works exclusively with exponential distributions. This assumption decreases the expressiveness of the languages. However, it simplifies several of the problems

* Work partially supported by the CICYT project TIC2000-0701-C02-01.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 321–335, 2001.
© Springer-Verlag Berlin Heidelberg 2001

that appear when considering general distributions. In particular, some quantities of interest, like reachability probabilities or steady-state probabilities, can be efficiently calculated by using well known methods on Markov chains. Nevertheless, the main weakness of non-exponential models, that is the analysis of properties, can be (partially) overcome by restricting the class of distributions. Phase-type distributions [Neu92] are a good candidate: They are closed under minimum, maximum, and convolution, and any other distribution over the interval $(0, \infty)$ can be approximated by arbitrarily accurate phase-type distributions. Moreover, the analysis of performance measures can be efficiently done in some general cases (see [BKLL95,EKN99] for the study of this kind of distributions in a stochastic process algebra).

In order to define the semantics of processes, the classical theory of testing [dNH84,Hen88] uses the idea of an *experimenter*. The notion of testing a process is specified by the interaction between the tested process and a set of tests. Usually, this interaction is modeled by the parallel composition of a process and a test. There have appeared testing semantics for probabilistic extensions (e.g. [Chr90,YL92,NdFL95,KN98,CDSY99]), timed extensions (e.g. [HR95,LdF97]), and probabilistic-timed extensions (e.g. [CLLS96,GLNP97]). Unfortunately, the testing theory has not been so extensively used in the field of stochastic process algebras. The definition of a testing semantics (fulfilling good properties) for this kind of languages is rather difficult, because sequences of stochastic transitions must be somehow abstracted and, in general, this is not an easy task. As far as we know, [BC00] represents the only proposal of a testing theory for stochastic process algebras. However, their study is restricted to processes which probability distribution functions are always exponential.

In this paper we will define a testing semantics for a stochastic process algebra where probability distributions are not restricted to be exponential. In our setting, processes may perform both standard actions (visible actions and internal τ actions) and stochastic actions which represent (random) delays. Our language will contain a probabilistic choice operator. In particular, this will imply that the selection among alternative stochastic transitions will be made by using a *preselection policy*. That is, according to the corresponding probabilities, one of the possible stochastic actions is chosen. Then, the process will be delayed an amount of time depending on the probability distribution associated with the chosen action.

Regarding tests, we will suppose that they cannot perform internal transitions. We impose this restriction because probabilistic testing produces very strange results if tests have the ability to perform internal transitions. For example, if we consider a CCS like probabilistic process algebras with a unique (probabilistic) choice operator[1] and we allow internal actions in tests, we will usually get that the processes $\tau\,;a\,;\texttt{STOP}$ and $a\,;\texttt{STOP}$ are not probabilistically testing equivalent. In [CDSY99] a more detailed discussion on probabilistic test-

[1] This is equivalent to consider (probabilistic) transitions systems where transitions are labeled by actions and probabilities.

ing semantics with and without internal actions in tests is presented. So, in order to keep a reasonable equivalence, we will not include internal actions in tests.

As usual, we will define the interaction between processes and tests as a parallel composition synchronizing in all the visible actions. This composition will produce a (multi)set of computations. In order to define the passing of tests, we will extract the appropriate information from these computations. This information will be a probability (indicating the probability of executing this computation) together with a random variable generated from all the random variables appearing in the sequence. So, our definition of passing tests takes into account not only the probabilities associated with computations, but also the *time* that these computations need to finish. For example, $0.3 = pass_{\leq 2}(P, T)$ indicates that the process P passes the test T with probability 0.3 before 2 units of time have passed. A similar mechanism is used in the testing semantics presented in [GLNP97]. Another alternative is given in [BC00] where the average time of the computation is used. Given the fact that we do not restrict the kind of probability distributions, in our setting, this technique would equate processes that present different stochastic behaviors.

In Fig. 1 we will use a graphical representation to introduce some stochastic processes. Greek letters like ξ, φ, and ψ (possibly decorated with an index) denote random variables, Latin letters represent visible actions, and τ represents an internal action. Transitions will be labeled by one of these actions together with a probability (we omit this value if it is equal to 1). Consider the processes P_1 and P_2. In this case, the key point consists in computing the probability to perform a before a certain amount of time has passed. If the random variable $\xi_1 + \xi_2$, that is the addition of the random variables ξ_1 and ξ_2, and the random variable ξ_3 are identically distributed, then we would like to equate P_1 and P_2. Let us note that $+$ denotes the addition of random variables, that is, the convolution of them. Consider now P_3, P_4 and P_5 in Fig. 1. These processes will be testing equivalent. Regardless of the temporal point where the (probabilistic) choice is taken, these processes follow the same *temporal* pattern (this would not be the case if a bisimulation semantics is considered). For example, P_3 will be firstly delayed according to ψ_1. Afterwards, it will be delayed either according to ψ_2 (with probability p) or according to ψ_3 (with probability $1 - p$). If we add the corresponding delays, we have that, with probability p, the action a (resp. b) will be performed after a delay determined by the addition of ψ_1 and ψ_2 (resp. the addition of ψ_1 and ψ_3). Intuitively, this is the very same situation for P_4 and P_5. Finally, P_6 and P_7 will also be testing equivalent. The reason is that both probabilistic choices produce the same result. This point motivates our presentation. In a stochastic process algebra where delays are separated from usual actions (as it is our case), stochastic *actions* must be considered somehow as *internal* actions carrying some additional information. Let us note that, in this example, if we replace φ_1 and φ_2 by *usual* actions b and c then the new processes are no longer testing equivalent.

The rest of the paper is structured as follows. In Sect. 2 we present our language and its operational semantics. In Sect. 3 we present our testing semantics

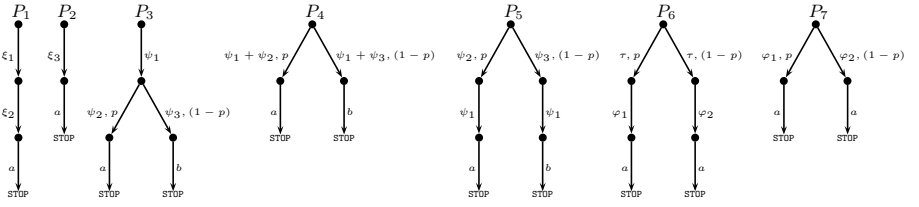


Fig. 1. Examples of stochastic processes.

by defining the set of tests, the interaction between processes and tests, and the corresponding notion of passing a test. Besides, a set of essential tests is given. In Sect. 4 we relate our semantics with other models of testing. Specifically, we will consider classical testing, pure probabilistic testing, and an adaptation of [BC00] to our framework. We will show that our testing semantics is a conservative extension of the first two. Finally, in Sect. 5 we present our conclusions and some lines for future work.

An extended version of this paper can be found in [LN01]. There, we present an alternative characterization of our testing equivalence. This characterization is based on a stochastic extension of the notion of *probabilistic acceptance sets* presented in [NdFL95].

2 Description of the Language

In this section we define our model of stochastic processes. First, we introduce some concepts on random variables. We will consider that the sample space (that is, the domain of random variables) is the set of real numbers $\mathbb{R}$ and that random variables take positive values only in $\mathbb{R}^+$, that is, given a random variable ξ we have $F_\xi(t) = 0$ for any $t < 0$. The reason for this restriction is that random variables will always be associated with time distributions.

Definition 1. Let ξ be a random variable. Its *probability distribution function*, denoted by F_ξ, is the function $F_\xi : \mathbb{R} \to [0, 1]$ such as $F_\xi(x) = \mathrm{P}(\xi \leq x)$, where $\mathrm{P}(\xi \leq x)$ is the probability that ξ assumes values less than or equal to x.

We consider a *distinguished* random variable. By *unit* we denote a random variable such that $F_{unit}(x) = 1$ for any $x \geq 0$, that is, *unit* is distributed as the Dirac distribution in 0. Let us note that if we consider the addition of random variables, for any random variable ξ, we have that $\xi + unit = \xi$.

We suppose a fixed set of visible actions Act ($a, a', \ldots$ to range over Act). We assume the existence of a special action $\tau \notin \mathrm{Act}$, which represents internal behaviour. We denote by Act_τ the set $\mathrm{Act} \cup \{\tau\}$ ($\alpha, \alpha', \ldots$ to range over Act_τ). We denote by $\mathcal{V}$ the set of random variables ($\xi, \psi, \ldots$ to range over $\mathcal{V}$); $\gamma, \gamma', \ldots$ will denote generic elements in $\mathrm{Act}_\tau \cup \mathcal{V}$. Finally, $Id_\mathcal{P}$ represents the set of process variables.

Definition 2. The set of processes, denoted by $\mathcal{P}$, is given by the following BNF-expression:

$$P ::= \texttt{STOP} \mid X \mid \sum_{i=1}^{n} [p_i]\gamma_i\,;P_i \mid \mathrm{rec}X.P$$

where $X \in Id_{\mathcal{P}}$, for any $1 \leq i \leq n$ we have $\gamma_i \in \mathrm{Act}_\tau \cup \mathcal{V}$, $0 < p_i \leq 1$, and $\sum p_i = 1$.

In the definition of processes, we will usually omit trailing occurrences of STOP. Besides, we will use some syntactic sugar for the case of unary and binary choices.[2] For example, $\gamma_1\,;P_1$ stands for $\sum_{i=1}^{1}[1]\gamma_i\,;P_i$, while $\gamma_1\,;P_1 \boxplus_p \gamma_2\,;P_2$ stands for $\sum_{i=1}^{2}[p_i]\gamma_i\,;P_i$, where $p_1 = p$ and $p_2 = 1-p$.

In the previous definition, STOP denotes the process that cannot execute any action. We have included an n-ary probabilistic choice operator. Let us remark that choices are not resolved in a pure probabilistic way. For example, consider the process $a\,;\texttt{STOP} \boxplus_p b\,;\texttt{STOP}$, and suppose that the environment offers only the action a. In this case, a will be executed with probability 1, regardless the value of p. This point will be clear when we define the testing semantics. For example, the processes $a;\texttt{STOP} \boxplus_p b;\texttt{STOP}$ and $\tau;a;\texttt{STOP} \boxplus_p \tau;b;\texttt{STOP}$ are not testing equivalent. Regarding the terms appearing in a choice, $\alpha\,;P$ (with $\alpha \in \mathrm{Act}_\tau$) denotes a process that performs α and after that behaves as P. Besides, a subterm $\xi\,;P$ (with $\xi \in \mathcal{V}$) indicates that the process P is delayed by a random amount of time according to ξ. Specifically, P will start its execution with a probability p before t units of time have been consumed, where $p = P(\xi \leq t)$. Finally, $\mathrm{rec}X.P$ denotes recursion in the usual way.

We will suppose that all the random variables appearing in the definition of a process are independent. This restriction avoids *side effects*. In particular, this implies that the same random variable cannot appear twice in the definition of a process. Note that this restriction does not imply that we cannot have identically distributed random variables (as long as they have different names). Anyway, for the sake of convenience, we will use sometimes in graphical representations the same random variable in different transitions. For example, two transitions labeled by the same random variable ξ is a shorthand to indicate that these two transitions are labeled by independent random variables ψ_1 and ψ_2 that are identically distributed.

We would like to finish the presentation of our syntax by commenting on two points. First, we have chosen an n-ary probabilistic choice only because the operational semantics is easier to define. As we will comment, we would like to keep *urgency*, that is, if a process may execute a τ action then delays are forbidden. This implies that the probabilities previously associated with those stochastic transitions must be redistributed among the remaining transitions. In our current setting, we only need a simple normalization function; if we use

[2] We use $\boxplus$ to denote the binary choice operator because $+$ denotes addition of random variables.

a binary choice, we need a much more complicated function (it has six cases) that needs two additional predicates (for checking stability and deadlock of the components of the choice). Given the fact that we do not lose expressiveness, we have preferred to keep the operational semantics as simple as possible. We would also like to comment on the absence of a parallel operator. The definition of the (operational) semantics of a parallel operator is straightforward if probability distributions are exponential (because of the *memoryless* property), but this is not the case if distributions are not restricted. There are already several proposals satisfactorily dealing with a parallel operator. Among them, [BG01] presents a language being very close to ours. In their model, there is no probabilistic relation between usual actions but stochastic actions are related by *weights*. Briefly, they deal with the interleaving of stochastic actions by splitting them into two events: Start and termination. This mechanism also works in our setting. Nevertheless, in this paper we have preferred to concentrate on the definition and study of an appropriate testing semantics, which can be adapted to other (possibly more expressive) non-Markovian frameworks, rather than in the definition of a more expressive process algebra. Indeed, dealing with a parallel operator means that our ideas on stochastic testing are more difficult to transmit.

In order to define the operational semantics of processes, transitions are labeled either by an action belonging to Act_τ or by a random variable belonging to $\mathcal{V}$. These transitions have an additional label: A probability. So, a derivation $P \xrightarrow{\gamma}_p P'$ expresses that there exists a transition from P to P' labeled by the action $\gamma \in \mathrm{Act}_\tau \cup \mathcal{V}$, and this transition is performed with probability p. As in most probabilistic models, we need to take into account the different occurrences of the same probabilistic transition.

Example 1. Consider the process $P = \sum_{i=1}^{n} [\frac{1}{n}]a \,;\, P'$. If we do not take care, we have that P has only the transition $P \xrightarrow{a}_{\frac{1}{n}} P'$. So, this process would not be equivalent to $Q = a \,;\, P'$.

There are several standard methods in the literature of probabilistic processes to deal with this problem. For example, in [GSS95] every transition of a term has a unique index, in [YL92] equal transitions are joined (by adding probabilities), and in [NdFL95] multisets of transitions are considered, that is, if a transition can be derived in several ways, each derivation generates a different instance. This last approach will be taken in this paper. For instance, in the previous example we have that the transition $P \xrightarrow{a}_{\frac{1}{n}} P'$ has multiplicity equal to n.

In the definition of the operational semantics (see Fig. 2), we use the auxiliary function $\mathcal{N}_1(P)$. This function computes the *total probability* of a process P to perform actions. This is a normalization function that takes care of keeping the previously commented *urgency* property. So, $\mathcal{N}_1(P)$ returns 1 if P cannot immediately perform τ's; otherwise, $\mathcal{N}_1(P)$ is equal to the addition of the probabilities associated with the actions belonging to Act_τ. The first rule says that if $\gamma \in \mathrm{Act}_\tau$ is one of the *first* actions of a choice, this action may be performed; the probability associated with γ will be *normalized*. The second rule is used for

$$\frac{\gamma_i \in \mathrm{Act}_\tau}{\sum_{i=1}^{n} [p_i]\gamma_i ; P_i \xrightarrow{\gamma_i} \frac{p_i}{\mathcal{N}_1(\sum_{i=1}^{n}[p_i]\gamma_i)} P_i} \qquad \frac{\gamma_i \in \mathcal{V} \wedge \mathcal{N}_1(\sum_{i=1}^{n}[p_i]\gamma_i)=1}{\sum_{i=1}^{n} [p_i]\gamma_i ; P_i \xrightarrow{\gamma_i}_{p_i} P_i} \qquad \frac{P[\mathrm{rec}X.P/X] \xrightarrow{\gamma}_p P'}{\mathrm{rec}X.P \xrightarrow{\gamma}_p P'}$$

$$\mathcal{N}_1(\sum_{i=1}^{n}[p_i]\gamma_i) = \begin{cases} 1 & \text{if } \tau \notin \{\gamma_i \mid 1 \le i \le n\} \\ \sum \{\!\{\, p_i \mid \gamma_i \in \mathrm{Act}_\tau \,\}\!\} & \text{otherwise} \end{cases}$$

Fig. 2. Operational Semantics.

stochastic actions. The side condition assures that no stochastic transition will be allowed if the process may immediately perform τ. Regarding this side condition, let us note that $\mathcal{N}_1$ takes the value 1 only in two cases: Either there does not exist j such that $\tau = \gamma_j$ or for any i we have $\gamma_i \in \mathrm{Act}_\tau$. In the latter case, the first condition of the second rule does not hold. The third rule is standard for CCS-like languages.

We use the following conventions: $P \xrightarrow{\gamma}$ stands for there exist $P' \in \mathcal{P}$ and $p \in (0,1]$ such that $P \xrightarrow{\gamma}_p P'$; we write $P \not\xrightarrow{\gamma}$ if there do not exist such P' and p. We write $P \Longrightarrow_p P'$ if there exist $P_1, \cdots, P_{n-1} \in \mathcal{P}$ and $p_1, \cdots, p_n \in (0,1]$ such that $P \xrightarrow{\tau}_{p_1} P_1 \xrightarrow{\tau}_{p_2} \cdots P_{n-1} \xrightarrow{\tau}_{p_n} P'$ and $p = \prod p_i$ (if $n = 0$, we have $P \Longrightarrow_1 P$). Besides, for any $\gamma \in \mathrm{Act} \cup \mathcal{V}$ and $p \in (0,1]$, $P \overset{\gamma}{\Longrightarrow}_p P'$ denotes that there exist two processes $P_1, P_2 \in \mathcal{P}$ and $p_1, p_2, p_3 \in (0,1]$ such that $P \Longrightarrow_{p_1} P_1 \xrightarrow{\gamma}_{p_2} P_2 \Longrightarrow_{p_3} P'$ and $p = p_1 \cdot p_2 \cdot p_3$. Finally, given a set $A \subseteq \mathrm{Act}_\tau \cup \mathcal{V}$ we write $P \xrightarrow{A}_p$ if we have that $p = \sum \{\!\{\, p' \mid \exists \gamma \in A, P' \in \mathcal{P} : \; P \xrightarrow{\gamma}_p P' \,\}\!\}$; otherwise, we write $P \not\xrightarrow{A}_p$.

3 Stochastic Testing Semantics

In this section we present our stochastic testing semantics. As usual, it is based on the interaction between tested processes and tests. First, we define our set of tests (we consider a set of test identifiers $Id_{\mathcal{T}}$).

Definition 3. The *set of tests*, denoted by $\mathcal{T}$, is given by the following BNF-expression:

$$T ::= \mathtt{STOP} \mid \sum_{i=1}^{n} [p_i]\alpha_i \,; T_i \mid \mathrm{rec}X.T$$

where $X \in Id_{\mathcal{T}}$, for any $1 \le i \le n$ we have $\alpha_i \in \mathrm{Act} \cup \{\omega\}$, $0 < p_i \le 1$, and $\sum p_i = 1$.

The same syntactic sugar that we gave for processes will be also used for tests. We have added a new action ω indicating successful termination of the testing procedure. As we commented in the introduction, we do not allow τ actions in tests. So, a test may perform only either visible actions, belonging to

Act, or the special action ω. We will explain the meaning of our tests by following the black box analogy described in [Mil81], where processes are considered as black boxes with buttons. The test $a\,;T$ corresponds to press the a-button and if it goes down then we continue the experiment with the test T. Regarding *non-probabilistic* tests, a test as $a\,;T \boxplus b\,;T'$ can be explained as pressing two buttons simultaneously. In our model, we do consider probabilistic tests. The test $a\,;T \boxplus_p b\,;T'$ is explained as pressing two buttons at the same time but with *different* strengths.

The operational behaviour of tests is the same as that for processes (considering ω as a *usual* action). Let us remark that the function $\mathcal{N}_1$ (used in Fig. 2 as normalization factor) will always take the value 1. The interaction between a process and a test is modeled by the parallel composition of the tested process P and the test T, denoted by $P \parallel T$. The rules describing how processes and tests interact are given in Fig. 3. We have to make a trade-off between the classical testing framework and the probabilistic framework. In the former, if a test may perform the ω action then the testing procedure may finish. In particular, a test as $a\,;T_1 \boxplus_p \omega\,;T_2$ would behave exactly as the test $\omega\,;\texttt{STOP}$. If we consider the probabilistic framework given in [NdFL95], the testing procedure finishes (with probability 1) only if the tested process is stable. So, we will consider that synchronizations in visible actions can be performed only if the test cannot perform an ω action (this is expressed in the first rule in Fig. 3). If the process may perform either τ or stochastic actions then they are performed (second and third rules, respectively). Finally, if the test can perform ω then the interaction of process and test does so. In order to avoid useless computations, we *cut* the testing procedure by evolving into STOP. This transition is performed with a probability equal to 1 minus the measure of *instability* of the tested process. The side condition assures that a 0 probability transition is not generated. As usually, we have a normalization function. The function $\mathcal{N}_2(P \parallel T)$ computes the sum of the probabilities associated with transitions whose labels belong to $\mathrm{Act}_\tau \cup \mathcal{V}$ such that $P \parallel T$ may perform them.

In the following definition we introduce the notion of *successful* computation. We will also define some concepts on successful computations which will be used when defining the notion of passing a test.

Definition 4. Let P be a process and T be a test. A *computation* C is a sequence of transitions $C = P \parallel T \xrightarrow{\gamma_1} {}_{p_1}P_1 \parallel T_1 \xrightarrow{\gamma_2} {}_{p_2}P_2 \parallel T_2 \xrightarrow{\gamma_3} {}_{p_3} \cdots \xrightarrow{\gamma_n} {}_{p_n}P_n \parallel T_n \cdots$. We say that $P \parallel T$ is the *initial state* of C or C is a computation from $P \parallel T$.

A computation C is *successful* if $P_n \parallel T_n \xrightarrow{\omega} {}_p\texttt{STOP}$ for some $n \geq 0$ and $p > 0$. In this case, we say that $\mathit{length}(C) = n$. We denote by $\mathit{Success}(P \parallel T)$ the multiset of successful computations from $P \parallel T$.

Let $C \in \mathit{Success}(P \parallel T)$. We define the *random variable associated with* C, denoted by $\mathit{random}(C)$, as:

$$\mathit{random}(C) = \begin{cases} \mathit{unit} & \text{if } C = P \parallel T \xrightarrow{\omega} {}_p\texttt{STOP} \\ \mathit{random}(C') & \text{if } C = P \parallel T \xrightarrow{\gamma} {}_pC' \ \wedge\ \gamma \in \mathrm{Act}_\tau \\ \gamma + \mathit{random}(C') & \text{if } C = P \parallel T \xrightarrow{\gamma} {}_pC' \ \wedge\ \gamma \in \mathcal{V} \end{cases}$$

$$\frac{P \xrightarrow{a}_p P',\ T \xrightarrow{a}_q T',\ T \not\xrightarrow{\omega}}{P \| T \xrightarrow[\mathcal{N}_2(P\|T)]{a} P' \| T'} \qquad \frac{P \xrightarrow{\tau}_p P'}{P \| T \xrightarrow[\mathcal{N}_2(P\|T)]{\tau} P' \| T}$$

$$\frac{P \xrightarrow{\xi}_p P'}{P \| T \xrightarrow[\mathcal{N}_2(P\|T)]{\xi} P' \| T} \qquad \frac{T \xrightarrow{\omega},\ P \not\xrightarrow{\mathcal{V}\cup\{\tau\}}_1}{P \| T \xrightarrow{\omega}_{1-instab(P)} \texttt{STOP}}$$

$$instab(P) = \sum \{\!|\ p \mid \exists \gamma \in \mathcal{V} \cup \{\tau\}, P' \in \mathcal{P} :\ P \xrightarrow{\gamma}_p P' \ |\!\}$$

$$\mathcal{N}_2(P \parallel T) = \begin{cases} 1 & \text{if } T \xrightarrow{\omega} \\ \sum \{\!|\ p \cdot q \mid \exists a, P', T' : P \xrightarrow{a}_p P' \wedge T \xrightarrow{a}_q T' \ |\!\} + instab(P) & \text{if } T \not\xrightarrow{\omega} \end{cases}$$

Fig. 3. Interaction between processes and tests.

Let $C \in Success(P \parallel T)$. We define the *probability* of C, denoted by $Prob(C)$, as

$$Prob(C) = \begin{cases} p & \text{if } C = P \parallel T \xrightarrow{\omega}_p \\ p \cdot Prob(C') & \text{if } C = P \parallel T \xrightarrow{\gamma}_p C' \end{cases}$$

First, let us note that we will have a multiset of computations. The random variable $random(C)$ is used to compute the time that C needs to be executed, that is, for any time t we have that $\mathrm{P}(random(C) \leq t)$ gives the probability of executing C before a time t has passed. Finally, $Prob(C)$ cumulates the probabilities of all the decisions taken to obtain that particular computation.

Example 2. Let us consider the computations depicted in Fig. 4, where the symbol $\circledcirc$ represents the successful termination of the testing procedure. Consider $P_1 = (\xi_1 \,;\, a) \boxplus_{\frac{1}{3}} (a \,;\, b)$ and $T_1 = (a \,;\, \omega) \boxplus_{\frac{1}{4}} (a \,;\, b)$. The first graph in Fig. 4 describes the multiset of computations from $P_1 \parallel T_1$.

Consider the recursive process $P_2 = \mathrm{rec}X.(\xi_2 ; P_2 \boxplus_{\frac{1}{2}} a)$ and the tests $T_2 = a;\omega$ and $T_3 = \omega$. In this case, the multisets of computations from $P_2 \parallel T_2$ and $P_2 \parallel T_3$ are both infinite (See the second and third graphs in Fig. 4).

In our model, the notion of passing tests has an additional value as parameter: The time that the process needs to pass the test. A process P passes a test T before a certain amount of time t with a probability p if p is equal to the addition of all the probabilities associated with successful computations from $P \parallel T$ that take a time less than or equal to t to be finished.

Definition 5. Let P be a process, T be a test, $p \in (0, 1]$, and $t \in \mathbb{R}^+$. We say that P *passes* T *before time* t *with probability* p, denoted by $pass_{\leq t}(P, T) = p$, if

$$\sum_{C \,\in\, Success(P \,\parallel\, T)} \mathrm{P}(random(C) \leq t) \cdot Prob(C) = p$$

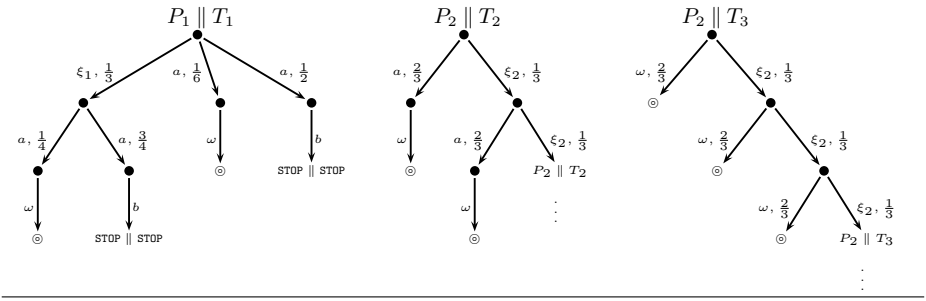


Fig. 4. Examples of computations.

The following result gives an alternative definition of the previous notion. The proof follows straightforward from the fact that successful computations have finite length.

Lemma 1. Let P be a process, T be a test, and $t \in \mathbb{R}^+$. We have that

$$pass_{\leq t}(P,T) = \lim_{n \to \infty} \sum_{\substack{C \in Success(P \parallel T) \\ length(C) < n}} \mathrm{P}(random(C) \leq t) \cdot Prob(C)$$

Definition 6. *(Testing equivalence)* Let P, Q be processes, and $\mathcal{T}' \subseteq \mathcal{T}$ a family of tests. We say that P is *stochastically testing equivalent* to Q with respect to $\mathcal{T}'$, denoted by $P \sim_{\mathcal{T}'} Q$, if for any $T \in \mathcal{T}'$ and any $t \in \mathbb{R}^+$ we have $pass_{\leq t}(P,T) = pass_{\leq t}(Q,T)$. If we consider the whole family of tests $\mathcal{T}$, we write $P \sim_{stoc} Q$ instead of $P \sim_{\mathcal{T}} Q$, and we say that P and Q are stochastically testing equivalent.

In the following example we present some processes that are not stochastically testing equivalent and some tests are given to distinguish them.

Example 3. Let us consider the following processes: $Q_1 = \tau;a \boxplus_p \tau;b$, $Q_2 = a \boxplus_p b$, and $Q_3 = \tau\,;\, a \boxplus_p b$. As it is the case for probabilistic models, they are not equivalent in our semantics. Considering $T = a\,;\omega$, we have that for any $t \in \mathbb{R}^+$, $pass_{\leq t}(Q_1,T) = p$, meanwhile $pass_{\leq t}(Q_2,T) = pass_{\leq t}(Q_3,T) = 1$. Moreover, the test $T' = b;\omega$ shows that Q_2 and Q_3 are not stochastically testing equivalent: For any $t \in \mathbb{R}^+$ we have $pass_{\leq t}(Q_2,T') = 1$ but $pass_{\leq t}(Q_3,T') = 1 - p$.

Consider the processes $R_1 = \xi;a$ and $R_2 = \psi;a$. If ξ and ψ are not identically distributed, then there exists a time $t_1 \in \mathbb{R}^+$ such that $\mathrm{P}(\xi \leq t_1) \neq \mathrm{P}(\psi \leq t_1)$. So, $pass_{\leq t_1}(P,\omega) \neq pass_{\leq t_1}(Q,\omega)$.

Consider $R_3 = \xi\,;\, a$ and $R_4 = a\,;\,\xi$, and suppose that ξ is not distributed as *unit*. This implies that there exists $t_1 \in \mathbb{R}^+$ such that $\mathrm{P}(\xi \leq t_1) = p < 1$. Then these two processes can be distinguished by the test ω, because we have that $pass_{\leq t_1}(R_3,\omega) = p$ and $pass_{\leq t_1}(R_4,\omega) = 1$.

Consider the processes and tests of Example 2. Once we have computed the corresponding set of computations the probability of passing the tests can be

computed. We only need to add the probabilities associated with successful computations. So, we have that for any $t \in \mathbb{R}^+$, $pass_{\leq t}(P_1, T_1) = \frac{1}{12} \cdot \mathrm{P}(\xi_1 \leq t) + \frac{1}{6}$ and $pass_{\leq t}(P_2, T_2) = pass_{\leq t}(P_2, T_3) = \sum_{i=0}^{\infty} (\frac{1}{3})^i \cdot \frac{2}{3} \cdot \mathrm{P}(i \cdot \xi_2 \leq t)$, where $n \cdot \xi$ stands for the addition of ξ with itself n times.

We will finish this section by showing that the set of tests can be reduced. Specifically, we will give a family of test which has the same discriminatory power as the whole family of tests $\mathcal{T}$. First, we can restrict ourselves to finite tests (i.e. non-recursive tests). The proof is made by using an appropriate extension of the technique given in [GN99], where a similar result is given for a probabilistic process algebra.

Lemma 2. Let $\mathcal{T}_f \subseteq \mathcal{T}$ be the set of tests without occurrences of the recursion operator, and P, Q be processes. Then $P \sim_{\mathcal{T}_f} Q$ iff $P \sim_{stoc} Q$.

We will show that the set of tests can be restricted even more. In the following definition, a set of *essential* tests is given.

Definition 7. The set of *essential tests*, denoted by $\mathcal{T}_e$, is given by the following BNF-expression:

$$T ::= \sum_{i=1}^{n} [p_i](a_i \,;\, T_i) \mid \omega \qquad \text{where } T_i = \begin{cases} T & \text{if } i = n \\ \texttt{STOP} & \text{otherwise} \end{cases}$$

where $\{a_1, \ldots, a_n\} \subseteq \mathrm{Act}$, for any $1 \leq i \leq n$ we have $0 < p_i \leq 1$, and $\sum p_i = 1$.

An essential test is either the test ω or a generalized probabilistic choice among a set of visible actions. In the latter case, all the *continuations* except one are equal to `STOP`. Let us remark that similar sets of essential tests appear in [NdFL95,CDSY99]. The proof of this result follows the same pattern as the given in [Núñ96].

Theorem 1. Let P, Q be processes. Then $P \sim_{\mathcal{T}_e} Q$ iff $P \sim_{stoc} Q$.

4 Relation with Other Notions of Testing

In this section we compare our stochastic testing semantics with other testing models. Specifically, we will consider the *may* and *must* notions of testing, a probabilistic testing semantics similar to that of [CDSY99], and the (Markovian) testing semantics of [BC00]. First, we will define a subset of the whole set of processes $\mathcal{P}$.

Definition 8. The set of *probabilistic processes*, denoted by $\mathcal{P}_P$, is given by the BNF-expression

$$P ::= \texttt{STOP} \mid X \mid \sum_{i=1}^{n} [p_i]\alpha_i \,;\, P_i \mid \mathrm{rec}X.P$$

where $X \in Id_{\mathcal{P}}$, for any $1 \leq i \leq n$ we have $\alpha_i \in \mathrm{Act}_\tau$, $0 < p_i \leq 1$, and $\sum p_i = 1$.

Next, we will study the notions of *may* and *must* testing [dNH84,Hen88]. We can define equivalent notions for our language $\mathcal{P}_P$.

Definition 9. Let $P \in \mathcal{P}_P$ and $T \in \mathcal{T}$. We write P *may* T if $Success(P \parallel T) \neq \emptyset$ and P *must* T if for any *maximal* (i.e. that it cannot be extended) computation C we have that $C \in Success(P \parallel T)$. Let $P, Q \in \mathcal{P}_P$. We write $P \sim_{may} Q$ if for any $T \in \mathcal{T}$ we have P *may* T iff Q *may* T. We write $P \sim_{must} Q$ if for any $T \in \mathcal{T}$ we have P *must* T iff Q *must* T.

Note that in the previous definitions we do not use the probabilistic information contained in either the processes or the tests. We can recover these notions of testing in our framework as the following result states (the proof is straightforward).

Lemma 3. Let $P \in \mathcal{P}_P$ and $T \in \mathcal{T}$. We have P *may* T iff $\exists t \in \mathbb{R}^+$ such that $pass_{\leq t}(P, T) > 0$. If P is divergence free, then P *must* T iff $\exists t \in \mathbb{R}^+$ such that $pass_{\leq t}(P, T) = 1$.

Note that, in the previous lemma, the values of t are irrelevant. The following result (whose proof is trivial) states that our testing semantics is a strict refinement of the classical notions for divergence free processes.

Corollary 1. Let $P, Q \in \mathcal{P}_P$. We have that $P \sim_{stoc} Q$ implies $P \sim_{may} Q$. Moreover, if P and Q are divergence free, we also have that $P \sim_{stoc} Q$ implies $P \sim_{must} Q$.

Let us remark that the previous result does not hold for must testing if we consider divergent processes. For example, the (non-probabilistic) processes $\mathrm{rec}X.(a \boxplus \tau \,;\, X)$ and $a \,;\, \mathtt{STOP}$ are not must testing equivalent. However, for any $p \in (0, 1)$ the probabilistic processes $\mathrm{rec}X.(a \boxplus_p \tau \,;\, X)$ and $a \,;\, \mathtt{STOP}$ are stochastically testing equivalent. So, for a process presenting divergent behavior, passing a test with probability 1 is not equivalent to pass it in the must semantics. A more extended discussion on this can be found in [NR99].

The probabilistic testing theory defined in [CDSY99] computes the probability of passing a test as the sum of the probabilities associated with all the successful computations. First, they study a testing semantics where tests are τ-free. Then they study the general case. We will define a (pure) probabilistic testing semantics following the lines of [CDSY99].

Definition 10. Let $P \in \mathcal{P}_P$, $T \in \mathcal{T}$, and $p \in (0, 1]$. We write P *pass*$_p$ T iff $\sum_{C \in Success(P \parallel T)} Prob(C) = p$.

Two processes $P, Q \in \mathcal{P}_P$ are *probabilistically testing equivalent*, denoted by $P \sim_P Q$, if for any test T we have P *pass*$_p$ T iff Q *pass*$_p$ T.

This notion of testing can be easily included in our framework as the following result states. The proof is trivial just taking into account that if $P \in \mathcal{P}_P$ and T is a test, then for any successful computation C from $P \parallel T$ we have $random(C) = unit$.

Lemma 4. Let $P \in \mathcal{P}_P$, $T \in \mathcal{T}$, and $p \in (0,1]$. We have P $pass_p$ T iff $\forall t \in \mathbb{R}^+$ $pass_{\leq t}(P,T) = p$. Moreover, for any $P, Q \in \mathcal{P}_P$ we have $P \sim_P Q$ iff $P \sim_{stoc} Q$.

In [BC00] a Markovian testing theory is presented. Given the fact that our language is very different from theirs, we cannot automatically compare both testing semantics. In the following, we will adapt their notion of testing to our framework. The testing theory presented in [BC00] does not compute additions of random variables. Instead, they consider the *average time* that computations need to be performed. In our case, we can consider the *expected value* of the random variable associated with the execution of a computation. Due to our assumption of independence of random variables, this expected value is equal to the addition of the expected values of the random variables performed along the computation.

Definition 11. Let $P \in \mathcal{P}$ and $T \in \mathcal{T}$. For any $t \in \mathbb{R}^+$ the probability with which P *passes* T in *average time* before time t has passed, denoted by $pass_{rate \leq t}(P,T)$, is defined as

$$pass_{rate \leq t}(P,T) = \sum_{C \in Success(P \parallel T)} Prob(C) \cdot \mathrm{P}(\mathrm{E}[random(\mathrm{C})] \leq \mathrm{t})$$

Two processes $P, Q \in \mathcal{P}$ are *testing equivalent in average time*, denoted by $P \sim_{av} Q$, if for any test T and any $t \in \mathbb{R}^+$, $pass_{rate \leq t}(P,T) = pass_{rate \leq t}(Q,T)$.

The following result trivially follows from the corresponding notions of testing. It is also trivial to see that the reverse implication does not hold. Consider two processes $P_1 = \xi_1 \,;\, \mathtt{STOP}$ and $P_2 = \xi_2 \,;\, \mathtt{STOP}$ such that $\mathrm{E}[\xi_1] = \mathrm{E}[\xi_2]$, but ξ_1 and ξ_2 not identically distributed. We have $P_1 \sim_{av} P_2$ while $P_1 \not\sim_{stoc} P_2$.

Lemma 5. Let $P, Q \in \mathcal{P}_P$. We have $P \sim_{stoc} Q$ implies $P \sim_{av} Q$.

5 Conclusions and Future Work

In this paper we have studied a testing semantics for a class of stochastic processes with general distributions. We have given a set of essential tests. We have also compare our framework with other notions of testing. Regarding future work, we are interested in the study of an axiomatization of our testing semantics. We would also like to present our semantic framework for a more expressive language (containing a parallel operator). We have already used the model defined in [BG01] but the presentation of our testing framework is rather involved.

Acknowledgments. We would like to thank the anonymous referees of this paper for the careful reading and the useful comments.

References

[ABC+94] M. Ajmone Marsan, A. Bianco, L. Ciminiera, R. Sisto, and A. Valenzano. A LOTOS extension for the performance analysis of distributed systems. *IEEE/ACM Transactions on Networking*, 2(2):151–165, 1994.

[BB93] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1993.

[BBG98] M. Bravetti, M. Bernardo, and R. Gorrieri. Towards performance evaluation with general distributions in process algebras. In *CONCUR'98, LNCS 1466*, pages 405–422. Springer, 1998.

[BC00] M. Bernardo and W.R. Cleaveland. A theory of testing for markovian processes. In *CONCUR'2000, LNCS 1877*, pages 305–319. Springer, 2000.

[BG98] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.

[BG01] M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-markov processes. To appear in Theoretical Computer Science, 2001.

[BKLL95] E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):553–565, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Computer Science 18. Cambridge University Press, 1990.

[CDSY99] R. Cleaveland, Z. Dayar, S.A. Smolka, and S. Yuen. Testing preorders for probabilistic processes. *Information and Computation*, 154(2):93–148, 1999.

[Chr90] I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In *CONCUR'90, LNCS 458*, pages 126–140. Springer, 1990.

[CLLS96] R. Cleaveland, I. Lee, P. Lewis, and S.A. Smolka. A theory of testing for soft real-time processes. In *8th International Conference on Software Engineering and Knowledge Engineering*, 1996.

[DKB98] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems. In *Programming Concepts and Methods*, pages 126–147. Chapman & Hall, 1998.

[dNH84] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[EKN99] A. El-Rayes, M. Kwiatkowska, and G. Norman. Solving infinite stochastic process algebra models through matrix-geometric methods. In *7th International Workshop on Process Algebra and Performance Modelling*, pages 41–62, 1999.

[GHR93] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *16th Int. Symp. on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE'93), LNCS 729*, pages 121–146. Springer, 1993.

[GLNP97] C. Gregorio, L. Llana, M. Núñez, and P. Palao. Testing semantics for a probabilistic-timed process algebra. In *4th International AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software, LNCS 1231*, pages 353–367. Springer, 1997.

[GN99] C. Gregorio and M. Núñez. Denotational semantics for probabilistic refusal testing. In *PROBMIV'98, Electronic Notes in Theoretical Computer Science 22*. Elsevier, 1999.

[GSS95] R. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[Han91] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Department of Computer Systems. Uppsala University, 1991.

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

[HHK01] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. To appear in Theoretical Computer Science, 2001.

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

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HR95] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

[HS00] P.G. Harrison and B. Strulo. SPADES – a process algebra for discrete event simulation. *Journal of Logic Computation*, 10(1):3–42, 2000.

[KN98] M. Kwiatkowska and G.J. Norman. A testing equivalence for reactive probabilistic processes. In *EXPRESS'98, Electronic Notes in Theoretical Computer Science 16*. Elsevier, 1998.

[LdF97] L. Llana and D. de Frutos. Denotational semantics for timed testing. In *4th AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software, LNCS 1231*, pages 368–382, 1997.

[LN01] N. López and M. Núñez. A testing theory for generally distributed stochastic processes. Available at: `http://dalila.sip.ucm.es/~natalia/papers/stoctesting.ps.gz`, 2001.

[Low95] G. Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138:315–352, 1995.

[LS91] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[Mil81] R. Milner. A modal characterization of observable machine-behaviour. In *6th CAAP, LNCS 112*, pages 25–34. Springer, 1981.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[NdF95] M. Núñez and D. de Frutos. Testing semantics for probabilistic LOTOS. In *Formal Description Techniques VIII*, pages 365–380. Chapman & Hall, 1995.

[NdFL95] M. Núñez, D. de Frutos, and L. Llana. Acceptance trees for probabilistic processes. In *CONCUR'95, LNCS 962*, pages 249–263. Springer, 1995.

[Neu92] M. Neuts. Two further closure properties of Ph-distributions. *Asia-Pacific Journal of Operational Research*, 9(1):77–85, 1992.

[NR99] M. Núñez and D. Rupérez. Fair testing through probabilistic testing. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XII), and Protocol Specification, Testing, and Verification (XIX)*, pages 135–150. Kluwer Academic Publishers, 1999.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed process, ATP: Theory and application. *Information and Computation*, 114(1):131–178, 1994.

[Núñ96] M. Núñez. *Semánticas de Pruebas para Álgebras de Procesos Probabilísticos*. PhD thesis, Universidad Complutense de Madrid, 1996.

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

[YL92] W. Yi and K.G. Larsen. Testing probabilistic and nondeterministic processes. In *Protocol Specification, Testing and Verification XII*, pages 47–61. North Holland, 1992.

An Algorithm for Quantitative Verification of Probabilistic Transition Systems

Franck van Breugel[1]* and James Worrell[2]**

[1] York University, Department of Computer Science
4700 Keele Street, Toronto, M3J 1P3, Canada
[2] Tulane University, Department of Mathematics
6823 St Charles Avenue, New Orleans LA 70118, USA

Abstract. In an earlier paper we presented a pseudometric on the class of reactive probabilistic transition systems, yielding a quantitative notion of behavioural equivalence. The pseudometric is defined via the terminal coalgebra of a functor based on the Hutchinson metric on probability measures. In the present paper we give an algorithm, based on linear programming, to calculate the distance between two states up to prescribed degree of accuracy.

1 Introduction

It has been argued that notions of exact behavioural equivalence sit uneasily with models of systems which feature quantitative data, like probabilities. Real numbers are ideal entities; computers, and humans for that matter, only deal with approximations. If we only have an approximate description of a system, it makes no sense to ask if any two states behave exactly the same. Even if we have a precise description of a system, we may still want express the idea that two states exhibit *almost* the same behaviour.

In this paper we consider reactive probabilistic transition systems. One of the standard equivalences for such systems is *probabilistic bisimulation*, introduced by Larsen and Skou [12]. Briefly, a probabilistic bisimulation is an equivalence relation on states such that for any two related states their probability of making a transition to any equivalence class is equal. Two states are either bisimilar or they are not bisimilar, and a slight change in the probabilities associated to a system can cause bisimilar states to become non-bisimilar and vice-versa. Consider, for example, the system depicted in the diagram below. The states s_0 and s_1 are only bisimilar if ε is 0. However, the states give rise to almost the same behaviour for very small ε different from 0.

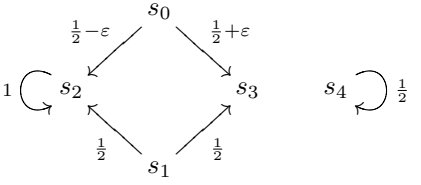
Motivated by such examples, Giacalone, Jou and Smolka [7] defined a *pseudometric* on the states of a (restricted type of) probabilistic transition system. This yields a smooth, *quantitative* notion of behavioural equivalence. A pseudometric differs from an ordinary metric in that different elements, that is, states,

* Supported by Natural Sciences and Engineering Research Council of Canada.
** Supported by the US Office of Naval Research.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 336–350, 2001.
© Springer-Verlag Berlin Heidelberg 2001

can have distance 0. The distance between states, a real number between 0 and 1, can be used to express the similarity of the behaviour of the system started in those states. The smaller the distance, the more alike the behaviour is. In particular, the distance between states is 0 if they are indistinguishable.

In [1], we presented a pseudometric for reactive probabilistic systems. In fact, we introduced a family of pseudometrics, parametric in a constant CF strictly between 0 and 1 (we will discuss the significance of this constant later). For instance, the distance between states s_0 and s_1 is $\text{CF} \cdot \varepsilon$. Also, 0-distance coincides with probabilistic bisimilarity. Our pseudometric is similar to one presented by Desharnais, Gupta, Jagadeesan and Panangaden [3]; for a detailed comparison see Section 7 and [1].



The main contribution of the present paper is to give an algorithm to calculate distances in our pseudometric to a prescribed degree of accuracy. We have implemented the algorithm[1]. As we explain below, the key ingredients of our pseudometric and algorithm are coalgebras, the Hutchinson metric and linear programming.

Many different kinds of transition system can be viewed as *coalgebras*; Rutten [15] provides numerous examples. De Vink and Rutten [16] have shown that probabilistic transition systems correspond to P'-coalgebras, where P' is an endofunctor on the category of 1-bounded complete ultrametric spaces and nonexpansive functions. Furthermore, they have proved that the functor P' is locally contractive. Hence, according to Rutten and Turi's (ultra)metric terminal coalgebra theorem [14], there exists a *terminal* P'-coalgebra. By definition, there is a unique map from an arbitrary P'-coalgebra, that is, a probabilistic transition system, to the terminal P'-coalgebra. De Vink and Rutten have also shown that the *kernel* of this unique map is probabilistic bisimilarity on the states of the probabilistic transition system. That is, two states are mapped to the same element in the terminal P'-coalgebra by the unique map if and only if they are probabilistic bisimilar.

In this paper, we study a variation on the endofunctor P'. Our endofunctor P on the category $\mathcal{CM}et_1$ of 1-bounded complete metric spaces and nonexpansive function is based on the *Hutchinson metric* on probability measures. This metric arises in very different contexts including statistics and fractal geometry, and under different names including the Kantorovich metric and the Wasserstein metric. Like P'-coalgebras, also P-coalgebras can be seen as probabilistic transition systems, as we will show. Furthermore, we observe that the functor

[1] http://www.cs.yorku.ca/~franck

P is locally contractive and hence has a terminal coalgebra. Since the terminal P-coalgebra carries a metric, we can also consider the *metric kernel* of the unique map from a P-coalgebra to the terminal P-coalgebra. This is a pseudometric on the carrier of the P-coalgebra. The distance between two states of a P-coalgebra, that is, a probabilistic transition system, is the distance in the terminal P-coalgebra of their images under the unique map. Since our functor is similar to the one considered by De Vink and Rutten, we still have that two states are bisimilar if and only if they are mapped to the same element in the terminal P-coalgebra and hence have distance 0.

As Rutten and Turi [14] have shown, the unique map from an F-coalgebra to the terminal F-coalgebra, where F is a locally contractive endofunctor on the category $\mathcal{CM}et_1$, can be defined as the unique fixed point $\mathit{fix}\,(\Phi)$ of a function Φ from a complete metric space to itself. Since the functor F is locally contractive, the function Φ is contractive. Hence, according to Banach's fixed point theorem, Φ has a unique fixed point $\mathit{fix}\,(\Phi)$. This fixed point can be approximated by a sequence of functions $(\phi_n)_n$. The function ϕ_0 is an arbitrary function from the F-coalgebra to the terminal F-coalgebra and the other functions are defined by $\phi_n = \Phi\,(\phi_{n-1})$. Not only the metric kernel of the unique map $\mathit{fix}\,(\Phi)$ defines a pseudometric $d_{\mathit{fix}\,(\Phi)}$ on the carrier of the F-coalgebra. Also the metric kernels of the approximations ϕ_n induce pseudometrics d_{ϕ_n}. We will show that the pseudometric $d_{\mathit{fix}\,(\Phi)}$ can be approximated by the pseudometrics d_{ϕ_n}. In particular, to calculate the $d_{\mathit{fix}\,(\Phi)}$-distances to a prescribed degree of accuracy δ, we only have to calculate the $\phi_1, \ldots, \phi_{\log_{\mathrm{CF}}(\delta/2)}$-distances.

Next, we discuss how to compute the distance $d_{\phi_n}(s, s')$, where s and s' are elements of the carrier of the P-coalgebra, that is, states of the probabilistic transition system. We will show that this problem can be reduced to a particular linear programming problem: the *transshipment problem*. The transshipment problem is to find the cheapest way to ship a prescribed amount of a commodity from specified origins to specified destinations through a concrete transportation network. This network is represented by a directed graph. There is a demand for some commodity at some nodes and a supply (or negative demand) of some commodity at other nodes. With each edge, we associate the cost of shipping a unit amount along the edge. For a detailed discussion of this problem and algorithms which can solve this problem in polynomial time we refer the reader to, for example, Chvátal's textbook [2].

2 A Metric Terminal Coalgebra Theorem

In this section, we introduce coalgebras and Rutten and Turi's metric terminal coalgebra theorem [14]. For more details about the theory of coalgebras we refer the reader to, for example, the tutorial [9] of Jacobs and Rutten.

Definition 1. *Let $\mathcal{C}$ be a category. Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. An F-coalgebra consists of an object C in $\mathcal{C}$ together with an arrow $f : C \to F(C)$ in $\mathcal{C}$. The object C is called the carrier. An F-homomorphism from F-coalgebra $\langle C, f\rangle$ to*

F-coalgebra $\langle D, g\rangle$ is an arrow $\phi : C \to D$ in $\mathcal{C}$ such that $F(\phi) \circ f = g \circ \phi$.

$$\begin{array}{ccc} C & \xrightarrow{\phi} & D \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ F(C) & \xrightarrow[F(\phi)]{} & F(D) \end{array}$$

The F-coalgebras and F-homomorphisms form a category. If this category has a terminal object, then this object is called the terminal F-coalgebra.

We restrict our attention to the category $\mathcal{CM}et_1$ of 1-bounded complete metric spaces and nonexpansive functions. A metric space is 1-bounded if all its distances are bounded by 1. A function is nonexpansive if it does not increase any distances. We denote the collection of nonexpansive functions from the space X to the space Y by $X \underset{1}{\to} Y$. This collection can be turned into a metric space by endowing the functions with the supremum metric.

Let c be a constant between 0 and 1. A function is c-contractive if it decreases all distances by at least a factor c.

Definition 2. *A functor $F : \mathcal{CM}et_1 \to \mathcal{CM}et_1$ is locally c-contractive if for all 1-bounded complete metric spaces X and Y, the function $F_{X,Y} : (X \underset{1}{\to} Y) \to (F(X) \underset{1}{\to} F(Y))$ defined by*

$$F_{X,Y}(f) = F(f)$$

is c-contractive.

In the rest of this section, we restrict ourselves to locally c-contractive functors. For these functors, we have

Theorem 1. *There exists a terminal F-coalgebra $\langle fix(F), \iota\rangle$.*

Proof. See [14, Theorem 4.8]. □

For the rest of this section, we fix $\langle X, \mu\rangle$ to be an F-coalgebra. To characterize the unique map from the F-coalgebra $\langle X, \mu\rangle$ to the terminal F-coalgebra $\langle fix(F), \iota\rangle$ we introduce the following function.

Definition 3. *The function $\Phi : (X \underset{1}{\to} fix(F)) \to (X \underset{1}{\to} fix(F))$ is defined by*

$$\Phi(\phi) = \iota^{-1} \circ F(\phi) \circ \mu.$$

$$\begin{array}{ccc} X & \xrightarrow{\phi} & fix(F) \\ {\scriptstyle \mu}\downarrow & & \uparrow{\scriptstyle \iota^{-1}} \\ F(X) & \xrightarrow[F(\phi)]{} & F(fix(F)) \end{array}$$

Since the functor F is locally c-contractive, we have that the function Φ is c-contractive.

Proposition 1. *The function Φ is c-contractive.*

Proof. See proof of [14, Theorem 4.5]. □

Since Φ is a contractive function from a complete metric space to itself, we can conclude from Banach's theorem that it has a unique fixed point $\mathit{fix}\,(\Phi)$.

Proposition 2. *The function $\mathit{fix}\,(\Phi)$ is the unique F-homomorphism from the F-coalgebra $\langle X, \mu\rangle$ to the terminal F-coalgebra $\langle \mathit{fix}\,(F), \iota\rangle$.*

Proof. See proof of [14, Theorem 4.5]. □

We conclude this section by showing that the unique map $\mathit{fix}\,(\Phi)$ can be approximated by the maps ϕ_n.

Definition 4. *Let $\phi_0 : X \underset{1}{\to} \mathit{fix}\,(F)$ be some constant function. For $n > 0$, the function $\phi_n : X \underset{1}{\to} \mathit{fix}\,(F)$ is defined by*

$$\phi_n = \Phi\,(\phi_{n-1}).$$

Proposition 3. *For all $n \geq 0$,*

$$d_{X \underset{1}{\to} \mathit{fix}\,(F)}\,(\phi_n, \mathit{fix}\,(\Phi)) \leq c^n.$$

Proof. By induction on n. □

3 Metric Kernels

Our pseudometric on the states of a probabilistic transition system will be defined as the so-called metric kernel induced by the unique map from the probabilistic transition system, viewed as a coalgebra, to the terminal coalgebra. In this section, we introduce metric kernels. Furthermore, we show that the metric kernel induced by $\mathit{fix}\,(\Phi)$ can be approximated by the metric kernels induced by ϕ_n.

A function ϕ from the space X to the space $\mathit{fix}\,(F)$ defines a distance function d_ϕ on X. We call this distance function the metric kernel induced by ϕ. The distance between x_1 and x_2 in X is defined as the distance of their ϕ-images in the metric space $\mathit{fix}\,(F)$.

Definition 5. *Let $\phi \in X \underset{1}{\to} \mathit{fix}\,(F)$. The distance function $d_\phi : X \times X \to [0,1]$ is defined by*

$$d_\phi\,(x_1, x_2) = d_{\mathit{fix}\,(F)}\,(\phi\,(x_1), \phi\,(x_2)).$$

One can easily verify that the metric kernel d_ϕ is a pseudometric. Note that x_1 and x_2 have distance 0 only if they are mapped by ϕ to the same element in $fix\,(F)$.

The pseudometric $d_{fix\,(\Phi)}$ can be approximated by the pseudometrics d_{ϕ_n} as is shown in

Proposition 4. *For all $n \geq 0$ and $x_1,\ x_2 \in X$,*

$$|d_{\phi_n}\,(x_1, x_2) - d_{fix\,(\Phi)}\,(x_1, x_2)| \leq 2 \cdot c^n.$$

Proof.

$$\begin{aligned}
&|d_{\phi_n}\,(x_1, x_2) - d_{fix\,(\Phi)}\,(x_1, x_2)| \\
&= |d_{fix\,(F)}\,(\phi_n\,(x_1), \phi_n\,(x_2)) - d_{fix\,(F)}\,(fix\,(\Phi)\,(x_1), fix\,(\Phi)\,(x_2))| \\
&\leq d_{fix(F)}(\phi_n(x_1), fix(\Phi)(x_1)) + d_{fix(F)}(\phi_n(x_2), fix(\Phi)(x_2)) \quad \textit{[triangle inequality]} \\
&\leq 2 \cdot d_{X \underset{1}{\to} fix\,(F)}\,(\phi_n, fix\,(\Phi)) \\
&\leq 2 \cdot c^n \quad \textit{[Proposition 3]}
\end{aligned}$$

To compute the $d_{fix\,(\Phi)}$-distances up to accuracy δ, it suffices to calculate the $d_{\phi_{\lceil \log_c (\delta/2) \rceil}}$-distances.

Proposition 5. *For all $0 < \delta < 1$ and $x_1,\ x_2 \in X$,*

$$|d_{\phi_{\lceil \log_c (\delta/2) \rceil}}\,(x_1, x_2) - d_{fix\,(\Phi)}\,(x_1, x_2)| \leq \delta.$$

Proof.

$$\begin{aligned}
&|d_{\phi_{\lceil \log_c (\delta/2) \rceil}}\,(x_1, x_2) - d_{fix\,(\Phi)}\,(x_1, x_2)| \\
&\leq 2 \cdot c^{\lceil \log_c (\delta/2) \rceil} \quad \textit{[Proposition 4]} \\
&\leq 2 \cdot c^{\log_c (\delta/2)} \\
&= \delta.
\end{aligned}$$

4 The Hutchinson Functor

In this section we introduce the probabilistic analogs of transition systems and bisimulation. Then we show that probabilistic transition systems can be seen as coalgebras of an endofunctor P on $\mathcal{CM}et_1$ based on the Hutchinson metric on probability measures. Since P is locally contractive it has a terminal coalgebra by Theorem 1. The metric kernel of the unique map to the terminal P-coalgebra defines a pseudometric on the carrier of a P-coalgebra and hence on the states of a probabilistic transition system. For simplicity we only consider unlabelled transitions, though all our results generalize to the labelled case.

Definition 6. *A probabilistic transition system consists of a finite set S of states together with a transition function $\pi : S \times S \to [0, 1]$ such that, for each $s \in S$, $\sum_{s' \in S} \pi\,(s, s') \leq 1$.*

This is the so-called reactive model of Larsen and Skou [12]. The transition function π is a conditional sub-probability distribution determining the reaction of the system to an action by the environment. $\pi(s, s')$ is the probability that the system ends up in state s' given that it was in state s before the action. We impose the restriction $\sum_{s' \in S} \pi(s, s') \leq 1$ instead of the more common, but also more restrictive, condition $\sum_{s' \in S} \pi(s, s') = 1$ or 0—the latter corresponding to refusal. We interpret $1 - \sum_{s' \in S} \pi(s, s')$ as the probability that the system refuses the action in state s. To simplify our presentation we add a special state $\mathbf{0}$ for refusal: $\pi(s, \mathbf{0}) = 1 - \sum_{s' \in S} \pi(s, s')$.

Larsen and Skou adapted bisimulation for probabilistic transition systems as follows.

Definition 7. *Let $\langle S, \pi \rangle$ be a probabilistic transition system. An equivalence relation $\mathcal{R}$ on the set of states S is a probabilistic bisimulation if $s_1 \mathcal{R} s_2$ implies $\sum_{s' \in E} \pi(s_1, s') = \sum_{s' \in E} \pi(s_2, s')$ for all $\mathcal{R}$-equivalence classes E. States s_1 and s_2 are probabilistic bisimilar if $s_1 \mathcal{R} s_2$ for some probabilistic bisimulation $\mathcal{R}$.*

In [8], Hutchinson introduced a metric on the set of Borel probability measures on a metric space. We restrict ourselves to spaces in which the distances are bounded by 1, since they serve our purpose. Let X be a 1-bounded metric space. We denote the set of Borel probability measures on X by $M(X)$. The Hutchinson distance on $M(X)$ is introduced in

Definition 8. *The distance function $d_{M(X)} : M(X) \times M(X) \to [0, 1]$ is defined by*

$$d_{M(X)}(\mu_1, \mu_2) = \sup \left\{ \int_X f d\mu_1 - \int_X f d\mu_2 \mid f \in X \underset{1}{\to} [0, \infty) \right\}.$$

For a proof that $d_{M(X)}$ is a 1-bounded metric, we refer the reader to, for example, Edgar's textbook [5, Proposition 2.5.14]. In the rest of this paper, we focus on Borel probability measures which are completely determined by their values for the compact subsets of the space X.

Definition 9. *A Borel probability measure μ on X is tight if for all $\varepsilon > 0$ there exists a compact subset K_ε of X such that $\mu(X \setminus K_\varepsilon) < \varepsilon$.*

Under quite mild conditions on the space, for example, completeness and separability, every measure is tight (see, for example, Parthasarathy's textbook [13, Theorem II.3.2]). In particular, all probabilistic transition systems can be represented using tight measures as we will see in Example 1. We denote the set of tight Borel probability measures on X by $M_t(X)$, and consider it as a metric space with the Hutchinson distance. We are interested in tight measures because of the following

Theorem 2. *X is complete if and only if $M_t(X)$ is complete.*

Proof. See, for example, [5, Theorem 2.5.25]. □

In [1] we show that M_t can be extended to a locally nonexpansive endofunctor on the category $\mathcal{CM}et_1$ by defining $M_t(f) : M_t(X) \to M(Y)$ by $M_t(f)(\mu) = \mu \circ f^{-1}$, where $f : X \to Y$ is nonexpansive.

Now, we are ready to present the functor P. But first we introduce the functor T which models refusal:

$$T = \mathbf{1} + \text{CF} \cdot - : \mathcal{CM}et_1 \to \mathcal{CM}et_1,$$

where $\mathbf{1}$ is the terminal object[2] functor, $+$ is the coproduct[3] functor, and $\text{CF} \cdot$ is the scaling[4] functor. The functor P is defined by

$$P = M_t \circ T : \mathcal{CM}et_1 \to \mathcal{CM}et_1.$$

Every probabilistic transition system can be seen as a P-coalgebra as is demonstrated in

Example 1. Let $\langle S, \pi \rangle$ be a probabilistic transition system. We endow the set of states S with the discrete metric. Consequently, every subset of the 1-bounded complete metric space $T(S)$ is a Borel set. For every state s, the Borel probability measure μ_s is the discrete Borel probability measure determined by

$$\begin{aligned} \mu_s(\mathbf{1}) &= \pi(s, \mathbf{0}) \\ \mu_s(\{s'\}) &= \pi(s, s') \end{aligned}$$

Obviously, the measure μ_s is tight. Because S is endowed with the discrete metric, the function μ mapping the state s to the measure μ_s is nonexpansive. Hence, every probabilistic transition system can be viewed as a P-coalgebra.

Since the functor $\text{CF} \cdot$ is locally CF-contractive and the functors $+$ and M_t are locally nonexpansive, the functor P is locally CF-contractive. Thus, according to Theorem 1, there exists a terminal P-coalgebra. Our pseudometric on a probabilistic transition system is defined as the metric kernel $d_{fix(\Phi)}$ where $fix(\Phi)$ is the unique map from the probabilistic transition system, viewed as a P-coalgebra, to the terminal P-coalgebra. In this pseudometric, states have distance 0 only if they are probabilistic bisimilar.

Proposition 6. *Two states s_1 and s_2 are probabilistic bisimilar if and only if $d_{fix(\Phi)}(s_1, s_2) = 0$.*

[2] The terminal object of $\mathcal{CM}et_1$ is the singleton space $\mathbf{1}$ whose single element we denote by $\mathbf{0}$.

[3] The coproduct object of the objects X and Y in $\mathcal{CM}et_1$ is the disjoint union of the sets underlying the spaces X and Y endowed with the metric

$$d_{X+Y}(v, w) = \begin{cases} d_X(v, w) & \text{if } v \in X \text{ and } w \in X \\ d_Y(v, w) & \text{if } v \in Y \text{ and } w \in Y \\ 1 & \text{otherwise.} \end{cases}$$

[4] The scaling by $\text{CF} \cdot$ of an object in $\mathcal{CM}et_1$ leaves the set unchanged and multiplies all distances by CF.

More generally, the distance between states is a trade-off between the depth of observations needed to distinguish the states and the amount each observation differentiates the states. The relative weight given to these two factors is determined by the constant CF lying between 0 and 1: the smaller the value of CF the greater the discount on observations made at greater depth.

For the system depicted in the introduction, the distances are given in the table below.

	s_0	s_1	s_2	s_3	s_4
s_0	0				
s_1	$\varepsilon \cdot \mathrm{CF}$	0			
s_2	$\mathrm{CF} \cdot (\frac{1}{2} + \varepsilon)$	$\frac{1}{2} \cdot \mathrm{CF}$	0		
s_3	1	1	1	0	
s_4	$\frac{1}{2} + \frac{\mathrm{CF}}{4}$	$\frac{1}{2} + \frac{\mathrm{CF}}{4}$	$\frac{1}{2-\mathrm{CF}}$	$\frac{1}{2}$	0

5 Metric Kernels for P-Coalgebras

Our pseudometric on a probabilistic transition system is defined as the metric kernel $d_{\mathit{fix}\,(\Phi)}$ where $\mathit{fix}\,(\Phi)$ is the unique map from the probabilistic transition system, viewed as a P-coalgebra, to the terminal P-coalgebra. As we have already seen in Section 3, $d_{\mathit{fix}\,(\Phi)}$ can be approximated by the metric kernels d_{ϕ_n}. In this section, we present a characterization of the pseudometrics d_{ϕ_n} for P-coalgebras. Furthermore, we will show that the d_{ϕ_n}-distances are smaller than or equal to the $d_{\mathit{fix}\,(\Phi)}$-distances.

To prove our characterizations, we need the following

Proposition 7. *Let $\phi \in X \underset{1}{\to} \mathit{fix}\,(P)$. Then composition with $T\,(\phi)$ induces a surjection between $T\,(\mathit{fix}\,(P)) \underset{1}{\to} [0,\infty)$ and $T\,\langle X, d_\phi\rangle \underset{1}{\to} [0,\infty)$.*

Proof. By a slight abuse of notation, ϕ may be regarded as an isometric embedding of the pseudometric space $\langle X, d_\phi\rangle$ in $\mathit{fix}\,(P)$. Thus $T\,(\phi)$ is an isometric embedding of $T\,\langle X, d_\phi\rangle$ into $T\,(\mathit{fix}\,(P))$. Now [11, Corollary on page 162] tells us that any nonexpansive map $f : T\,\langle X, d_\phi\rangle \to [0,\infty)$ has an extension $g : T\,(\mathit{fix}\,(P)) \to [0,\infty)$ in the sense that $g \circ T\,(\phi) = f$. □

We can characterize the pseudometric d_{ϕ_n} on the carrier of a P-coalgebra $\langle X, \mu\rangle$ as follows.

Theorem 3. *For all x_1, $x_2 \in X$,*

$$d_{\phi_0}\,(x_1, x_2) = 0.$$

For all $n > 0$ and x_1, $x_2 \in X$,

$$d_{\phi_n}\,(x_1, x_2) = \sup\left\{ \int_{T\,(X)} g\, d\mu_{x_1} - \int_{T\,(X)} g\, d\mu_{x_2} \mid g \in T\,\langle X, d_{\phi_{n-1}}\rangle \underset{1}{\to} [0,\infty) \right\}.$$

Proof. Obviously, $d_{\phi_0}(x_1, x_2) = d_{fix(P)}(\phi_0(x_1), \phi_0(x_2)) = 0$, *since* ϕ_0 *is a constant function. Furthermore, for all* $n > 0$ *we have*

$$
\begin{aligned}
& d_{\phi_n}(x_1, x_2) \\
&= d_{fix(P)}(\phi_n(x_1), \phi_n(x_2)) \\
&= d_{fix(P)}(\Phi(\phi_{n-1})(x_1), \Phi(\phi_{n-1})(x_2)) \\
&= d_{fix(P)}((\iota^{-1} \circ P(\phi_{n-1}) \circ \mu)(x_1), (\iota^{-1} \circ P(\phi_{n-1}) \circ \mu)(x_2)) \quad \textit{[Definition 3]} \\
&= d_{P(fix(P))}((P(\phi_{n-1}) \circ \mu)(x_1), (P(\phi_{n-1}) \circ \mu)(x_2)) \quad \textit{[}\iota\textit{ is isometric]} \\
&= \sup \left\{ \int_{T(fix(P))} f\, d((P(\phi_{n-1}) \circ \mu)(x_1)) - \right. \\
&\qquad \left. \int_{T(fix(P))} f\, d((P(\phi_{n-1}) \circ \mu)(x_2)) \mid f \in T(fix(P)) \underset{1}{\to} [0, \infty) \right\} \\
&= \sup \left\{ \int_{T(X)} (f \circ T(\phi_{n-1}))\, d\mu_{x_1} - \right. \\
&\qquad \left. \int_{T(X)} (f \circ T(\phi_{n-1}))\, d\mu_{x_2} \mid f \in T(fix(P)) \underset{1}{\to} [0, \infty) \right\} \\
&= \sup \left\{ \int_{T(X)} g\, d\mu_{x_1} - \int_{T(X)} g\, d\mu_{x_2} \mid g \in T\langle X, d_{\phi_{n-1}} \rangle \underset{1}{\to} [0, \infty) \right\} \quad \textit{[Prop. 7]}
\end{aligned}
$$

□

We conclude this section with a proof that the d_{ϕ_n}-distances are smaller than or equal to the $d_{fix(\Phi)}$-distances.

Proposition 8. *For all* $n \geq 0$,

$$d_{\phi_n} \leq d_{\phi_{n+1}}.$$

Proof. By induction on n. *The case* $n = 0$ *is trivial. Let* $n > 0$. *For all* x_1, $x_2 \in X$,

$$
\begin{aligned}
& d_{\phi_n}(x_1, x_2) \\
&= \sup \left\{ \int_{T(X)} g d\mu_{x_1} - \int_{T(X)} g d\mu_{x_2} \mid g \in T\langle X, d_{\phi_{n-1}} \rangle \underset{1}{\to} [0, \infty) \right\} \quad \textit{[Theorem 3]} \\
&\leq \sup \left\{ \int_{T(X)} g\, d\mu_{x_1} - \int_{T(X)} g\, d\mu_{x_2} \mid g \in T\langle X, d_{\phi_n} \rangle \underset{1}{\to} [0, \infty) \right\} \\
&\qquad \textit{[by induction } d_{\phi_{n-1}} \leq d_{\phi_n}\textit{, so } T\langle X, d_{\phi_{n-1}} \rangle \underset{1}{\to} [0, \infty) \subseteq T\langle X, d_{\phi_n} \rangle \underset{1}{\to} [0, \infty)\textit{]} \\
&= d_{\phi_{n+1}}(x_1, x_2) \quad \textit{[Theorem 3]}
\end{aligned}
$$

□

Corollary 1. *For all* $n \geq 0$, $d_{\phi_n} \leq d_{fix(\Phi)}$.

6 The Algorithm

Suppose that the P-coalgebra considered in the previous section represents a probabilistic transition system $\langle S, \pi \rangle$, where $S = \{s_0, \ldots, s_{N-1}\}$. We show that the calculation of $d_{\phi_n}(s_p, s_q)$ can be reduced to the transshipment problem. First we introduce some notation. For $0 \leq i, j < N$ we define $c_{ij} = \text{CF} \cdot d_{\phi_{n-1}}(s_i, s_j)$, $c_{iN} = 1$, $c_{Nj} = 1$ and $c_{NN} = 0$. Also we define $\nu_i = \pi(s_p, s_i)$ and $\rho_i = \pi(s_q, s_i)$ for $0 \leq i < N$. Finally, we set $\nu_N = \pi(s_p, \mathbf{0})$ and $\rho_N = \pi(s_q, \mathbf{0})$.

Since integration against discrete measures reduces to summation, according to Theorem 3 to calculate $d_{\phi_n}(s_p, s_q)$ we need to

$$\begin{array}{lll} \text{Maximize}^5 & \displaystyle\sum_{i=0}^{N} \alpha_i \nu_i - \sum_{i=0}^{N} \alpha_i \rho_i & \\ \text{subject to} & \alpha_i - \alpha_j \leq c_{ij} & 0 \leq i, j \leq N \\ & 0 \leq \alpha_i & 0 \leq i \leq N. \end{array}$$

The above is a linear programming problem. In the following analysis we use a few notions from the theory of linear programming; these are explained in numerous texts, including Chvátal's textbook [2].

Next, we transform the problem to one with the same optimal solution. We add extra dimensions to the feasible region by introducing new decision variables β_i for $0 \leq i \leq N$. The constraints in the transformed problem ensure that $\beta_i = \alpha_i$ (since $c_{ii} = 0$) for all $0 \leq i \leq N$. In fact, it is easy to see that the vector $(\alpha_0, \ldots, \alpha_N, \beta_0, \ldots, \beta_N)^T$ satisfies the constraints of the transformed problem iff $(\alpha_0, \ldots, \alpha_N)^T$ satisfies the constraints of the original problem and is equal to $(\beta_0, \ldots, \beta_N)^T$.

$$\begin{array}{lll} \text{Maximize} & \displaystyle\sum_{i=0}^{N} \alpha_i \nu_i - \sum_{i=0}^{N} \beta_i \rho_i & \\ \text{subject to} & \alpha_i - \beta_j \leq c_{ij} & 0 \leq i, j \leq N \\ & \beta_i - \alpha_i \leq 0 & 0 \leq i \leq N \\ & 0 \leq \alpha_i, \beta_j & 0 \leq i, j \leq N. \end{array}$$

Dualizing the above (primal) problem yields:

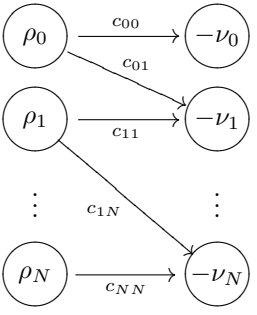
$$\begin{array}{lll} \text{Minimize} & \displaystyle\sum_{i,j=0}^{N} c_{ij} \lambda_{ij} & \qquad (1) \\ \text{subject to} & \sum_{i=0}^{N} \lambda_{ij} - \gamma_j = \nu_j & 0 \leq j \leq N \\ & \sum_{j=0}^{N} \lambda_{ij} - \gamma_i = \rho_i & 0 \leq i \leq N \\ & 0 \leq \lambda_{ij}, \gamma_i & 0 \leq i, j \leq N. \end{array}$$

[5] In general one can show from compactness arguments that the supremum in the statement of Theorem 3 is attained. The fact that the supremum is attained in this particular instance also follows from the theory of linear programming.

The duality theorem of linear programming tells us that the dual problem has an optimal solution with the same value as the optimal solution of the primal problem. Next we show that the dual problem can be simplified to:

$$\begin{aligned} \text{Minimize } & \sum_{i,j=0}^{N} c_{ij}\lambda_{ij} \\ \text{subject to } & \textstyle\sum_{i=0}^{N} \lambda_{ij} = \nu_j \quad 0 \le j \le N \\ & \textstyle\sum_{j=0}^{N} \lambda_{ij} = \rho_i \quad 0 \le i \le N \\ & 0 \le \lambda_{ij} \qquad\quad 0 \le i, j \le N. \end{aligned}$$

A visualization of the above problem is to find the minimum transshipment cost over the following network of sources (on the left) and sinks (on the right), where each source is connected to each sink via an edge is labelled with a cost.



The problem (1) entails finding the minimum transshipment cost over the network above, augmented with an edge of cost zero *from* the node labelled ν_i *to* the node labelled ρ_i for $0 \le i \le N$. But the triangle inequality for d_{ϕ_n} says that $c_{ik} \le c_{ij} + c_{jk}$ for all i, j, k; this means that it is never more expensive to send units directly from a given source to a given sink, rather than indirectly. It is not difficult to see that, as a consequence, the last two minimization problems have the same optimal value.

Finally, we are in a position to present our algorithm for calculating the pseudometric $d_{fix\,(\Phi)}$ on $\langle S, \pi \rangle$ up to a prescribed degree of accuracy δ. The algorithm iteratively calculates d_{ϕ_n}, with the value of $d_{\phi_n}(s_i, s_j)$ being stored in $dist_{ij}$. By Proposition 5, $\lceil \log_{\mathrm{CF}}(\delta/2) \rceil$ cycles of the main loop suffice to get within δ of $d_{fix\,(\Phi)}$. Also, recall from Corollary 1 that the d_{ϕ_n} approximate $d_{fix\,(\Phi)}$ from below.

STEP 1 (Initialization)
We initialize the distance matrix by setting $dist_{pq} := 0$ for $0 \le p, q < N$. The main body of the algorithm also uses an $N + 1$ by $N + 1$ matrix *cost*, and we initialize some of the entries of this matrix thus: $cost_{pN} = 1$ and $cost_{Nq} = 1$ for $0 \le p, q < N$, and $cost_{NN} = 0$ (these values never change during the execution of the algorithm).

STEP 2 (Main loop)

For $n = 0$ to $\lceil \log_{\mathrm{CF}}(\delta/2) \rceil$ do
  For $p, q = 0$ to $N - 1$ do
    $cost_{pq} := \mathrm{CF} \cdot dist_{pq}$
  For $p, q = 0$ to $N - 1$ do

$$dist_{pq} := \{ \text{ minimum value of } \sum_{i,j=0}^{N} cost_{ij} \cdot \lambda_{ij}$$

subject to:

$$\sum_{i=0}^{N} \lambda_{ij} = \pi(x_q, x_j) \quad 0 \leq j \leq N$$

$$\sum_{j=0}^{N} \lambda_{ij} = \pi(x_p, x_i) \quad 0 \leq i \leq N$$

$$0 \leq \lambda_{ij} \quad 0 \leq i, j \leq N \ \}.$$

The presentation above is aimed at clarity. There is an obvious redundancy in that the distance matrix is always symmetric and has 0's along its diagonal, so we only ever need to calculate $dist_{pq}$ for $p < q$.

7 Related and Future Work

In this paper, we considered probabilistic transition systems without labels to simplify the presentation. However, all our results can easily be generalized to a setting with labels. The coalgebras of the functor

$$L \to P : \mathcal{CM}et_1 \to \mathcal{CM}et_1,$$

where L is the finite set of labels, represent labelled probabilistic transition systems. To compute the d_{ϕ_n}-distance between states of a labelled system, for each label we consider only the transitions with that label and compute the d_{ϕ_n}-distance between the states, and take the maximum of all the computed distances.

We see our pseudometric as a measure of the behavioural proximity of states. Now we very briefly outline how this view is sound with respect to the process testing scenario considered by Larsen and Skou in [12]. There one has a class of tests, seen as corresponding to 'button pressing' experiments on processes. A remarkable feature of the testing framework is that the experimenter is allowed to take a finite number of copies of a process and independently test each copy. Associated to each test t there is a set of possible observations O_t, such that a state s of a probabilistic transition system induces a probability distribution $P_{t,s}$ over O_t. For $e \in O_t$, $P_{t,s}(e)$ is the probability of observing e when the test t is performed on s. Larsen and Skou showed that two states are bisimilar just in case they induce the same probability distribution over O_t for each test t.

For our part we can show that for a given test-observation pair (t, e), there is a constant $k_{t,e}$ such that

$$|P_{t,s_1}(e) - P_{t,s_2}(e)| \leq k_{t,e} \cdot d_{fix\,(\Phi)}(s_1, s_2)$$

for states s_1, s_2 of a probabilistic transition system.

The presence of the constant $k_{t,e}$ is partly due to the contraction factor CF. Since our pseudometric discounts the future, $k_{t,e}$ is greater for deeper observations e. The other determinant of $k_{t,e}$ is how much t employs the copying facility.

In [3], Desharnais et al. presented a pseudometric for probabilistic transition systems. Their pseudometric is defined by means of a real-valued modal logic where, in particular, the modal connective is interpreted by integration. Also in their setting, states have distance 0 if and only if they are probabilistic bisimilar. In [1] we showed that by adding negation[6] to their logic one recovers the pseudometric $d_{fix\,(\Phi)}$. We also argued that this leads to distances which are more intuitive.

Desharnais et al. present an algorithm to calculate distances in their metric up to a prescribed degree of accuracy. The algorithm involves the generation of a representative set of formulas of their logic. They only consider formulas with a restricted number of nested occurrences of the modal connective. This corresponds to our approximation of $d_{fix\,(\Phi)}$ by d_{ϕ_n}. Both restrict the depth at which observations are considered. Their algorithm calculates the distances in exponential time, whereas our algorithm computes them in polynomial time. Furthermore, it is not clear to us whether their algorithm can be adapted (in a straightforward manner) to the logic with negation.

Many process combinators, like parallel composition, prefixing and probabilistic choice, can be shown to be nonexpansive with respect to our pseudometric. This quantitative analogue of congruence allows for compositional verification (see also [3,7]).

In the present paper we have been concerned with discrete, indeed finite state, probabilistic transition systems. However, as we will show elsewhere, the terminal coalgebra of the endofunctor P also serves as a domain in which one can interpret continuous systems, in particular the labelled Markov processes of [4]. This paves the way for metric versions of some of the domain theoretic results from that paper.

The transshipment network in Section 6 is very similar to one occurring in the proof of the 'splitting lemma' by Jones and Plotkin in [10], except there one has a network whose edges have capacities rather than costs. In fact, hiding in Section 6 there is a splitting lemma for the Hutchinson metric which can be used to characterize $M(X)$ as a free algebra. Again, details will be presented elsewhere. The same network also appears in the proof that the functor of De Vink and Rutten preserves weak pullbacks, cf. [16].

[6] In a draft version, but not in the final version, of [4] negation was considered.

Acknowledgements. We would like to thank the referees for their suggestions to improve the presentation of the final version. The first author would like to thank Jeff Edmonds for stimulating discussions about linear programming.

References

1. F. van Breugel and J. Worrell. Towards Quantitative Verification of Probabilistic Transition Systems. To appear in *Proceedings of 28th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science*, Crete, July 2001. Springer-Verlag.
2. V. Chvátal. *Linear Programming.* W.H. Freeman and Company, New York/San Francisco, 1983.
3. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for Labeled Markov Systems. In J.C.M. Baeten and S. Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 258–273, Eindhoven, August 1999. Springer-Verlag.
4. J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Approximating Labeled Markov Processes. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 95–106, Santa Barbara, June 2000. IEEE.
5. G.A. Edgar. *Integral, Probability, and Fractal Measures.* Springer-Verlag, 1998.
6. M. Giry. A Categorical Approach to Probability Theory. In B. Banaschewski, editor, *Proceedings of the International Conference on Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85, Ottawa, August 1981. Springer-Verlag.
7. A. Giacalone, C.-C. Jou, and S.A. Smolka. Algebraic Reasoning for Probabilistic Concurrent Systems. In *Proceedings of the IFIP WG 2.2/2.3 Working Conference on Programming Concepts and Methods*, pages 443–458, Sea of Gallilee, April 1990. North-Holland.
8. J.E. Hutchinson. Fractals and Self Similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.
9. B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the EATCS*, 62, June 1997.
10. C. Jones and G.D. Plotkin. A Probabilistic Powerdomain of Evaluations. In *Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science*, pages 186–195, California, June 1989. IEEE.
11. F.W. Lawvere. Metric Spaces, Generalized Logic, and Closed Categories. *Rendiconti del Seminario Matematico e Fisico di Milano*, 43:135–166, 1973.
12. K.G. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Information and Computation*, 94(1):1–28, September 1991.
13. K.R. Parthasarathy. *Probability Measures on Metric Spaces.* Academic Press, 1967.
14. D. Turi and J.J.M.M. Rutten. On the Foundations of Final Semantics: non-standard sets, metric spaces, partial orders. *Mathematical Structures in Computer Science*, 8(5):481-540, October 1998.
15. J.J.J.M. Rutten. Universal Coalgebra: a Theory of Systems. *Theoretical Computer Science*, 249(1):3–80, October 2000.
16. E.P. de Vink and J.J.M.M. Rutten. Bisimulation for Probabilistic Transition Systems: a Coalgebraic Approach. *Theoretical Computer Science*, 221(1/2):271–293, June 1999.

Compositional Methods for Probabilistic Systems*

Luca de Alfaro, Thomas A. Henzinger, and Ranjit Jhala

Electrical Engineering and Computer Sciences, University of California, Berkeley
{dealfaro,tah,jhala}@eecs.berkeley.edu

Abstract. We present a compositional trace-based model for probabilistic systems. The behavior of a system with probabilistic choice is a stochastic process, namely, a probability distribution on traces, or "bundle." Consequently, the semantics of a system with both nondeterministic and probabilistic choice is a set of bundles. The bundles of a composite system can be obtained by combining the bundles of the components in a simple mathematical way. Refinement between systems is bundle containment. We achieve assume-guarantee compositionality for bundle semantics by introducing two scoping mechanisms. The first mechanism, which is standard in compositional modeling, distinguishes inputs from outputs and hidden state. The second mechanism, which arises in probabilistic systems, partitions the state into probabilistically independent regions.

1 Introduction

A system model is *compositional* if the model of a composite system can be obtained by composing the models of the components. Compositionality comes in two flavors: *shallow* and *deep.* Shallow compositionality is essentially a syntactic notion: given two components P and Q, we can construct their composition $P\|Q$, but the semantics of this composition is not directly related to that of P and Q. On the other hand, deep compositionality relates not only the syntax, but also the semantics: not only can we combine P and Q into $P\|Q$, but the semantics $[\![P\|Q]\!]$ of $P\|Q$ can be obtained by combining $[\![P]\!]$ and $[\![Q]\!]$. A simple model with deep compositionality is that of transition systems with trace semantics [Dil89, Lam93,Lyn96,AH99]. In the variable-based version of this model, a state is an assignment of values to a set of variables, a trace is a sequence of states, and the semantics $[\![P]\!]$ of a component P consists of the set of all traces that correspond to behaviors of P. If the variables written by P are read by another component Q, and vice versa, and components interact synchronously, then composition corresponds to the intersection of trace sets: $[\![P\|Q]\!] = [\![P]\!] \cap [\![Q]\!]$. If each component has also private variables, which are invisible to the other component, then

* This research was supported in part by the SRC contract 99-TJ-683.003, the AFOSR MURI grant F49620-00-1-0327, the MARCO GSRC grant 98-DT-660, the NSF Theory grant CCR-9988172, and the DARPA SEC grant F33615-C-98-3614.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 351–365, 2001.
© Springer-Verlag Berlin Heidelberg 2001

we obtain the observable traces of $P \| Q$ via projection from the behaviors of P and Q that agree on the mutually visible variables.

The chief advantage of deep over shallow compositionality is that deep compositionality enables not only the composition of systems, but also the composition of properties. In particular, it becomes possible to prove properties of systems by proving properties of their components. Since each component is simpler than the composite system, such a compositional approach can be markedly more efficient. A basic application of property composition consists in proving a refinement relation $P \| Q \preceq P' \| Q'$ between a composite implementation $P \| Q$ and a composite specification $P' \| Q'$ by proving independently the two component refinements $P \preceq P'$ and $Q \preceq Q'$. In practice, a more powerful *assume-guarantee* rule is preferred, where the proofs of each component refinement rely on the hypothesis that the other component refinement holds, yielding the proof obligations $P \| Q' \preceq P' \| Q'$ and $P' \| Q \preceq P' \| Q'$. Such a circular assume-guarantee rule is available, for example, for [MC81,AL95,McM97,AH99]. In spite of the advantages of deeply compositional models, no such model has thus far been presented for systems with both probability and nondeterminism. The difficulty, as we will detail in Section 2, lies in the interaction between the resolution of nondeterministic choice, mediated by *schedulers,* and composition.

We introduce a deeply compositional model for systems with both probabilistic and nondeterministic choice, and we show how the model leads to the first assume-guarantee rule for checking refinement between probabilistic systems. The model is based on a synchronous, variable-based view of systems, as in *reactive modules* [AH99]. The semantics of a component is obtained by generalizing trace semantics: instead of a trace, our basic semantical unit is a probability distribution on traces —i.e., a stochastic process over the state space— which we call a "bundle." A bundle represents a single (probabilistic) behavior of a component, once all nondeterminism has been resolved by a scheduler. Thus, the semantics $[\![P]\!]$ of a component P consists of a set of bundles. This is very similar to the semantics for the *probabilistic I/O automata* of [SL94,Seg95]. Unlike the models of [SL94,Seg95], however, our models are deeply compositional. In our models, the semantics of composition is essentially intersection, as in the nonprobabilistic case: for two components P and Q that share the same variables, we have $[\![P \| Q]\!] = [\![P]\!] \cap [\![Q]\!]$; this is now an intersection of bundles, rather than traces. This relationship will be generalized also to components with private variables.

Our deeply compositional semantics opens the way to the use of assume-guarantee methods for probabilistic systems. In the trace-based semantics of nondeterministic systems, refinement is defined as trace containment. In analogy, we define refinement as bundle containment: $P \preceq P'$ iff $[\![P]\!] \subseteq [\![P']\!]$. This definition, together with deep compositionality, ensures that refinement can be proven in a compositional fashion: $P \preceq P'$ and $Q \preceq Q'$ imply $P \| Q \preceq P' \| Q'$. Furthermore, we show that a circular assume-guarantee rule for refinement can be applied: $P \| Q' \preceq P' \| Q'$ and $P' \| Q \preceq P' \| Q'$ imply $P \| Q \preceq P' \| Q'$. This does not follow immediately from deep compositionality, but requires inductive reasoning, as in the nonprobabilistic case. Arguably, the ability of studying systems

in a compositional fashion is even more beneficial for probabilistic than for purely nondeterministic systems, due to the greater complexity of the verification algorithms and symbolic data structures [dAKN+00]. We therefore believe that our deeply compositional semantics, together with the assume-guarantee rule for proving refinement, represent a step forward in the analysis and verification of complex probabilistic systems.

2 Motivational Examples

In systems with both probabilistic and nondeterministic choice, the resolution of nondeterministic choice is mediated by *schedulers,* which specify how to choose between nondeterministic alternatives [Der70,Var85,SL94,BdA95]. Once a scheduler is fixed, the behavior of a system is a stochastic process, namely, a bundle. Following [SL94,Seg95], we define the semantics $[\![P]\!]$ of a component P as the set of bundles that arise from all possible schedulers for P. While *deterministic* schedulers resolve each choice in a deterministic manner [Var85], we opt for *randomized* schedulers, which select probability distributions of outcomes [SL94, BdA95], thus resolving nondeterminism in a randomized fashion, similarly to Markov decision processes [Der70]. Our preference for randomized schedulers is motivated by refinement: under randomized scheduling, if we replace probability by nondeterminism in a component P, obtaining the component P', then P refines P'. Hence, nondeterminism can be seen as "unspecified probability," and it can be used to encode imprecise probability values [JL91,dA98]. To see this, consider the following example.

Example 1 Assume that P and P' are two components, each writing once to a variable x that can take the two values 0 or 1. In P, the variable x is set to 0 or 1 with probability $\frac{1}{2}$ each; in P', the choice between setting x to 0 or 1 is entirely nondeterministic. Since there is no nondeterminism in P, there is a single scheduler for P (taking no choices), which gives rise to the behavior (bundle) with the two one-step traces $x:0$ and $x:1$, each with probability $\frac{1}{2}$. There are two deterministic schedulers for P': the first sets x to 0 and yields the bundle with the single trace $x:0$; the second sets x to 1 and yields the bundle with the single trace $x:1$. Therefore, with deterministic scheduling, $[\![P]\!] \cap [\![P']\!] = \emptyset$. There are infinitely many randomized schedulers for P', one for each real number $\delta \in [0,1]$: the scheduler σ_δ sets x to 0 with probability δ, and sets x to 1 with probability $1-\delta$, and thus yields the bundle with the two traces $x:0$ (probability δ) and $x:1$ (probability $1-\delta$). Choosing $\delta = \frac{1}{2}$, we see that using randomized schedulers, $[\![P]\!] \subseteq [\![P']\!]$, as desired. ∎

We adopt a purely variable-based view of systems: each state is a valuation of a set of typed variables. Following *reactive modules* [AH99], our components, called *probabilistic modules,* have a set of *private* variables, visible to the module alone, a set of *interface* variables, which are the outputs of the module, and a set of *external* variables, which are the inputs of the module. Together, the interface

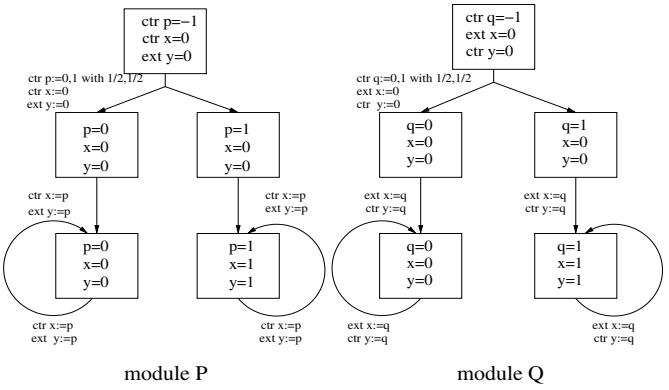


Fig. 1. Bundle of P and Q, but not of $P\|Q$

and external variables are called *observable*, and the private and interface variables are called the *controlled* variables of the module. Here, we justify some of the definitions that are necessary to achieve a deeply compositional semantics.

Example 2 The module P has private variable p, interface variable x, and external variable y. All variables are modified repeatedly, in a sequence of discrete steps. The initial values of p and x are -1 and 0, respectively. When $p = -1$, the module P updates p to 0 or 1 with equal probability $\frac{1}{2}$, and updates x to 0. When $p \neq -1$, the module P leaves p unchanged, and updates x to p. The initial value and updates of the external variable y are entirely nondeterministic. Let σ_P be the scheduler for P that initially sets y to 0, and then updates y to 0 when $p = -1$, and updates y to p when $p \neq -1$. The module Q, and its scheduler σ_Q, are defined symmetrically, with q in place of p, and x, y interchanged. The behavior of these two modules, under their respective schedulers, is illustrated in Figure 1. Under the scheduler σ_P, the observable part of the behavior of P is a bundle $\mathbf{b}$ consisting of the two infinite traces $(x:0, y:0), (0,0), (0,0), \ldots$ and $(x:0, y:0), (0,0), (1,1), \ldots$ with probability $\frac{1}{2}$ each. The bundle $\mathbf{b}$ also results from Q under the scheduler σ_Q. However, $\mathbf{b}$ is not a bundle of $P\|Q$, under any scheduler. In fact, there is no nondeterminism in $P\|Q$: the values for p and q are chosen independently, and the unique observable behavior of $P\|Q$ is the bundle that, for each $i, j \in \{0, 1\}$, contains the trace $(x:0, y:0), (0,0), (i,j), \ldots$ with probability $\frac{1}{4}$. ∎

Thus, in order to obtain a compositional semantics, the values of the external variables must be chosen without looking at the values of private variables. On the other hand, the choice of values for the controlled variables should be able to depend on the values of private variables. Hence, we need at least two schedulers for each module: one for the external variables, which cannot look at the values of the private variables, and one for the controlled variables, which can.

Example 3 Consider a module P with an interface variable x and an external variable y, and a module Q with the interface variable y and the external variable x. Both variables can have value 0 or 1, and are updated completely nondeterministically. The behavior of P is thus determined by two schedulers: a module scheduler π_P that provides a probability distribution for the choice between values 0 and 1 for x, and an environment scheduler η_P that provides a probability distribution for the choice between values 0 and 1 for y. Symmetrically, the behavior of Q is determined by the two schedulers π_Q and η_Q. In the compositon $P\|Q$, the variables x and y are both controlled variables. If we postulate that all controlled variables are controlled by the same scheduler, then there is a module scheduler $\pi_{P\|Q}$ for $P\|Q$ that chooses for (x, y) the values $(0, 0)$ with probability $\frac{1}{2}$, and $(1, 1)$ with probability $\frac{1}{2}$. This scheduler gives rise to the bundle that contains the one-step trace $(x:0, y:0)$ with probability $\frac{1}{2}$, and the one-step trace $(x:1, y:1)$ with probability $\frac{1}{2}$. This bundle is neither a bundle of P, nor a bundle of Q, however, because in P and Q the values for x and y are chosen independently. ∎

In previous models of probabilistic systems, a single scheduler is used to resolve all nondeterminism in the choice of values for the controlled variables [Var85, SL94,Seg95,BdA95]. The above example indicates that in order to achieve deep compositionality, we must abandon this restriction, and allow multiple schedulers for the resolution of the nondeterminism within a module. For each scheduler, we must specify the set of variables that it affects, as well as the set of variables at which it can look. To this end, we partition the controlled variables of a module into *atoms:* each atom represents a set of controlled variables that are scheduled together, by a single scheduler. Each atom has also a set of *read* variables, which are the variables visible to the scheduler, on which the choice of values for the controlled variables may depend. When we compose modules, we take the union of their sets of atoms, thus ensuring that the scheduling dependencies between variables remain unchanged. In the example above, P would contain an atom for scheduling x, and Q an atom for scheduling y (there are no read variables). The composite system $P\|Q$ then inherits the atoms of P and Q, and has two schedulers, one for x, the other for y. Each pair of schedulers for $P\|Q$ corresponds to both a scheduler of P and a scheduler of Q, yielding $[\![P]\!] \cap [\![Q]\!] = [\![P\|Q]\!]$.

Our atoms are derived directly from the atoms of *reactive modules* [AH99]. However, while in [AH99] the atoms indicate which variables are updated jointly (i.e., interdependently), and dependent on which other variables, here atoms acquire additional meaning: they indicate which variables are scheduled jointly, and dependent on which other variables. In particular, while in the nonprobabilistic case the merging of atoms never changes the behaviors (traces) of a module, in the probabilistic case, the merging of atoms may increase the behaviors (bundles) by permitting strictly more probabilistic dependencies between variable values, as the previous example illustrates. This is because it is the atom structure of a module that determines probabilistic dependence and, importantly, *independence* between variables values.

3 Probabilistic Modules and Composition

3.1 Definition of Probabilistic Modules

Definition 1. [**States and moves**] *Let X be a set of typed variables. An X-*state *s is a function that maps each variable in X to a value of the appropriate type. We write* $\mathrm{Val}(X)$ *for the set of X-states. An X-*move $\mathbf{m}$ *is a probability distribution on X-states. The move $\mathbf{m}$ is* nonprobabilistic *if the support of m is a single state. Given two X-moves $\mathbf{m}_1$ and $\mathbf{m}_2$, and a real number $\delta \in [0,1]$, we write $\delta \cdot \mathbf{m}_1 + (1-\delta) \cdot \mathbf{m}_2$ for the X-move $\mathbf{m}$ such that $\mathbf{m}(s) = \delta \cdot \mathbf{m}_1(s) + (1-\delta) \cdot \mathbf{m}_2(s)$ for all X-states s.*

While a nonprobabilistic transition (s, s') consists of a source state s and a target state s', a probabilistic transition $(s, \mathbf{m})$ consists of a source state s and a probability distribution $\mathbf{m}$ of target states. A nondeterministic collection of transitions (probabilistic or not) is called an "action." Consider, for example, the action $F = \{f_1, f_2\}$ with the two transitions $f_1 = (s, \mathbf{m}_1)$ and $f_2 = (s, \mathbf{m}_2)$. Every action is resolved by a scheduler, which, given a sequence of actions, produces a sequence of states. Given the action $F = \{f_1, f_2\}$ in state s, a deterministic scheduler may choose either the transition f_1, whose outcome is determined by the probability distribution $\mathbf{m}_1$, or the transition f_2, whose outcome is determined by the probability distribution $\mathbf{m}_2$. A randomized scheduler may choose any convex combination of f_1 and f_2, say, f_1 with probability δ and f_2 with probability $1-\delta$.

Definition 2. [**Transitions and actions**] *Let X and Y be two sets of typed variables. A* probabilistic transition $(s, \mathbf{m})$ from X to Y *consists of an X-state s and a Y-move $\mathbf{m}$. The transition $(s, \mathbf{m})$ is* nonprobabilistic *if the move $\mathbf{m}$ is nonprobabilistic. A* probabilistic action F from X to Y *is a set of probabilistic transitions from X to Y. The action F is* deterministic *if for every X-state s, there is at most one Y-move $\mathbf{m}$ such that $(s, \mathbf{m}) \in F$. The action F is* nonprobabilistic *if all transitions in F are nonprobabilistic. The action F is* convex-closed *if for all X-states s, all Y-moves m_1 and m_2, and all real numbers $\delta \in [0,1]$, if $(s, \mathbf{m}_1) \in F$ and $(s, \mathbf{m}_2) \in F$, then $(s, \delta \cdot \mathbf{m}_1 + (1-\delta) \cdot \mathbf{m}_2) \in F$. The* convex-closure $\mathrm{ConvexClosure}(F)$ *is the least superset of F that is a convex-closed action from X to Y.*

A system proceeds in a sequence of discrete rounds. In the first round, all system variables are initialized in accordance with initial actions; in the subsequent rounds, all system variables are updated in accordance with update actions. Dependencies between variables are expressed by clustering the variables into "atoms." If two variables are controlled (i.e., initialized and updated) by the same atom, then their behaviors are interdependent. Consequently, if the behaviors of two variables are desired to be independent, then the variables must be put into different atoms. Consider, for example, two boolean variables x and y. First, suppose that x and y are jointly controlled by a single atom. The deterministic initial action $\frac{1}{2}(x, y := 0, 0) + \frac{1}{2}(x, y := 1, 1)$ with probability $\frac{1}{2}$ initializes both variables to 0, and with probability $\frac{1}{2}$ initializes both variables to 1. There are

two possible initial states, (0,0) and (1,1). Second, suppose that x and y are independently controlled by different atoms. The deterministic initial actions $\frac{1}{2}(x := 0) + \frac{1}{2}(x := 1)$ and $\frac{1}{2}(x := 0) + \frac{1}{2}(x := 1)$ initialize each variable with equal probability to 0 or 1. There are four possible initial states, (0,0), (0,1), (1,0), and (1,1). If x is controlled by one atom, and y by another atom, then x may still depend on y, because the atom controlling x may "read" the value of y at the beginning of each round. All such read dependencies must be declared explicitly; the absence of read dependencies (or transitively implied read dependencies) between different atoms means true independence, in the probabilistic sense.

Definition 3. [**Atoms**] *Let X be a set of typed variables. A* probabilistic X-atom A *consists of a set* $\mathrm{readX}(A) \subseteq X$ *of* read variables*, a set* $\mathrm{ctrX}(A) \subseteq X$ *of* controlled variables*, a probabilistic* initial action $\mathrm{initF}(A)$ *from* $\emptyset$ *to* $\mathrm{ctrX}(A)$*, and a probabilistic* update action $\mathrm{updateF}(A)$ *from* $\mathrm{readX}(A)$ *to* $\mathrm{ctrX}(A)$*. The atom A is* deterministic *if both* $\mathrm{initF}(A)$ *and* $\mathrm{updateF}(A)$ *are deterministic actions. The atom A is* nonprobabilistic *if both* $\mathrm{initF}(A)$ *and* $\mathrm{updateF}(A)$ *are nonprobabilistic.*

In addition to its atoms, which provide the initial and update actions for variables, an open probabilistic system —or "module"— also provides the capability to view variables that are not initialized and updated by the module, and the capability to hide variables from the view of other modules. The former variables are called "external"; the latter, "private." The variables that are neither external nor private —i.e., the variables that are initialized and updated by the module and can be viewed by other modules— are called "interface" variables.

Definition 4. [**Modules**] *A* probabilistic module P *consists of a declaration and a body. The module declaration is a finite set of typed variables $X(P)$ that is partitioned into three sets: the set* $\mathrm{extlX}(P)$ *of* external variables*, the set* $\mathrm{intfX}(P)$ *of* interface variables*, and the set* $\mathrm{privX}(P)$ *of* private variables*. The module body is a finite set* $\mathrm{Atoms}(P)$ *of probabilistic $X(P)$-atoms such that (1)* $\mathrm{intfX}(P) \cup \mathrm{privX}(P) = \bigcup_{A \in \mathrm{Atoms}(P)} \mathrm{ctrX}(A)$*, and (2) for all atoms A_1 and A_2 in* $\mathrm{Atoms}(P)$*, the sets* $\mathrm{ctrX}(A_1)$ *and* $\mathrm{ctrX}(A_2)$ *are disjoint. The module P is* deterministic *if all atoms in* $\mathrm{Atoms}(P)$ *are deterministic. The module P is* nonprobabilistic *if all atoms in* $\mathrm{Atoms}(P)$ *are nonprobabilistic.*

Given a module P, we call $\mathrm{intfX}(P) \cup \mathrm{extlX}(P)$ the set of *observable* variables of P, and we call $\mathrm{privX}(P) \cup \mathrm{intfX}(P)$ the set of *controlled* variables of P. The nonprobabilistic modules are exactly the *reactive modules* of [AH99] without await dependencies. We have omitted await dependencies, which are instrumental for synchronous communication, for simplicity; they can be added without changing the results of this paper. Modules without external variables are called "closed." Every closed module defines a Markov decision process; every closed deterministic module defines a Markov chain.

3.2 Operations on Probabilistic Modules

We define three operations on probabilistic modules: hiding, composition, and opening. The hiding (or abstraction) operation makes some interface variables private.

Definition 5. [**Hiding**] *Let P be a probabilistic module, and let $Y \subseteq \mathrm{intfX}(P)$ be a set of interface variables. By* hiding Y in P *we obtain the probabilistic module $P\backslash Y$ with the set* $\mathrm{extlX}(P\backslash Y) = \mathrm{extlX}(P)$ *of external variables, the set* $\mathrm{intfX}(P\backslash Y) = \mathrm{intfX}(P)\backslash Y$ *of interface variables, the set* $\mathrm{privX}(P\backslash Y) = \mathrm{privX}(P) \cup Y$ *of private variables, and the set* $\mathrm{Atoms}(P\backslash Y) = \mathrm{Atoms}(P)$ *of atoms.*

The (parallel) composition operation puts together two modules which control the behaviors of disjoint sets of variables. The composition operation can be applied only when the observable —i.e., external and interface— variables of two modules coincide. This constraint is necessary for a compositional semantics in the presence of probabilities. If a module has a private variable p and an external variables y, then the module semantics insists on the independence between p and y, because the environment, which controls y, cannot observe p. It is therefore illegal to compose a module with private p, but without external y, and an environment that controls y, because the module, which does not know about the existence of y, has no way of noting that p and y must be independent. We will illustrate this in Example 4, presented below after the necessary terminology has been introduced. This underlines how the scoping of variables in the probabilistic case is considerably more delicate than in the nonprobabilistic case, where incidental dependencies between variables cause no harm.

Definition 6. [**Composition**] *Two probabilistic modules P_1 and P_2* can be composed *if (1)* $\mathrm{extlX}(P_1) \cup \mathrm{intfX}(P_1) = \mathrm{extlX}(P_2) \cup \mathrm{intfX}(P_2)$*, (2)* $\mathrm{intfX}(P_1) \cap \mathrm{intfX}(P_2) = \emptyset$*, and (3)* $\mathrm{privX}(P_1) \cap X(P_2) = \emptyset$ *and* $X(P_1) \cap \mathrm{privX}(P_2) = \emptyset$*. Two* composition *of two probabilistic modules P_1 and P_2 that can be composed is the probabilistic module $P_1||P_2$ with the set* $\mathrm{extlX}(P_1||P_2) = (\mathrm{extlX}(P_1) \cup \mathrm{extlX}(P_2))\backslash\mathrm{intfX}(P_1||P_2)$ *of external variables, the set* $\mathrm{intfX}(P_1||P_2) = \mathrm{intfX}(P_1) \cup \mathrm{intfX}(P_2)$ *of interface variables, the set* $\mathrm{privX}(P_1||P_2) = \mathrm{privX}(P_1) \cup \mathrm{privX}(P_2)$ *of private variables, and the set* $\mathrm{Atoms}(P_1||P_2) = \mathrm{Atoms}(P_1) \cup \mathrm{Atoms}(P_2)$ *of atoms.*

The opening operation adds external variables to a module, and is unique to probabilistic modules. It is used to ensure that two modules have the same set of observable variables before they are composed.

Definition 7. [**Opening**] *Let P be a probabilistic module, and let Y be a set of typed variables disjoint from the set $X(P)$ of module variables. By* opening P to Y *we obtain the probabilistic module $P \uplus Y$ with the set* $\mathrm{extlX}(P \uplus Y) = \mathrm{extlX}(P) \cup Y$ *of external variables, the set* $\mathrm{intfX}(P \uplus Y) = \mathrm{intfX}(P)$ *of interface variables, the set* $\mathrm{privX}(P \uplus Y) = \mathrm{privX}(P)$ *of private variables, and the set* $\mathrm{Atoms}(P \uplus Y) = \mathrm{Atoms}(P)$ *of atoms.*

3.3 Trace Semantics of Probabilistic Systems

Definition of probabilistic languages. While the behavior of a deterministic and closed nonprobabilistic system is an infinite state sequence —called a "trace"— the behavior of a deterministic and closed probabilistic system (Markov chain) is a probability distribution on traces —called a "bundle." Consequently, the possible behaviors of a nondeterministic or open probabilistic system form a *set* of traces, and the possible behaviors of a nondeterministic or open probabilistic system (in the nondeterministic and closed case, a Markov decision process) form a *set* of bundles. We restrict ourselves to safe systems, where it suffices to consider *finite* behaviors, albeit of arbitrary length; this restriction is particularly technically convenient in the probabilistic case, as probability distributions on finite traces can be defined in a straightforward manner.

Definition 8. [**Traces and bundles**] *Let X be a set of typed variables, and let n be a nonnegative integer. An* X-trace t of length n *is a sequence of X-states with n elements. We write ε for the empty sequence, and given $1 \leq i \leq n$, we write $t(i)$ for the i-th element of t. We write* $\mathrm{Val}^n(X)$ *for the set of X-traces of length n. An* X-bundle of length n *is a probability distribution over X-traces of length n. The unique X-bundle of length 0, which assigns the probability 1 to ε, is called the* empty bundle. *The bundle* $\mathbf{b}$ *is* nonprobabilistic *if the support of* $\mathbf{b}$ *is a single trace. If $n > 0$, then the* prefix *of* $\mathbf{b}$ *is the X-bundle* $\mathbf{b}'$ *of length $n-1$ such that $\mathbf{b}'(t) = \sum_{s \in \mathrm{Val}(X)} \mathbf{b}(t \cdot s)$ for all X-traces t of length $n-1$. The X-bundle* $\mathbf{b}''$ *of length $n+1$ is an* extension *of* $\mathbf{b}$ *if* $\mathbf{b}$ *is the prefix of* $\mathbf{b}''$.

Each bundle records the outcome of a particular sequence of nondeterministic or randomized choices made by a system. Such a sequence of choices is called a "scheduler" (to be defined later). The set of bundles that result from all possible schedulers are collected forms a probabilistic languages.

Definition 9. [**Languages**] *Let X be a set of typed variables. A set L of X-bundles is* prefix-closed *if for every bundle* $\mathbf{b}$ *in L, the prefix of* $\mathbf{b}$ *is also in L. The set L of bundles is* extension-closed *if (1) the empty bundle is in L, and (2) for every bundle* $\mathbf{b}$ *in L, some extension of* $\mathbf{b}$ *is also in L. A* probabilistic X-language *L is a prefix-closed and extension-closed set of X-bundles. The language L is* deterministic *if for all nonnegative integers n, there is a single bundle of length n in L. The language L is* nonprobabilistic *if all bundles in L are nonprobabilistic.*

The nonprobabilistic languages are precisely the prefix-closed and extension-closed trace sets; that is, the languages generated by safe and deadlock-free discrete systems.

Operations on probabilistic languages. We define two operations on bundles and on probabilistic languages: projection and product. Properties of these operations are needed to prove the compositionality of hiding and composition for probabilistic systems.

Definition 10. [**Projection**] *Let X and $X' \subseteq X$ be two sets of typed variables. The* X'-projection *of an X-state s is the X'-state $s[X']$ such that $(s[X'])(x) =$*

$s(x)$ for all variables $x \in X'$. The X'-projection *of an X-move* $\mathbf{m}$ *is the X'-move* $\mathbf{m}[X']$ *such that* $(\mathbf{m}[X'])(s') = \sum_{s \in \mathrm{Val}(X) \text{ with } s[X']=s'} \mathbf{m}(s)$ *for all X'-states s'. The* X'-projection *of an X-trace t of length n is the X'-trace $t[X']$ of length n such that $(t[X'])(i) = (t(i))[X']$ for all $1 \leq i \leq n$. The* X'-projection *of an X-bundle* $\mathbf{b}$ *of length n is the X'-bundle* $\mathbf{b}[X']$ *of length n such that* $(\mathbf{b}[X'])(t') = \sum_{t \in \mathrm{Val}^n(X) \text{ with } t[X']=t'} \mathbf{b}(t)$ *for all X'-traces t of length n. The* X'-projection *of an X-language L is the X'-language $L[X'] = \{\mathbf{b}[X'] \mid \mathbf{b} \in L\}$.*

Definition 11. [**Product**] *Let X_1 and X_2 be two sets of typed variables. An X_1-state (resp. move; trace; bundle) s_1 and a X_2-state (resp. move; trace; bundle) s_2* can be multiplied *if $s_1[X_1 \cap X_2] = s_2[X_1 \cap X_2]$. The* product *of an X_1-state s_1 and an X_2-state s_2 that can be multiplied is the $(X_1 \cup X_2)$-state $s_1 \times s_2$ such that $(s_1 \times s_2)(x_1) = s_1(x_1)$ for all variables $x_1 \in X_1$, and $(s_1 \times s_2)(x_2) = s_2(x_2)$ for all $x_2 \in X_2$. The* product *of an X_1-move $\mathbf{m}_1$ and an X_2-move $\mathbf{m}_2$ that can be multiplied is the $(X_1 \cup X_2)$-move $\mathbf{m}_1 \times \mathbf{m}_2$ such that $(\mathbf{m}_1 \times \mathbf{m}_2)(s) = \mathbf{m}_1(s[X_1]) \cdot \mathbf{m}_2(s[X_2]) / \mathbf{m}_1(s[X_1 \cap X_2])$ for all $(X_1 \cup X_2)$-states s. The* product *of an X_1-trace t_1 and an X_2-trace t_2 that have length n and can be multiplied is the $(X_1 \cup X_2)$-trace $t_1 \times t_2$ of length n such that $(t_1 \times t_2)(i) = t_1(i) \times t_2(i)$ for all $1 \leq i \leq n$. The* product *of an X_1-bundle $\mathbf{b}_1$ and an X_2-bundle $\mathbf{b}_2$ that have length n and can be multiplied is the $(X_1 \cup X_2)$-bundle $\mathbf{b}_1 \times \mathbf{b}_2$ of length n such that $(\mathbf{b}_1 \times \mathbf{b}_2)(t) = \mathbf{b}_1(t[X_1]) \cdot \mathbf{b}_2(t[X_2]) / \mathbf{b}_1(t[X_1 \cap X_2])$ for all $(X_1 \cup X_2)$-traces t of length n. The* product *of an X_1-language L_1 and an X_2-language L_2 is the $(X_1 \cup X_2)$-language $L_1 \times L_2 = \{\mathbf{b}_1 \times \mathbf{b}_2 \mid \mathbf{b}_1 \in L_1$* and *$\mathbf{b}_2 \in L_2$* can be multiplied*$\}$.*

The product of bundle languages is the probabilistic analogue to the intersection of trace languages. This is captured in the following lemma.

Lemma 1. *Let L_1 be a probabilistic X_1-language, and let L_2 be a probabilistic X_2-language. Then $(L_1 \times L_2)[X_1 \cap X_2] = L_1[X_1 \cap X_2] \cap L_2[X_1 \cap X_2]$.*

Containment between probabilistic languages. Since the set of possible behaviors of a probabilistic system is a set of bundles, the appropriate notion of refinement between probabilistic systems is bundle containment: an implementation refines a specification iff every possible behavior (bundle) of the implementation is a legal behavior (bundle) of the specification.

Definition 12. [**Bundle containment**] *Let X and $X' \subseteq X$ be two sets of typed variables. If L is a probabilistic X-language, and L' is a probabilistic X'-language, then L* is bundle-contained in *L' if $L[X'] \subseteq L'$.*

3.4 Connecting Syntax and Semantics

Bundle semantics of probabilistic modules. We associate with every probabilistic module a probabilistic language, i.e., a set of bundles. The key concept for doing this is the concept of a "scheduler," which represents a possible sequence of choices taken by the module. Each scheduler, then, gives rise to an infinite

bundle that can be represented by all its finite prefixes. We permit randomized schedulers, which in each state can choose probability distributions over the enabled transitions. By contrast, a deterministic scheduler must choose exactly one of the enabled transitions.

Definition 13. [**Schedulers**] *Let X and Y be two sets of variables. A* scheduler *σ from X to Y is a function that maps every X-trace to a probability distribution on Y-states. The scheduler σ is* nonprobabilistic *if for all X-traces t, the support of $\sigma(t)$ is a single Y-state. If σ is a scheduler from X to X, then the* 0-outcome *of σ is the empty bundle, and for all positive integers $i > 0$, the* i-outcome *of σ is an inductively defined X-bundle $\mathbf{b}_i$ of length i: the bundle $\mathbf{b}_i$ is the extension of the bundle $\mathbf{b}_{i-1}$ such that $\mathbf{b}_i(t) = \mathbf{b}_{i-1}(t(1) \cdots t(i-1)) \cdot (\sigma(t(1) \cdots t(i-1)))(t(i))$ for all X-traces t of length i. We collect the set of i-outcomes of σ, for all $i \geq 0$, in the set* $\mathrm{Outcome}(\sigma)$ *of X-bundles.*

Each scheduler for a module consists of a scheduler for the environment, which chooses the initial and updated values for the external variables, together with a scheduler for each atom, which chooses the initial and updated values for the variables controlled by that atom.

Definition 14. [**Schedulers for an atom**] *Consider a probabilistic X-atom A. The set* $\mathrm{atom}\Sigma(A)$ *of* atom schedulers for *A contains all schedulers σ from* $\mathrm{readX}(A)$ *to* $\mathrm{ctrX}(A)$ *such that (1)* $(\cdot, \sigma(\varepsilon)) \in \mathrm{ConvexClosure}(\mathrm{initF}(A))$*, and (2)* $(t(n), \sigma(t)) \in \mathrm{ConvexClosure}(\mathrm{updateF}(A))$ *for all nonempty* $\mathrm{readX}(A)$*-traces t of length n. A scheduler σ in* $\mathrm{atom}\Sigma(A)$ *is* deterministic *if (1)* $(\cdot, \sigma(\varepsilon)) \in \mathrm{initF}(A)$*, and (2)* $(t(n), \sigma(t)) \in \mathrm{updateF}(A)$ *for all nonempty traces t of length n. Let* $\mathrm{atom}\Sigma^d(A)$ *be the set of deterministic schedulers in* $\mathrm{atom}\Sigma(A)$*.*

To compose the schedulers for several atoms, we define the product of schedulers.

Definition 15. [**Product of schedulers**] *Two schedulers σ_1 and σ_2 are* disjoint *if σ_1 is a scheduler from X_1 to Y_1, and σ_2 is a scheduler from X_2 to Y_2, and $Y_1 \cap Y_2 = \emptyset$. If σ_1 is a scheduler from X_1 to Y_1, and σ_2 is a scheduler from X_2 to Y_2, such that σ_1 and σ_2 are disjoint, then the* product *is the scheduler $\sigma_1 \times \sigma_2$ from $X_1 \cup X_2$ to $Y_1 \cup Y_2$ such that $(\sigma_1 \times \sigma_2)(t) = \sigma_1(t[X_1]) \times \sigma_2(t[X_2])$ for all $(X_1 \cup X_2)$-traces t. If σ_1 and σ_2 are two sets of schedulers such that every scheduler in Σ_1 is disjoint from every scheduler in Σ_2, then $\Sigma_1 \times \Sigma_2 = \{\sigma_1 \times \sigma_2 \mid \sigma_1 \in \Sigma_1 \text{ and } \sigma_2 \in \Sigma_2\}$.*

The environment scheduler can initialize and update the external variables in arbitrary, interdependent ways.

Definition 16. [**Schedulers for a module**] *Consider a probabilistic module P. The set* $\mathrm{extl}\Sigma(P)$ *of* environment schedulers for *P contains all schedulers from* $\mathrm{extlX}(P) \cup \mathrm{intfX}(P)$ *to* $\mathrm{extlX}(P)$*. Let* $\mathrm{extl}\Sigma^d(P)$ *be the set of nonprobabilistic schedulers in* $\mathrm{extl}\Sigma(P)$*. The set* $\mathrm{mod}\Sigma(P)$ *of* module schedulers for *P contains the schedulers from $X(P)$ to $X(P)$ such that* $\mathrm{mod}\Sigma(P) = \mathrm{extl}\Sigma(P) \times \prod_{A \in \mathrm{Atoms}(P)} \mathrm{atom}\Sigma(A)$*. Let* $\mathrm{mod}\Sigma^d(P) = \mathrm{extl}\Sigma^d(P) \times \prod_{A \in \mathrm{Atoms}(P)} \mathrm{atom}\Sigma^d(A)$*.*

We are finally ready to define the "trace semantics" of a probabilistic module.

Definition 17. [Semantics of a module] *Given a probabilistic module P, we define* $L(P) = \{\text{Outcome}(\sigma) \mid \sigma \in \text{mod}\Sigma(P)\}$ *and* $L^d(P) = \{\text{Outcome}(\sigma) \mid \sigma \in \text{mod}\Sigma^d(P)\}$. *The* trace semantics of the probabilistic module P *is* $[\![P]\!] = L(P)[\text{extlX}(P) \cup \text{intfX}(P)]$. *The* deterministic trace semantics of P *is* $[\![P]\!]^d = L^d(P)[\text{extlX}(P) \cup \text{intfX}(P)]$.

It it not difficult to verify that the bundle semantics of a module is indeed prefix-closed and extension-closed: if P is a probabilistic module, then $[\![P]\!]$ is a probabilistic $(\text{extlX}(P) \cup \text{intfX}(P))$-language. In general, $[\![P]\!]^d \subset [\![P]\!]$. For nonprobabilistic modules, the traditional trace semantics corresponds to bundle semantics with only deterministic schedulers: according to [AH99], the *trace semantics* of a reactive module P is $[\![P]\!]^d$.

Bundle interpretation of module operations. The hiding of variables in a module corresponds to a projection on the bundle language of the module: it is easy to check that for every probabilistic module P, and every set $Y \subseteq \text{intfX}(P)$ of interface variables, $[\![P \backslash Y]\!] = [\![P]\!][\text{extlX}(P) \cup \text{intfX}(P) \backslash Y]$. The composition of two modules corresponds to a product on the respective bundle languages. This observation, which is stated in the following proposition, will be instrumental to the compositionality properties of probabilistic modules given in the next section.

Theorem 1. *If P_1 and P_2 are two probabilistic modules that can be composed, then* $[\![P_1 || P_2]\!] = [\![P_1]\!] \times [\![P_2]\!] = [\![P_1]\!] \cap [\![P_2]\!]$.

Proof. By induction on the length of bundles, we show the following two observations, which rely heavily on the fact that schedulers have restricted visibility. First, $\mathbf{b} \in L(P_1 \| P_2)$ implies that $\mathbf{b}[X(P_1)] \in L(P_1)$ and $\mathbf{b}[X(P_2)] \in L(P_2)$. Moreover, $\mathbf{b} = \mathbf{b}[X(P_1)] \times \mathbf{b}[X(P_2)]$. This in turn also means that every bundle in $[\![P_1 \| P_2]\!]$ can be "factored," via projection, into bundles in $[\![P_1]\!]$ and $[\![P_2]\!]$. Second, for all bundles $\mathbf{b}_1 \in L(P_1)$ and $\mathbf{b}_2 \in L(P_2)$ such that $\mathbf{b}_1[X(P_1) \cap X(P_2)] = \mathbf{b}_2[X(P_1) \cap X(P_2)]$, we have $\mathbf{b}_1 \times \mathbf{b}_2 \in L(P_1 \| P_2)$. These two observations combine to give the first equality. The observation that $X(P_1) \cap X(P_2)$ is the set of observables of both P_1 and P_2, coupled with Lemma 1, gives the second equality. ∎

The following example illustrates the need for restricting composition to modules with identical sets of observable variables.

Example 4 Consider two modules P and Q defined as in Example 2, except that the variables p and q are both interface variables, and thus observable. We assume that each controlled variable of P and Q belongs to a different atom. Under the scheduler σ_P, the behavior of P is a bundle $\mathbf{b}_P$ consisting of the two infinite traces $(p:-1, x:0, y:0), (0,0,0), (0,0,0), \ldots$ and $(p:-1, x:0, y:0), (1,0,0), (1,1,1), \ldots$ with probability $\frac{1}{2}$ each (to be precise, there are infinitely many bundles, each consisting of two finite traces, whose limit is $\mathbf{b}_P$, but in examples, we find it convenient to informally consider bundles of infinite traces). Similarly, the behavior of Q under σ_Q is the bundle $\mathbf{b}_Q$ that contains the two traces

$(x:0, y:0, q:-1), (0,0,0), (0,0,0), \ldots$ and $(x:0, y:0, q:-1), (0,0,1), (1,1,1), \ldots$, each with probability $\frac{1}{2}$ (see Figure 1). The bundles $\mathbf{b}_P$ and $\mathbf{b}_Q$ can be multiplied, but their product $\mathbf{b}_P \times \mathbf{b}_Q$ is not a bundle of $P \| Q$. In fact, $\mathbf{b}_P \times \mathbf{b}_Q$ consists of the two traces $(p:-1, x:0, y:0, q:-1), (0,0,0,0), (0,0,0,0), \ldots$ and $(p:-1, x:0, y:0, q:-1), (1,0,0,1), (1,1,1,1), \ldots$ On the other hand, since the values of p and q are chosen independently, $[\![P \| Q]\!]$ consists of a single bundle $\mathbf{b}_{P\|Q}$, containing for each $i, j \in \{0,1\}$ the trace $(p:-1, x:0, y:0, q:-1), (i,0,0,j), (i,i,j,j), \ldots$ with probability $\frac{1}{4}$. It follows that $[\![P]\!] \times [\![Q]\!] \supset [\![P \| Q]\!]$. ∎

4 Refinement between Probabilistic Modules

4.1 Definition of Probabilistic Refinement

The refinement relation between probabilistic modules is defined essentially as bundle containment. Unlike in the nonprobabilistic case, however, we require an additional constraint on the atom structure of the two modules, which ensures that an implementation cannot exhibit more variable dependencies than a specification. In other words, all variables that are specified to be independent must be implemented independently.

Definition 18. [Refinement between modules] *Let P and P' be two probabilistic modules. The module P* structurally refines *P', written $P \preceq_S P'$, if (1)* $\mathrm{intfX}(P) \supseteq \mathrm{intfX}(P')$ *and* $\mathrm{extlX}(P) \cup \mathrm{intfX}(P) \supseteq \mathrm{extlX}(P')$*, (2) for all variables $x_1, x_2 \in \mathrm{intfX}(P')$, if there is an atom $A \in \mathrm{Atoms}(P)$ such that $x_1, x_2 \in \mathrm{ctrX}(A)$, then there is an atom $A' \in \mathrm{Atoms}(P')$ such that $x_1, x_2 \in \mathrm{ctrX}(A')$, and (3) for all variables $x \in \mathrm{intfX}(P')$ and $y \in \mathrm{intfX}(P') \cup \mathrm{extlX}(P')$, if there is an atom $A \in \mathrm{Atoms}(P)$ such that $x \in \mathrm{ctrX}(A)$ and $y \in \mathrm{readX}(A)$, then there is an atom $A' \in \mathrm{Atoms}(P')$ such that $x \in \mathrm{ctrX}(A')$ and $y \in \mathrm{readX}(A')$. The module P (*behaviorally*) refines P', written $P \preceq P'$, if $P \preceq_S P'$ and (4) $[\![P]\!]$ is bundle-contained in $[\![P']\!]$.*

It is easy to check that the refinement relation $\preceq$ it is a preorder. Furthermore, every probabilistic module refines its nonprobabilistic abstraction. The nonprobabilistic abstraction of a probabilistic action F is the nonprobabilistic action $\{(s, s') \mid (s, \mathbf{m}) \in F \text{ and } s' \in \mathrm{Support}(\mathbf{m})\}$. The *nonprobabilistic abstraction* $\mathrm{Nonprob}(P)$ of a probabilistic module P is the nonprobabilistic module that results from P by replacing all initial and update actions of P with their nonprobabilistic abstractions. Then, $P \preceq \mathrm{Nonprob}(P)$.

Refinement between nonprobabilistic modules, however, does not quite agree with refinement between *reactive modules*, as defined in [AH99]. The reason is that conditions (2) and (3) are absent from the definition of refinement for reactive modules, which is purely behavioral (namely, trace containment). For example, two atoms of a reactive module specification can be implemented by a single atom. If atoms are viewed structurally, say, as blocks in a block diagram, then such a refinement breaks component boundaries. This is brought to the fore formally in the probabilistic case, where atoms carry meaning as boundaries between independent variables. We submit that it is the definition above,

including the structural conditions (2) and (3), which is more sensible also in the nonprobabilistic case of reactive modules. Once conditions (2) and (3) are added to the refinement between reactive modules, then the probabilistic case is a conservative extension: for nonprobabilistic modules P and P', we have $P \preceq P'$ iff $P \preceq_S P'$ and (4^d) $[\![P]\!]^d$ is bundle-contained in $[\![P']\!]^d$.

4.2 Compositionality of Probabilistic Refinement

The following theorem summarizes the compositionality properties of the refinement relation between probabilistic modules. In particular, refinement is a congruence with respect to all module operations, and the refinement between composite modules can be decomposed using circular assume-guarantee reasoning.

Theorem 2. **[Compositionality]** *The following statements are true, provided all subexpressions are well-defined:*

- $P \preceq P\backslash Y$.
- $P \uplus Y \preceq P$.
- $P||Q \preceq P$.
- *If* $P \preceq P'$, *then* $P\backslash Y \preceq P'\backslash Y$.
- *If* $P \preceq P'$, *then* $P \uplus Y \preceq P' \uplus Y$.
- *If* $P \preceq P'$, *then* $P||Q \preceq P'||Q$.
- *If* $P||Q' \preceq Q$ *and* $Q||P' \preceq Q'$, *then* $P||P' \preceq Q||Q'$.

The last assertion is an assume-guarantee rule for probabilistic modules. Its proof uses the following lemma, whose proof relies on Theorem 1. Essentially, the lemma states that the observable part of a bundle of length i is obtained from the observable part of its prefix of length $i-1$, the environment scheduler and the "observable" behaviour of the module scheduler, and the last may be written as the product of the observable behaviours of the atom schedulers and the environment scheduler. Given a scheduler σ from $X(P)$ to $\mathrm{ctrX}(P)$, an $X(P)$-bundle $\mathbf{b}$, and its projection $\mathbf{b}^* = \mathbf{b}[\mathrm{intfX}(P) \cup \mathrm{extlX}(P)]$, define the *observable scheduler* σ^* w.r.t. $\mathbf{b}$ as the scheduler from $\mathrm{intfX}(P) \cup \mathrm{extlX}(P)$ to $\mathrm{intfX}(P)$ such that for every $(\mathrm{intfX}(P) \cup \mathrm{extlX}(P))$-trace s of length $i-1$,

$$\sigma^*(s(1)\cdots s(i-1)) = \sum_{t^*=s} \frac{\mathbf{b}_{i-1}(t(1)\cdots t(i-1))}{\mathbf{b}^*_{i-1}(s(1)\cdots s(i-1))} \cdot \sigma(t(1)\cdots t(i-1))[\mathrm{intfX}(P)],$$

where $t^* = t[\mathrm{intfX}(P) \cup \mathrm{extlX}(P)]$. Recall that if $P = P_1 \| P_2$ is defined, then it must be that $\mathrm{intfX}(P) \cup \mathrm{extlX}(P) = \mathrm{intfX}(P_1) \cup \mathrm{extlX}(P_1) = \mathrm{intfX}(P_2) \cup \mathrm{extlX}(P_2)$.

Lemma 2. *Let* $P = P_1 \| P_2$ *be a probabilistic module, and let* $\mathbf{b} \in L(P_1 \| P_2)$ *be the outcome of a scheduler* $\sigma = \sigma_{Env} \times \sigma_{P_1} \times \sigma_{P_2}$ *with* $\sigma_{Env} \in \mathrm{extl}\Sigma(P)$ *and* $\sigma_{P_j} \in \prod_{A \in \mathrm{Atoms}(P_j)} \mathrm{atom}\Sigma(A)$ *for* $j = 1, 2$. *For every* $(\mathrm{intfX}(P) \cup \mathrm{extlX}(P))$-*trace* t *of length* i, *we have* $\mathbf{b}^*_i(t) = \mathbf{b}^*_{i-1}(t(1)\cdots t(i-1)) \cdot (\sigma_{Env} \times \sigma^*_{P_1} \times \sigma^*_{P_2})(t(i))$, *where* $\sigma^*_{P_j}$ *is the observable scheduler w.r.t.* $\mathbf{b}[X(P_j)]$ *for* $j = 1, 2$.

Using this lemma, the soundness of the assume-guarantee rule can be proved in a fashion similar to that for nonprobabilistic systems like *reactive modules* [AH99].

References

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design* 15:7–48, 1999.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Programming Languages and Systems*, 17:507–534, 1995.

[BdA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lect. Notes in Comp. Sci.*, pages 499–513. Springer-Verlag, 1995.

[dA98] L. de Alfaro. Stochastic transition systems. In *Concurrency Theory*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 423–438. Springer-Verlag, 1998.

[dAKN+00] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 395–410. Springer-Verlag, 2000.

[Der70] C. Derman. *Finite State Markovian Decision Processes.* Academic Press, 1970.

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* The MIT Press, 1989.

[JL91] B. Jonsson and K.G. Larsen. Specification and refinement of probabilistic processes. In *Proc. Symp. Logic in Computer Science*, pages 266–277. IEEE Computer Society Press, 1991.

[Lam93] L. Lamport. Specifying concurrent program modules. *ACM Trans. Progamming Languages and Systems*, 5:190–222, 1993.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Engineering*, SE-7:417–426, 1981.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer-Aided Verification*, volume 1254 of *Lect. Notes in Comp. Sci.*, pages 24–35. Springer-Verlag, 1997.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, 1995. Technical Report MIT/LCS/TR-676.

[SL94] R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. In *Concurrency Theory*, volume 836 of *Lect. Notes in Comp. Sci.*, pages 481–496. Springer-Verlag, 1994.

[Var85] M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state systems. In *Proc. Symp. Foundations of Computer Science*, pages 327–338. IEEE Computer Society Press, 1985.

Towards an Efficient Algorithm for Unfolding Petri Nets

Victor Khomenko and Maciej Koutny

Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
{Victor.Khomenko, Maciej.Koutny}@ncl.ac.uk

Abstract. Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the ways to exploit such a semantics is to consider (finite prefixes of) net unfoldings, which contain enough information to reason about the reachable markings of the original Petri nets. In this paper, we propose several improvements to the existing algorithms for generating finite complete prefixes of net unfoldings. Experimental results demonstrate that one can achieve significant speedups when transition presets of a net being unfolded have overlapping parts.

Keywords: Model checking, Petri nets, unfolding, concurrency.

1 Introduction

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use formal methods and, especially, computer aided verification tools implementing model checking [2] — a technique in which the verification of a system is carried out using a finite representation of its state space. The main drawback of model checking is that it suffers from the state space explosion problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To help in coping with this, a number of techniques have been proposed, which can roughly be classified as aiming at an implicit compact representation of the full state space of a reactive concurrent system, or at an explicit generation of its reduced (though sufficient for a given verification task) representation. Techniques aimed at reduced representation of state spaces are typically based on the independence (commutativity) of some actions, often relying on the partial order view of concurrent computation. Such a view is the basis for algorithms employing McMillan's (finite prefixes of) Petri net unfoldings ([7,14]), where the entire state space of a system is represented implicitly, using an acyclic net to represent relevant system's actions and local states.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 366–380, 2001.
© Springer-Verlag Berlin Heidelberg 2001

In view of the development of fast model checking algorithms employing unfoldings ([10,11,12]), the problem of efficiently building them is becoming increasingly important. Recently, [6,7,8] addressed this issue — considerably improving the original McMillan's technique — but we feel that the problem of generating net unfoldings deserves further investigation. Though there are negative theoretical results concerning this problem ([5,10]), in practice unfoldings can often be built quite efficiently. [7] stated that the slowest part of their unfolding algorithm was building possible extensions of the branching process being constructed (the decision version of this problem is NP-complete, see [10]). To compute them, [6] suggests to keep the concurrency relation and provides a method of maintaining it. This approach is fast for simple systems, but soon deteriorates as the amount of memory needed to store the concurrency relation may be quadratic in the number of conditions in the already built part of the unfolding.

In this paper, we propose another method of computing possible extensions and, although it is compatible with the concurrency relation approach, we decided to abandon this data structure in order to be able to construct larger prefixes. We show how to find new transition instances to be inserted in the unfolding, not by trying the transitions one-by-one, but several at once, merging the common parts of the work. Moreover, we provide some additional heuristics. Experimental results demonstrate that one can achieve significant speedups if the transitions of a safe Petri net being unfolded have overlapping parts. All missing proofs can be found in [13].

2 Basic Notions

A *net* is a triple $N \stackrel{\text{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e. $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and markings are shown by placing tokens within circles. As usual, we will denote ${}^\bullet z \stackrel{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z, y) \in F\}$, for all $z \in P \cup T$, and ${}^\bullet Z \stackrel{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (P, T, F)$ and an *initial* marking M_0. A transition $t \in T$ is *enabled* at a marking M if for every $p \in {}^\bullet t$, $M(p) \geq 1$. Such a transition can be *executed*, leading to a marking $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$. The set of *reachable* markings of Σ is the smallest (w.r.t. set inclusion) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[t\rangle M'$ (for some $t \in T$) then $M' \in [M_0\rangle$. Σ is *safe* if for every reachable marking M, $M(P) \subseteq \{0, 1\}$; and *bounded* if there is $k \in \mathbb{N}$ such that $M(P) \subseteq \{0, \ldots, k\}$, for every reachable marking M. Unless stated otherwise, we will assume that a net system Σ to be unfolded is safe, and use PreMax to denote the maximal size of transition preset in Σ.

Branching processes. Two nodes (places or transitions), y and y', of a net $N = (P, T, F)$ are *in conflict*, denoted by $y \# y'$, if there are distinct transitions

$t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *self-conflict* if $y \# y$.

An *occurrence net* is a net $ON \stackrel{\text{df}}{=} (B, E, G)$ where B is a set of *conditions* (places) and E is a set of *events* (transitions). It is assumed that: ON is acyclic (i.e. $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of G. $Min(ON)$ will denote the set of minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation*. Two nodes are *concurrent*, denoted $y \; co \; y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$. We also denote by $x \; co \; C$, where C is a set of pairwise concurrent nodes, the fact that a node x is concurrent to each node from C. Two events e and f are *separated* if there is an event g such that $e \prec g \prec f$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to P \cup T$ such that: $h(B) \subseteq P$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$; the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$; the restriction of h to $Min(ON)$ is a bijection between $Min(ON)$ and M_0; and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$. If $h(x) = y$ then we will often refer to x as *y-labelled*.

A *branching process* of Σ ([4]) is a quadruple $\pi \stackrel{\text{df}}{=} (B, E, G, h)$ such that (B, E, G) is an occurrence net and h is a homomorphism from ON to Σ. A branching process $\pi' = (B', E', G', h')$ of Σ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted by $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ.

A *configuration* of an occurrence net ON is a set of events C such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. The configuration $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of $e \in E$. A set of conditions B' such that for all distinct $b, b' \in B'$, $b \; co \; b'$, is called a *co-set*. A *cut* is a maximal (w.r.t. set inclusion) co-set. Every marking reachable from $Min(ON)$ is a cut.

Let C be a finite configuration of a branching process π. Then $Cut(C) \stackrel{\text{df}}{=} (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset of places $Mark(C) \stackrel{\text{df}}{=} h(Cut(C))$ is a reachable marking of Σ. A marking M of Σ is *represented* in π if the latter contains a finite configuration C such that $M = Mark(C)$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

A branching process π of Σ is *complete* if for every reachable marking M of Σ: (i) M is represented in π; and (ii) for every transition t enabled by M, there is a finite configuration C and an event $e \notin C$ in π such that $M = Mark(C)$, $h(e) = t$ and $C \cup \{e\}$ is a configuration. Although, in general, the unfolding of a finite bounded net system Σ may be infinite, it is possible to truncate it and obtain a finite complete prefix, Unf_Σ. [15] proposes a technique for this, based on choosing an appropriate set E_{cut} of *cut-off* events, beyond which the unfolding is not generated. One can show ([7,9]) that it suffices to designate an event e newly added during the construction of Unf_Σ as a cut-off event, if the already

input : $\Sigma = (N, M_0)$ — a bounded net system
output : Unf_Σ — a finite and complete prefix of Σ's unfolding

$Unf_\Sigma \leftarrow$ the empty branching process
add instances of the places from M_0 to Unf_Σ
$pe \leftarrow$ PotExt(Unf_Σ)
$cut\_off \leftarrow \emptyset$
while $pe \neq \emptyset$ **do**
  choose $e \in pe$ **such that** $[e] \in \min_\lhd\{[f] \mid f \in pe\}$
  if $[e] \cap cut\_off = \emptyset$
  then
    add e and new instances of the places from $h(e)^\bullet$ to Unf_Σ
    $pe \leftarrow$ PotExt(Unf_Σ)
    if e is a cut-off event of Unf_Σ **then** $cut\_off \leftarrow cut\_off \cup \{e\}$
  else $pe \leftarrow pe \setminus \{e\}$

Fig. 1. The unfolding algorithm presented in [7].

built part of the prefix contains a *corresponding* configuration C without cut-off events, such that $Mark(C) = Mark([e])$ and $C \lhd [e]$, where $\lhd$ is an *adequate order* on the finite configurations of a branching process (see [7] for the definition of $\lhd$).

The unfolding algorithm presented in [7,8] is parameterised by an adequate order $\lhd$, and can be formulated as in figure 1. It is assumed that the function PotExt finds the set of possible extensions of the already constructed part of a prefix, which can be defined in the following way (see [7]).

Definition 1. *Let π be a branching process of a net system Σ. A* possible extension *of π is a pair (t, D), where D is a co-set in π and t is a transition of Σ, such that $h(D) = {}^\bullet t$ and π contains no t-labelled event with the preset D.*

For simplicity, in figure 1 and later in this paper, we do not distinguish between a possible extension (t, D) and a (virtual) t-labelled event e with the preset D, provided that this does not create an ambiguity.

The efficiency of the algorithm in figure 1 heavily depends on a good adequate order $\lhd$, allowing early detection of cut-off events. It is advantageous to choose 'dense' (ideally, total) orders. [7,8] propose such an order for safe net systems, and show that if a total order is used, then the number of the non-cut-off events in the resulting prefix will never exceed the number of reachable markings in the original net system (though usually it is much smaller). Using a total order allows one to simplify some parts of the unfolding algorithm in figure 1, e.g., testing whether an event is a cut-off event can be reduced to a single look-up in a hash table if only local corresponding configurations are allowed (using non-local ones can be very time consuming, see [9]).

3 Finding Possible Extensions

Almost all the steps of the unfolding algorithm in figure 1 can be implemented quite efficiently. The only hard part is to calculate the set of possible extensions, PotExt(Unf_Σ), and we will make it the focus of our attention. As the decision version of the problem is NP-complete in the size of the already built part of the prefix ([10]), it is unlikely that we can achieve substantial improvements in the worst case for a single call to the PotExt procedure. However, the following approaches can still be attempted: (i) using heuristics to reduce the cost of a single call; and (ii) merging the common parts of the work performed to insert individual instances of transitions. An excellent example of a method aimed at reducing the amount of work is the improvement, proposed in [7], where a total order on configurations is used to reduce both the size of the constructed complete prefix and the number of calls to PotExt. Another method is outlined in [6,15], where the algorithm does not have to recompute all the possible extensions in each step: it suffices to update the set of possible extensions left from the previous call, by adding events consuming conditions from $e^\bullet$, where e is the last inserted event.

Definition 2. *Let π be a branching process of a net system Σ, and e be one of its events. A possible extension (t, D) of π is a* (π, e)-extension *if $e^\bullet \cap D \neq \emptyset$, and e and (t, D) are not separated.*

With this approach, the set *pe* in the algorithm in figure 1 can be seen as a priority queue (with the events ordered according to the adequate order $\lhd$ on their local configurations) and implemented using, e.g., a binary heap. The call to PotExt(Unf_Σ) in the body of the main loop of the algorithm is replaced by UpdatePotExt(pe, Unf_Σ, e), which finds all (π, e)-extensions and inserts them into the queue. Note that in the important special case of binary synchronisation, when the size of transition preset is at most 2, say ${}^\bullet t = \{h(c), p\}$ and $c \in e^\bullet$, the problem becomes equivalent to finding the set $\{c' \in h^{-1}(p) \mid c' \; co \; c\}$, which can be efficiently computed (the problem is vacuous when $|{}^\bullet t| = 1$). This technique leads to a further simplification since now we never compute any possible extension more then once, and so we do not have to add the cut-off events (and their postsets) into the the prefix being built until the very end of the algorithm. Hence, we can altogether avoid checking whether a configuration contains a cut-off event.

We now observe that in definition 2, e and (t, D) are not separated events, which basically suggests that any sufficient condition for being a pair of separated events may help in reducing the computational cost involved in calculating the set of (π, e)-extensions. In what follows, we identify two such cases.

In the pseudo-code given in [15], the conditions $c \in e^\bullet$ are inserted into the unfolding one by one, and the algorithm tries to insert new instances of transitions from $h(c)^\bullet$ with c in their presets. Such an approach can be improved as the algorithm is sub-optimal in the case when a transition t can consume more then one condition from $e^\bullet$. Indeed, t is considered for insertion after each condition from $e^\bullet$ it can consume has been added, and this may lead to a significant

overhead when the size of t's preset is large. Therefore, it is better to insert into the unfolding the whole post-set $e^\bullet$ at once, and use the following simple result, which essentially means that possible extensions being added consume as many conditions from $e^\bullet$ as possible (note that this results in an improvement whenever there is a (π, e)-extension, which can consume more than one condition produced by e).

Proposition 1. *Let e and f be events in the unfolding of a safe net system such that $f \in (e^\bullet)^\bullet$ and $h(e^\bullet \cap {}^\bullet f) \neq h(e)^\bullet \cap {}^\bullet h(f)$. Then e and f are separated.*

Corollary 1. *Let π be a branching process of a safe net system, e be an event of π, and (t, D) be a (π, e)-extension. Then $|e^\bullet \cap D| = |h(e)^\bullet \cap {}^\bullet t|$.*

Another way of reducing the number of calls to PotExt is to ignore some of the transitions from $(u^\bullet)^\bullet$, which the algorithm attempts to insert after a u-labelled event e. For in a safe net system, if the preset ${}^\bullet t$ of a transition $t \in (u^\bullet)^\bullet$ has non-empty intersection with ${}^\bullet u \setminus u^\bullet$, then t cannot be executed immediately after u. Therefore, in the unfolding procedure, an instance f of t cannot be inserted immediately after a u-labelled event e (though f *may* actually consume conditions produced by e, as shown in figure 2; note that in such a case e and f are separated).

Proposition 2. *Let e and f be events in the unfolding of a safe net system such that $f \in (e^\bullet)^\bullet$ and $({}^\bullet h(e) \setminus h(e)^\bullet) \cap {}^\bullet h(f) \neq \emptyset$. Then e and f are separated.*

Corollary 2. *Let π be a branching process of a safe net system, e be an event of π, and (t, D) be a (π, e)-extension. Then $({}^\bullet h(e) \setminus h(e)^\bullet) \cap {}^\bullet t = \emptyset$.*

In view of the above corollary, the algorithm may consider only transitions from the set $(h(e)^\bullet)^\bullet \setminus ({}^\bullet h(e) \setminus h(e)^\bullet)^\bullet$ rather than $(h(e)^\bullet)^\bullet$ as the candidates for insertion after e.

The resulting algorithm for updating the set of possible extensions after inserting an event e into the unfolding is fairly straightforward ([13]); moreover, it is possible not to maintain the concurrency relation, as suggested in [6], by rather to mark conditions which are not concurrent to a constructed part of a transition preset as unusable, and to unmark them during the backtracking.

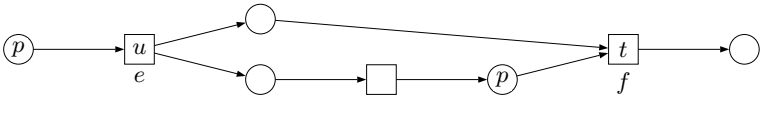


Fig. 2. A t-labelled event f cannot be inserted immediately after a u-labelled event e if $p \in ({}^\bullet u \setminus u^\bullet) \cap {}^\bullet t \neq \emptyset$, even though it can consume a condition produced by e.

Merging computation common to several calls. The presets of candidate transitions for inserting after an event e often have overlapping parts besides the places from $h(e)^\bullet$, and the algorithm may be looking for instances of the same places in the unfolding several times. To avoid this, one may identify the common parts of the presets, and treat them only once. The main idea is illustrated below.

Let e be the last event inserted into the prefix being built and $h(e)^\bullet = \{p\}$. Moreover, let t_1, t_2, t_3 and t_4 be possible candidates for inserting after e such that ${}^\bullet t_1 = \{p, p_1, p_2, p_3, p_4\}$, ${}^\bullet t_2 = \{p, p_1, p_2, p_3\}$, ${}^\bullet t_3 = \{p, p_1, p_2, p_3, p_5\}$, and ${}^\bullet t_4 = \{p, p_2, p_3, p_4, p_5\}$. The condition labelled by p in each case comes from the postset of e. To insert t_i, the algorithm has to find a co-set C_i such that e *co* C_i and $h(C_i) = {}^\bullet t_i \setminus \{p\}$ (if there are several such co-sets, then several instances of t_i should be inserted). By gluing the common parts of the presets, one can obtain a tree shown in figure 3(a), which can then be used to reduce the task of finding the co-sets C_i. Formally, we proceed as follows.

Definition 3. *Let u be a transition of a net system Σ and $U = (u^\bullet)^\bullet \setminus ({}^\bullet u \setminus u^\bullet)^\bullet$. A* preset tree *of u, PT_u, is a directed tree satisfying the following:*

- *Each vertex is labelled by a set of places, so that the root is labelled by $\emptyset$, and the sets labelling the nodes of any directed path are pairwise disjoint.*
- *Each transition $t \in U$ has an associated vertex v, such that the union of all the place sets along the path from the root to v is equal to ${}^\bullet t \setminus u^\bullet$ (different transitions may have the same associated vertex).*
- *Each leaf is associated to at least one transition (unless the tree consists of one vertex only).*

The weight *of PT_u is defined as the sum of the weights of all the nodes, where the* weight *of a node is the cardinality of the set of places labelling it.*

Having built a preset tree, we can use the algorithm in figure 4 to update the set of possible extensions, aiming at avoiding redundant work (sometimes there are gains even when PreMax $= 2$ and no two transitions have the same preset, see [13]). Note that we only need one preset tree PT_u per transition u of the net system, and it can be built during the preprocessing stage.

Building preset trees. Two problems which we now address are: (i) how to evaluate the 'quality' of preset trees, and (ii) how to efficiently construct them. If we use the 'totally non-optimised' preset tree shown in figure 3(b) instead of that in figure 3(a) as an input to the algorithm in figure 4, it will work in a way very similar to that of the standard algorithm trying the candidate transitions one-by-one. However, gluing the common parts of the presets decreases both the weight of the preset tree and the number of times the algorithm attempts to find new conditions concurrent to the already constructed part of event presets. This suggests that preset trees with small weight should be preferred. Such a 'minimal weight' criterion may be seen as rather rough, since it is hard to predict during the preprocessing stage which preset tree will be better, as different ones might be better for different instances of the same transition. Another problem

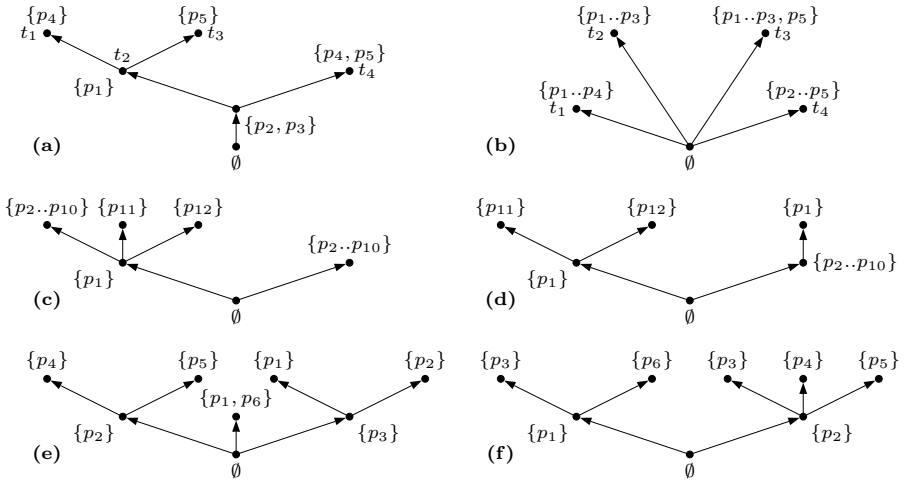


Fig. 3. An optimised **(a)** and non-optimised **(b)** preset trees of weight 7 and 15; **(c)** a tree of weight 21, produced by the bottom-up algorithm with $A_1 = \{p_1, \ldots, p_{10}\}$, $A_2 = \{p_2, \ldots, p_{10}\}$, $A_3 = \{p_1, p_{11}\}$, and $A_4 = \{p_1, p_{12}\}$ (p_1 was chosen on the first iteration of the algorithm), and **(d)** a tree of weight 13, corresponding to the same sets; **(e)** a tree of weight 8, produced by the top-down algorithm for the sets $A_1 = \{p_1, p_3\}$, $A_2 = \{p_1, p_6\}$, $A_3 = \{p_2, p_3\}$, $A_4 = \{p_2, p_4\}$, and $A_5 = \{p_2, p_5\}$ (the intersection $\{p_3\} = A_1 \cap A_3$ was chosen on the first iteration), and **(f)** a tree of weight 7, corresponding to the same sets.

procedure UPDATEPOTEXT(pe, Unf_Σ, e)
  $tree \leftarrow$ preset tree for $h(e)$ /* pre-calculated */
  $C \leftarrow$ all conditions concurrent to e
  COVER(C, $tree$, e, $\emptyset$)

procedure COVER(C, $tree$, e, $preset$)
  for all transitions t labelling the root of $tree$ **do**
    $pe \leftarrow pe \cup \{(t, (e^\bullet \cap h^{-1}({}^\bullet t)) \cup preset)\}$
  for all sons $tree'$ of $tree$ **do**
    $R \leftarrow$ places labelling the root of $tree'$
    for all co-sets $CO \subseteq C$ **such that** $h(CO) = R$ **do**
      COVER($\{c \in C \mid c \ co \ CO\}$, $tree'$, e, $preset \cup CO$)

Fig. 4. An algorithm for updating the set of possible extensions.

is that the reduction of the weight of a preset tree leads to the creation of new vertices and splitting of the sets of places among them, effectively decreasing the weight of a single node. This may reduce the efficiency of the heuristics, which potentially might be used for finding co-sets in the algorithm in figure 4. But this drawback is usually more than compensated for by the speedup gained

by merging the common parts of the work spent on finding co-sets forming the presets of newly inserted events.

Since there may exist a whole family of minimal-weight preset trees for the same transition, one could improve the criterion by taking into account the remark about heuristics for resolving the non-deterministic choice, and prefer minimal weight preset trees which also have the minimal number of nodes. Furthermore, we could assign coefficients to the vertices, depending on the distance from the root, the cardinality of the labelling sets of places, etc., and devise more complex optimality criterion. However, this may get too complicated and the process of building preset trees can easily become more time consuming then the unfolding itself. And, even if a very complicated criterion is used, the time spent on building a highly optimised preset tree can be wasted: the transition may be dead, and the corresponding preset tree will never be used by the unfolding algorithm. Therefore, in the actual implementation, we decided to adopt the simple 'minimal weight' criterion and, in the view of the next result, it was justifiable to implement a relatively fast greedy algorithm aiming at 'acceptably light' preset trees.

Proposition 3. *Building a minimal-weight preset tree is an NP-complete problem in the size of a Petri net, even if* PreMax $= 3$.

Proof. The decision version of this problem is to determine whether there exists a preset tree of the weight at most w, where w is given. It is in NP as the size of a preset tree is polynomial in the size of a Petri net, and we can guess it and check its weight in polynomial time.

The proof of NP-hardness is by reduction from the vertex cover problem. Given an undirected graph $G = (V, E)$, construct a Petri net as follows: take $V \cup \{p\}$ as the set of places, and for each edge $\{v_1, v_2\} \in V$ take a transition with $\{p, v_1, v_2\}$ as its preset. Moreover, take another transition t with the postset $\{p\}$ (note that all the other transitions belong to $(t^\bullet)^\bullet$). There is a bijection between minimal weight preset trees for t and minimal size vertex covers for G. Therefore, the problem of deciding whether there is a preset tree of at most a given size is NP-hard. □

In figure 5, we outlined simple bottom-up and top-bottom algorithms for constructing 'light' preset trees. In each case, the input is a set of sets of places $\{A_1, \ldots, A_k\} = \{{}^\bullet t \setminus u^\bullet \mid t \in U\} \cup \{\emptyset\}$ and, as it is obvious how to assign vertices to the transitions, we omit this part. $Tree(v, \{Tr_1, \ldots, Tr_l\})$ is a tree with the root v and the sons $Tr_1, \ldots, Tr_l$, which are also trees, and '$\cdot$' stands for a set of son trees if their identities are irrelevant.

The two algorithms do not necessarily give an optimal solution, but in most cases the results are acceptable. We implemented both, to check which approach performs better. The tests indicated that in most cases the resulting trees had the same weight, but sometimes a bad choice on the first step(s) causes the bottom-up approach to yield very poor results, as illustrated in figure 3(c,d). The top-down algorithm appeared to be more stable, and only in rare cases (see figure 3(e,f)) produced 'heavier' trees then the bottom-up one. Therefore, we will focus our attention on its efficient implementation.

function BUILDTREE($S = \{A_1, ..., A_k\}$) $\quad$ / * *bottom* − *up* * /
  $root \leftarrow \bigcap_{A \in S} A$
  $S \leftarrow \{A_1 \setminus root, \ldots, A_k \setminus root\}$
  $TS \leftarrow \emptyset$
  while $\bigcup_{A \in S} A \neq \emptyset$ **do** /* while there are non-empty sets */
    choose $p \in \bigcup_{A \in S} A$ **such that** $|\{A \in S \mid p \in A\}|$ is maximal
    $Tree(v, ts) \leftarrow$ BUILDTREE($\{A \setminus \{p\} \mid A \in S \wedge p \in A\}$)
    $TS \leftarrow TS \cup \{Tree(v \cup \{p\}, ts)\}$
    $S \leftarrow \{A \in S \mid p \notin A\}$
  return $Tree(root, TS)$

function BUILDTREE($\{A_1, ..., A_k\}$) $\quad$ / * *top* − *down* * /
  $TS \leftarrow \{Tree(A_1, \emptyset), \ldots, Tree(A_k, \emptyset)\}$
  while $|TS| > 1$ **do**
    choose $Tree(A', \cdot) \in TS$ and $Tree(A'', \cdot)$ in TS
        such that $A' \neq A''$ and $|A' \cap A''|$ is maximal
    $I \leftarrow A' \cap A''$
    $T_{\subset} \leftarrow \{Tree(B \setminus I, ts) \mid Tree(B, ts) \in TS \wedge I \subset B\}$
    $T_{=} \leftarrow \bigcup\{ts \mid Tree(I, ts) \in TS \wedge ts \neq \emptyset\}$
    $TS \leftarrow TS \setminus \{Tree(B, \cdot) \in TS \mid I \subseteq B\}$
    $TS \leftarrow TS \cup \{Tree(I, T_{\subset} \cup T_{=})\}$
  /* $|TS| = 1$ */
  return the remaining tree $Tr \in TS$

Fig. 5. Two algorithms for building trees.

A sketch of a possible implementation of the top-down algorithm for building preset trees is shown in figure 6. It computes all pairwise intersections of the sets A_i before the main loop starts, and then maintain this data structure. On each step, the algorithm chooses a set I of maximal cardinality from *Intersec*, and updates the variables TS and *Intersec* in the following way: (i) it finds all the supersets of I in TS, and removes them; (ii) it removes from *Intersec* all the intersections corresponding to these sets; (iii) the intersections of I with the sets remaining in TS are added into *Intersec*; and (iv) I is inserted into TS.

Proposition 4. *The worst case time complexity of the top-down algorithm is* $O(\text{PREMAX} \cdot k^2 \cdot \log k)$. *It is also possible to implement it so that the average case complexity is given by* $O(\text{PREMAX} \cdot k^2)$ *(see [13] for implementation details).*

It is essential for the correctness of the algorithm that *Intersec* is a multiset, and we have to handle duplicates in our data structure. It is better to implement this by maintaining a counter for each set inserted into *Intersec*, rather then by keeping several copies of the same set, since the multiplicity of simple sets (e.g., singletons or the empty set) can by very high. Moreover, if multiplicities are calculated, we often can reduce the weights of produced trees. The idea is to choose in figure 6 among the sets with maximal cardinality those which have the maximal number of supersets in TS (note that this would improve the tree in figure 3(e), forcing $\{p_2\}$ to be chosen on the first iteration). Such sets have

```
function BuildTree({A_1, ..., A_k})
    TS ← {Tree(A_1, ∅), ..., Tree(A_k, ∅)}
    /* Intersec is a multiset of sets */
    Intersec ← {A' ∩ A'' | A' ≠ A'' ∧ Tree(A', ·) ∈ TS ∧ Tree(A'', ·) ∈ TS}
    while |TS| > 1 do
        choose I ∈ Intersec such that |I| is maximal
        T_⊂ ← {Tree(B \ I, ts) | Tree(B, ts) ∈ TS ∧ I ⊂ B}
        T_= ← ⋃{ts | Tree(I, ts) ∈ TS ∧ ts ≠ ∅}
        for all Tree(A, ts) ∈ TS such that I ⊆ A do
            TS ← TS \ {Tree(A, ts)}
            for all Tree(B, ·) ∈ TS do
                Intersec ← Intersec \ {A ∩ B}
        for all Tree(A, ·) ∈ TS do
            Intersec ← Intersec ∪ {I ∩ A}
        TS ← TS ∪ {Tree(I, T_⊂ ∪ T_=)}
    /* |TS| = 1 */
    return the remaining tree Tr ∈ TS
```

Fig. 6. A top-down algorithm for building preset trees.

the highest multiplicity among the sets with the maximal cardinality. Indeed, each time this choice is made by the algorithm, the values of TS and *Intersec* are 'synchronised' in the sense that *Intersec* contains all pairwise intersections of the sets marking the roots of the trees from TS, with the proper multiplicities. Now, let $I \in$ *Intersec* be a set with the maximal cardinality, which has n supersets in TS (note that $n \geq 2$). The intersection of two sets can be equal to I only if they both are supersets of I. Moreover, since there is no set in *Intersec* with cardinality greater then $|I|$, the intersections of any two distinct supersets of I from TS is exactly I. Hence the multiplicity of I is $C_n^2 = n(n-1)/2$. This function is strictly monotonic for all positive n, and so there is a monotonic one-to-one correspondence between the multiplicities of sets with the maximal cardinality from *Intersec* and the numbers of their supersets in TS. Thus, among the sets of maximal cardinality, those having the maximal multiplicity have the maximal number of supersets in TS. One can implement this improvement without affecting the asymptotic running time given by proposition 4 ([13]).

4 Experimental Results

The results of our experiments are summarised in tables 1 and 2, where we use *time* to indicate that the test had not stopped after 15 hours, and *mem* to indicate that the test terminated because of memory overflow. They were measured on a PC with $Pentium^{TM}$ III/500MHz processor and 128M RAM. For comparison, the `ERVunfold 4.5.1` tool, available from the Internet, was used. The methods implemented in it are described in [6,7]; in particular, it maintains the concurrency relation.

The meaning of the columns in the tables is as follows (from left to right): the name of the problem; the number of places and transitions, and the average/maximal size of transition presets in the original net system; the number of conditions, events and cut-off events in the complete prefix; the time spent by the `ERVunfold` tool (in seconds); the time spent by our algorithm on building the preset trees and unfolding the net; the ratio $W_{rat} = W_{opt}/W$, where W_{opt} is the sum of the weights of the constructed preset trees, and W is the sum of the weights of the 'totally non-optimised' preset trees as in figure 3(b). This ratio may be used as a rough approximation of the effect of employing preset trees: $W_{rat} = 1$ means that there is no optimisation. Note that when transition presets are large enough, employing preset trees gives certain gains, even if this ratio is close to 1 (see, e.g., the DME(n) series).

We attempted (table 1) the popular set of benchmark examples, collected by J.C. Corbett ([3]), K. McMillan, S. Melzer, S. Römer (this set was also used in [6,9,10,12,16]), and available from K. Heljanko's homepage.

The transitions in these examples usually have small sizes of presets (in fact, they do not exceed 2 for most of the examples in table 1; the only example in this set with a big maximal preset is BYZ(4,1), but it in fact has only one transition with the preset of size 30, and one transition with the the preset of size 13; the sizes of the other transition presets in this net do not exceed 5). Thus, the advantage of using preset trees is not substantial, and `ERVunfold` is usually faster as it maintains the concurrency relation. But when the size of this relation becomes greater then the amount of the available memory, `ERVunfold` slows down because of page swapping (e.g., in FTP(1), GASNQ(5), and KEY(4) examples). As for our algorithm, it is usually slower for these examples, but its running time is acceptable. Moreover, sometimes it scales better (e.g., for the DME(n), ELEV(n) and MMGT(n) series).

In order to test the algorithms on nets with larger presets, we have built a set of examples RND(m, n) in the following way. First, we created m loops consisting of n places and n transitions each; the first place of each loop was marked with one token. Then 500 additional transitions were added to this skeleton, so that each of them takes a token from a randomly chosen place in each loop and puts it back on another randomly chosen place in the same loop (thus, the net has $m{\cdot}n$ transitions with presets of size 1 and 500 transitions with presets of size m). It is easy to see that the nets built in this way are safe. The experimental results are shown in table 2.

To test a practical example with large transition presets, we looked at a data intensive application (where processes being modelled compute functions depending on many variables), namely the priority arbiter circuit described in [1]. We generated two series of examples: SPA(n) for n processes and linear priorities, and SPA(m, n) for m groups and n processes in each group. The results are summarised in table 2. Our algorithm scales better and is able to produce much larger unfoldings. We expect that other areas, where Petri nets with large presets are needed, will be identified (such nets may result from net transformations, e.g. adding complementary places or converting bounded nets into safe ones, see [8]). But even for nets with small transition presets our algorithm is quite quick,

Table 1. Experimental results: nets with small transition presets.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	a/m $\|{}^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{rat}
Bds(1)	53	59	1.88/2	12310	6330	3701	1.30	<0.01	3.87	0.53
Byz(1,4)	504	409	3.33/30	42276	14724	752	126	0.14	231	0.71
Ftp(1)	176	529	1.98/2	178085	89046	35197	*time*	0.16	2625	0.52
Q(1)	163	194	1.89/2	16123	8417	1188	8.69	0.03	39.43	0.81
Dme(7)	470	343	3.24/5	9542	2737	49	6.37	0.19	7.28	0.93
Dme(8)	537	392	3.24/5	13465	3896	64	14.12	0.09	16.08	0.92
Dme(9)	604	441	3.24/5	18316	5337	81	27.78	0.11	31.82	0.92
Dme(10)	671	490	3.24/5	24191	7090	100	51.67	0.13	58.14	0.92
Dme(11)	738	539	3.24/5	31186	9185	121	89.18	0.16	98.96	0.92
Dpd(4)	36	36	1.83/2	594	296	81	0.01	<0.01	0.02	0.71
Dpd(5)	45	45	1.82/2	1582	790	211	0.04	<0.01	0.16	0.71
Dpd(6)	54	54	1.81/2	3786	1892	499	0.22	<0.01	0.83	0.71
Dpd(7)	63	63	1.81/2	8630	4314	1129	1.16	<0.01	5.49	0.71
Dpfm(5)	27	41	1.98/2	67	31	20	0.00	0.01	<0.01	1.00
Dpfm(8)	87	321	2/2	426	209	162	0.01	0.08	0.01	1.00
Dpfm(11)	1047	5633	2/2	2433	1211	1012	0.05	89.35	0.74	1.00
Dph(5)	48	67	1.97/2	2712	1351	547	0.10	<0.01	0.36	1.00
Dph(6)	57	92	1.98/2	14590	7289	3407	2.16	<0.01	9.74	1.00
Dph(7)	66	121	1.98/2	74558	37272	19207	57.43	0.01	263	1.00
Elev(2)	146	299	1.95/2	1562	827	331	0.02	0.13	0.14	0.60
Elev(3)	327	783	1.97/2	7398	3895	1629	0.61	1.59	2.73	0.60
Elev(4)	736	1939	1.99/2	32354	16935	7337	16.15	25.57	68.43	0.61
Furn(1)	27	37	1.65/2	535	326	189	0.01	<0.01	0.02	0.50
Furn(2)	40	65	1.71/2	4573	2767	1750	0.19	<0.01	0.54	0.44
Furn(3)	53	99	1.75/2	30820	18563	12207	8.18	<0.01	29.10	0.41
Gasnq(3)	143	223	1.97/2	2409	1205	401	0.09	0.03	0.36	0.96
Gasnq(4)	258	465	1.98/2	15928	7965	2876	4.54	0.10	18.45	0.97
Gasnq(5)	428	841	1.99/2	100527	50265	18751	785	0.32	817	0.98
Gasq(2)	78	97	1.95/2	346	173	54	<0.01	<0.01	0.02	0.93
Gasq(3)	284	475	1.99/2	2593	1297	490	0.11	0.12	0.40	0.97
Gasq(4)	1428	2705	2/2	19864	9933	4060	7.93	7.91	29.70	0.99
Key(2)	94	92	1.97/2	1310	653	199	0.06	0.01	0.15	0.93
Key(3)	129	133	1.98/2	13941	6968	2911	2.51	0.03	10.48	0.94
Key(4)	164	174	1.98/2	135914	67954	32049	6247	0.06	864	0.94
Mmgt(2)	86	114	1.95/2	1280	645	260	0.03	0.03	0.08	0.64
Mmgt(3)	122	172	1.95/2	11575	5841	2529	1.75	0.07	6.09	0.64
Mmgt(4)	158	232	1.95/2	92940	46902	20957	188	0.14	504	0.64
Rw(6)	33	85	1.99/2	806	397	327	0.01	0.01	0.01	1.00
Rw(9)	48	181	1.99/2	9272	4627	4106	0.21	0.03	0.34	1.00
Rw(12)	63	313	2/2	98378	49177	45069	14.46	0.10	15.30	1.00
Sync(2)	72	88	1.89/3	3884	2091	474	0.29	<0.01	1.38	0.91
Sync(3)	106	270	2.21/4	28138	15401	5210	14.15	0.06	74.84	0.77

Table 2. Experimental results: nets with larger transition presets.

Problem	Net			Unfolding			Time, [s]			
	$\|S\|$	$\|T\|$	a/m $\|{}^\bullet t\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*ERV*	*p-trees*	*Unf*	W_{rat}
Rnd(5,5)	25	525	4.81/5	55698	14029	11689	11.45	7.36	3.66	0.39
Rnd(5,6)	30	530	4.77/5	84451	21774	17269	31.43	8.68	12.21	0.44
Rnd(5,7)	35	535	4.74/5	144700	36019	28922	82.92	8.90	30.69	0.50
Rnd(5,8)	40	540	4.70/5	235600	56691	46559	196	8.79	62.96	0.54
Rnd(5,9)	45	545	4.67/5	304656	72895	59840	324	7.43	105	0.58
Rnd(5,10)	50	550	4.64/5	419946	98477	82279	554	9.07	160	0.60
Rnd(5,11)	55	555	4.60/5	573697	132344	112310	994	6.20	246	0.63
Rnd(5,12)	60	560	4.57/5	627303	145378	122465	1187	5.72	322	0.65
Rnd(5,13)	65	565	4.54/5	718762	166093	140147	1560	5.27	420	0.67
Rnd(5,14)	70	570	4.51/5	802907	185094	156417	1952	5.58	507	0.69
Rnd(5,15)	75	575	4.48/5	842181	195228	163722	6685	6.63	616	0.70
Rnd(5,16)	80	580	4.45/5	886158	206265	171957	*time*	7.10	717	0.71
Rnd(5,17)	85	585	4.42/5	987605	229284	191576	—	3.78	863	0.72
Rnd(5,18)	90	590	4.39/5	1025166	239069	198524	—	5.62	998	0.73
Rnd(10,2)	20	520	9.65/10	34884	7136	6125	12.46	7.34	1.14	0.25
Rnd(10,3)	30	530	9.49/10	1415681	153628	144548	1638	3.90	82	0.49
Rnd(10,4)	40	540	9.33/10	2344821	252320	237000	*mem*	3.51	207	0.59
Rnd(10,5)	50	550	9.18/10	2485903	271083	250600	—	7.90	331	0.64
Rnd(10,6)	60	560	9.04/10	2535070	280560	255010	—	11.32	485	0.67
Rnd(10,7)	70	570	8.89/10	2537646	285323	254767	—	11.91	663	0.70
Rnd(10,8)	80	580	8.76/10	2534970	289550	254000	—	14.84	872	0.72
Rnd(15,2)	30	530	14.21/15	1836868	135307	128358	*mem*	32.28	70.24	0.37
Rnd(15,3)	45	545	13.84/15	3750719	271074	255560	—	14.69	259	0.57
Rnd(15,4)	60	560	13.50/15	3787575	280560	257515	—	7.54	456	0.67
Rnd(15,5)	75	575	13.17/15	3795090	288075	257515	—	6.38	718	0.73
Rnd(20,2)	40	540	18.59/20	4744587	256197	245750	*mem*	46.71	176	0.43
Rnd(20,3)	60	560	17.96/20	5040080	280560	260020	—	16.36	427	0.61
Rnd(20,4)	80	580	17.38/20	5050100	290580	260020	—	9.03	771	0.71
Spa(4)	98	81	2.77/5	1048	421	96	0.04	0.01	0.07	0.72
Spa(5)	121	113	3.34/6	3594	1362	457	0.26	0.03	0.53	0.63
Spa(6)	144	161	4.20/7	13334	4860	2145	3.79	0.08	5.51	0.56
Spa(7)	167	241	5.38/8	52516	18712	9937	64.22	0.28	75.54	0.49
Spa(8)	190	385	6.82/9	216772	76181	45774	*time*	1.26	943	0.43
Spa(9)	213	657	8.35/10	920270	320582	209449	—	6.66	12571	0.38
Spa(2,1)	52	37	2.16/4	111	52	4	<0.01	<0.01	<0.01	0.87
Spa(2,2)	98	81	2.77/5	1206	476	110	0.04	0.01	0.10	0.72
Spa(2,3)	144	161	4.20/7	15690	5682	2512	5.53	0.08	8.28	0.56
Spa(2,4)	190	385	6.82/9	253219	88944	52826	*time*	1.29	1326	0.43
Spa(3,1)	75	57	2.40/4	324	141	19	0.01	<0.01	0.02	0.79
Spa(3,2)	144	161	4.20/7	15690	5682	2512	5.49	0.08	9.09	0.56
Spa(3,3)	213	657	8.35/10	1142214	398850	256600	*time*	6.67	20594	0.38
Spa(4,1)	98	81	2.77/5	1048	421	96	0.04	0.01	0.09	0.72
Spa(4,2)	190	385	6.82/9	253219	88944	52826	*time*	1.27	1326	0.43

and may be used if the size of the finite prefix is expected to be large (note that it was slower than `ERVunfold` for some of the examples because we did not maintain the concurrency relation, trading speed for a possibility of building large prefixes — in principle, maintaining concurrency relation is compatible with all the described heuristics).

Future work. We plan to develop an effective parallel algorithm for constructing large unfoldings. Another promising direction is to consider non-local correspondent configurations proposed in [9].

Acknowledgements. Proposition 3 and its proof are due to P. Rossmanith. We would like to thank A. Bystrov for his suggestion to consider dual-rail logics circuits and help with modelling priority arbiters. We would also like to thank J. Esparza, K. Heljanko, and the anonymous referees for helpful comments. The first author was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293.

References

1. A. Bystrov, D. J. Kinniment and A. Yakovlev: Priority Arbiters. Proc. *ASYNC 2000*, IEEE Computer Society Press (2000) 128–137.
2. E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8 (1986) 244–263.
3. J. C. Corbett: *Evaluating Deadlock Detection Methods*. Univ. of Hawaii at Manoa (1994).
4. J. Engelfriet: Branching processes of Petri Nets. *Acta Inf.* 28 (1991) 575–591.
5. J. Esparza: Decidability and Complexity of Petri Net Problems — An Introduction. *Lectures on Petri Nets I: Basic Models* Springer, LNCS 1491 (1998) 374–428.
6. J. Esparza and S. Römer: An Unfolding Algorithm for Synchronous Products of Transition Systems. Proc. *CONCUR'99*, Springer, LNCS 1664 (1999) 2–20.
7. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. *TACAS'96*, Springer, LNCS 1055 (1996) 87–106.
8. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* (2001) to appear.
9. K. Heljanko: Minimizing Finite Complete Prefixes. Proc. *CS&P'99* (1999) 83–95.
10. K. Heljanko: Deadlock and Reachability Checking with Finite Complete Prefixes. Report A56, Laboratory for Theoretical Computer Science, HUT, Espoo (1999).
11. K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fund. Inf.* 37 (1999) 247–268.
12. V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. CS-TR-711, Dept. of Computing Science, Univ. of Newcastle (2000).
13. V. Khomenko and M. Koutny: An Efficient Algorithm for Unfolding Petri Nets. CS-TR-726, Dept. of Computing Science, Univ. of Newcastle (2001).
14. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. *CAV'92*, Springer, LNCS 663 (1992) 164–174.
15. K. L. McMillan: *Symbolic Model Checking*. PhD thesis, CMU-CS-92-131 (1992).
16. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. *CAV'97*, Springer, LNCS 1254 (1997) 352–363.

A Static Analysis Technique for Graph Transformation Systems*

Paolo Baldan, Andrea Corradini, and Barbara König

Dipartimento di Informatica, Università di Pisa, Italia
{baldan,andrea,koenigb}@di.unipi.it

Abstract. In this paper we introduce a static analysis technique for graph transformation systems. We present an algorithm which, given a graph transformation system and a start graph, produces a finite structure consisting of a hypergraph decorated with transitions (Petri graph) which can be seen as an approximation of the Winskel style unfolding of the graph transformation system. The fact that any reachable graph has an homomorphic image in the Petri graph and the additional causal information provided by transitions allow us to prove several interesting properties of the original system. As an application of the proposed technique we show how it can be used to verify the absence of deadlocks in an infinite-state Dining Philosophers system.

1 Introduction

Graphs are very useful to describe complex structures in a direct and intuitive way. Graph Transformation Systems (GTSs) [18] add to the static description given by graphs a further dimension which models graph evolution via the application of rules, usually having local effects only. GTSs have been recognized to have fruitful applications in various fields of Computer Science [5], and specifically in the modelling and specification of concurrent and distributed systems [6].

As a high-level specification formalism for concurrent systems, GTSs are known to be more expressive than (Place/Transition) Petri nets, which can be seen, indeed, as GTSs acting on discrete graphs only (i.e., multisets of tokens) [3]. However, even if the theory of GTSs is nowadays well developed and a number of tools for the support of specifications based on this formalism have been developed, GTSs are not yet used as widely as Petri nets. One reason for this could be the lack of analysis techniques, which have been proven to be extremely effective for Petri nets. Verification and validation techniques play an important role during the design of the specification of a complex system, as they offer the designer the possibility to raise confidence in the quality of the specification, for example by allowing the early detection of logical errors.

While several static analysis techniques have been proposed for Petri nets, ranging from the calculus of invariants [16] to model checking based on finite

* Research partially supported by the EC TMR Network GETGRATS, by the ESPRIT Working Group APPLIGRAPH, and by the MURST project TOSCA.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 381–395, 2001.
© Springer-Verlag Berlin Heidelberg 2001

complete prefixes [13],[1] the rich literature on GTSs does not contain many contributions to the static analysis of such systems (see [11,12]).

In this paper we present an original analysis technique for a class of (hyper)graph transformation systems, which, given a system and a start hypergraph, extracts from them an *approximated unfolding*, which is a finite structure (called *Petri graph*) consisting of a hypergraph and of a P/T net over it. Both the graphical and Petri net components of the approximated unfolding can be used to analyze the original system. For example, we will show that every hypergraph reachable from the start graph can be mapped homomorphically to the (graphical component of the) approximated unfolding. Therefore, if a property over graphs is reflected by graph morphisms, then if it holds on the approximated unfolding it also holds on all reachable graphs. Among these properties we mention the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). Furthermore, the transitions of the Petri net component of the approximated unfolding can be seen as (approximated) occurrences of rules of the original graph transformation system, and indeed every reachable graph of the GTS corresponds (in a sense formalized later) to a reachable marking of the net. This allows one to prove other properties directly on the Petri net component, including upper and lower bounds on the number of times an edge with a certain label is present in a reachable graph and certain causal dependencies among rule applications. Notice that in general the net component of the approximated unfolding is neither safe nor acyclic; roughly one can say that, at least for certain properties, the analysis of a graph transformation system can be reduced to the analysis of a Petri net, which is a computationally less powerful model and for which the existing analysis techniques can be used.

The construction of the approximated unfolding of a graph transformation system is similar in spirit to the construction of the finite complete prefix [13] of a net, but more complex. Both are based on the unfolding construction, which in the case of nets [15] unwinds a Petri net into a branching occurrence net (a particularly simple Petri net satisfying suitable acyclicity and conflict freeness requirements), behaviourally equivalent to the original net. The unfolding cannot be used "directly" for verification purposes, since it is usually infinite. In the case of bounded nets, McMillan has observed in [13] that it is possible to truncate the unfolding in such a way that the resulting finite structure, the *finite complete prefix*, contains as much information as the unfolding itself, and can therefore be used for checking efficiently behavioural properties ([8,9,19]).

The unfolding construction has been generalized to graph transformation systems [17,2,1], and the technique we propose makes use of unfolding steps for generating the (finite) approximated unfolding, but the analogy with the finite prefix construction of nets ends here. In fact the GTSs we consider are not finite-state in general, hence, we must abandon the idea of finding a complete *finite* part of the unfolding, where every state reachable in the considered GTS has an *isomorphic* image. Even if we relax the last requirement, by asking only that

[1] We use the term "static analysis" in a quite wide meaning, as for example it is used in the community of the Static Analysis Symposia.

every reachable state has an *homomorphic* image in the constructed unfolding, since the states of the systems we consider are more structured (graphs *versus* multisets), it is not possible to rudely truncate the unfolding construction: at certain stages we have to merge parts of the unfolding already constructed. Because of this merging, the resulting structure is not acyclic (unlike the finite complete prefixes), and part of the information on the causality and concurrency of the system is lost. For what concerns state reachability, every state reachable in the original system is also reachable in the approximated unfolding, but we loose the converse implication (which instead holds for the finite complete prefix).

Technically, the algorithm that computes the approximated unfolding of a GTS is defined through two basic transformations, called *unfolding* and *folding* operations, which are applied as long as possible to the (Petri graph representing the) start graph of the system. Since both folding and unfolding are applied only if certain conditions are satisfied, the algorithm can be shown to terminate, a fact which guarantees that the resulting Petri graph is finite. Furthermore, although the proposed algorithm is non-deterministic, a local confluence property of the unfolding and folding transformations ensures that the approximated unfolding of a GTS is uniquely determined.

The paper is organized as follows. In Section 2 we introduce the class of GTSs on which our static analysis technique will be defined, as well as Petri graphs and some basic operations on them. The algorithm computing the approximated unfolding of a GTS is presented in Section 3, while Section 4 collects the main results about the algorithm, namely its termination, its confluence, and the fact that every reachable graph can be mapped to a reachable subgraph of the approximated unfolding. Section 5 illustrates the proposed method by applying it to the classical dining philosophers, both in a finite- and in an infinite-state variant. Section 6 concludes and hints at possible developments of the ideas presented in the paper.

2 Hypergraph Rewriting, Petri Nets, and Petri Graphs

In this section we first introduce the class of (hyper)graph transformation systems considered in the paper. Then, after recalling some basic notions for Petri nets, we will define Petri graphs, the structure combining hypergraphs and Petri nets, which will be used to approximate the behaviour of GTSs.

2.1 Graph Transformation Systems

In the following, given a set A we denote by A^* the set of finite strings of elements of A. Furthermore, if $f : A \to B$ is a function then we denote by $f^* : A^* \to B^*$ its extension to strings. Throughout the paper Λ denotes a fixed set of *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$.

Definition 1 (hypergraph). *A* (Λ-)hypergraph G *is a tuple* (V_G, E_G, c_G, l_G), *where* V_G *is a finite set of nodes,* E_G *is a finite set of edges,* $c_G : E_G \to V_G{}^*$ *is a connection function and* $l_G : E_G \to \Lambda$ *is the labelling function for edges satisfying* $ar(l_G(e)) = |c_G(e)|$ *for every* $e \in E_G$. *Nodes are not labelled.*

A node $v \in V_G$ is called isolated *if it is not connected to any edge, i.e. if there are no edges $e \in E_G$ and $u, w \in {V_G}^*$ such that $c_G(e) = uvw$.*

Let G, G' be (Λ-)hypergraphs. A hypergraph morphism *$\varphi : G \to G'$ consists of a pair of total functions $\langle \varphi_V : V_G \to V_{G'},\ \varphi_E : E_G \to E_{G'} \rangle$ such that for every $e \in E_G$ it holds that $l_G(e) = l_{G'}(\varphi_E(e))$ and ${\varphi_V}^*(c_G(e)) = c_{G'}(\varphi_E(e))$.*

In the sequel, when dealing with hypergraph morphisms we will often omit the subscripts V and E when referring to the components of a morphism φ.

Definition 2 (rewriting rule). *A* rewriting rule *r is a triple (L, R, α), where L and R are hypergraphs, called* left-hand side *and* right-hand side*, respectively, and $\alpha : V_L \to V_R$ is an injective mapping.*

A rule $r = (L, R, \alpha)$ is called basic *if l_L is injective, i.e., different edges in the left-hand side L have different labels, no node in L is isolated and no node in $V_R - \alpha(V_L)$ is isolated in R.*

In the following we will consider *only* basic rules. This restriction is not strictly needed, but makes the presentation simpler. For example, a morphism of a left-hand side into a hypergraph is completely determined by the image of its edges. Furthermore, to simplify the notation we will assume, without loss of generality, that $V_L \subseteq V_R$, $E_L \cap E_R = \emptyset$ and that the mapping α is the identity.

Intuitively, a rule $r = (L, R, \alpha)$ specifies that an occurrence of the left-hand side L can be "replaced" by R, according to the following definition.

Definition 3 (hypergraph rewriting). *Let $r = (L, R, \alpha)$ be a rewriting rule. A* match *of r in a hypergraph G is any morphism $\varphi : L \to G$. In this case we write $G \Rightarrow_{r,\varphi} H$ or simply $G \Rightarrow_r H$, where H is defined as follows: $V_H = V_G \uplus (V_R - V_L)$, $E_H = (E_G - \varphi(E_L)) \uplus E_R$, and if $\overline{\varphi} : V_R \to V_H$ is the obvious extension of φ then*

$$c_H(e) = \begin{cases} c_G(e) & \text{if } e \in E_G - \varphi(E_L) \\ \overline{\varphi}^*(c_R(e)) & \text{if } e \in E_R \end{cases}, \qquad l_H(e) = \begin{cases} l_G(e) \text{ if } e \in E_G - \varphi(E_L) \\ l_R(e) \text{ if } e \in E_R \end{cases}$$

Given a graph transformation system (GTS)*, i.e., a finite set of rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$. Furthermore we will denote the transitive closure of $\Rightarrow_{\mathcal{R}}$ by $\Rightarrow^*_{\mathcal{R}}$. A GTS with a start graph $(\mathcal{R}, G_{\mathcal{R}})$ is called a* graph grammar*.*

The application of the rule r to G at the match φ first removes from G the image of the edges of L. Then the graph G is extended by adding the new nodes in R (i.e., the nodes in $V_R - V_L$) and the edges of R. Observe that the (images of) the nodes in L are "preserved", i.e., not affected by the rewriting step.

The reader which is familiar with the double-pushout (DPO) approach [4] to graph rewriting would have recognized that our rules (L, R, α) can be seen as DPO rules $(L \hookleftarrow V_L \overset{\alpha}{\hookrightarrow} R)$ and that our notion of rewriting is equivalent to a DPO construction. Hence compared to general DPO rules $L \overset{\varphi_L}{\hookleftarrow} K \overset{\varphi_R}{\to} R$ we have that (i) K is discrete, i.e., it contains no edges, (ii) no two edges in the left-hand side L have the same label, (iii) the morphism φ_L is surjective on nodes, (iv) V_L and $V_R - \varphi_R(V_K)$ do not contain isolated nodes.

2.2 Petri Nets

In this subsection we fix some basic notation for Petri nets [16,14]. Given a set A we will denote by $A^{\oplus}$ the free commutative monoid over A, whose elements will be called *multisets* over A. Given a function $f : A \to B$, by $f^{\oplus} : A^{\oplus} \to B^{\oplus}$ we denote its monoidal extension. On multisets $m, m' \in A^{\oplus}$, we use some common relations and operations, like *inclusion*, defined by $m \leq m'$ when there exists $m'' \in A^{\oplus}$ such that $m \oplus m'' = m'$ and *difference*, which, in the same situation, is defined by $m' - m = m''$. Furthermore, for $m \in A^{\oplus}$ and $a \in A$ we write $a \in m$ for $a \leq m$. Often we will confuse a subset $X \subseteq A$ with the multiset $\bigoplus_{x \in X} x$.

Definition 4 (Petri net). *Let A be a finite set of action labels. An A-*labelled Petri net *is a tuple $N = (S, T, {}^{\bullet}(), ()^{\bullet}, p)$ where S is a set of places, T is a set of transitions, ${}^{\bullet}(), ()^{\bullet} : T \to S^{\oplus}$ assign to each transition its pre-set and post-set and $p : T \to A$ assigns an action label to each transition.*

The Petri net is called irredundant *if there are no distinct transitions with the same label and pre-set, i.e., if for any $t, t' \in T$*

$$p(t) = p(t') \ \wedge \ {}^{\bullet}t = {}^{\bullet}t' \quad \Rightarrow \quad t = t'. \tag{1}$$

A marked Petri net *is pair (N, m_N), where N is a Petri net and $m_N \in S^{\oplus}$ is the initial marking.*

The irredundancy condition (1) requires that two distinct transitions differ for the label or for the pre-set. This condition, in the case of branching processes, allows one to interpret each transition as an occurrence of firing of a transition in the original net, uniquely determined by its causal history (see [7]). Similarly, here it aims at avoiding the presence of multiple events which are indistinguishable for what regards the behaviour of the system. Hereafter *all* the considered Petri nets will be implicitly assumed irredundant, unless stated otherwise.

Definition 5 (causality relation). *Let N be a (marked) Petri net. The* causality relation $<_N$ *over N is the least transitive relation such that, for any $t \in T$, $s \in S$, we have (i) $s <_N t$ if $s \in {}^{\bullet}t$ and (ii) $t <_N s$ if $s \in t^{\bullet}$. For any $x \in S \cup T$ we define its sets of* causes $\lfloor x \rfloor = \{y \in S \cup T \mid y <_N x\}$.

Observe that, since we want to use Petri nets to represent the causality structure of a system only in an approximated way, no assumptions are made concerning the acyclicity of the net.

2.3 Petri Graphs

We next introduce the structure that we intend to use to approximate graph transformation systems, the so-called Petri graphs, which consist of an hypergraph and of a Petri net whose places are the edges of the graph.

Definition 6 (Petri graph). *Let $\mathcal{R}$ be a GTS. A Petri graph (over $\mathcal{R}$) is a tuple $P = (G, N, \mu)$ where G is a hypergraph, $N = (E_G, T_N, {}^{\bullet}(), ()^{\bullet}, p_N)$ is an $\mathcal{R}$-labelled Petri net where the places are the edges of G, and μ associates to*

each transition $t \in T_N$, with $p_N(t) = (L, R, \alpha)$, a hypergraph morphism $\mu(t) : L \cup R \to G$ such that

$$^\bullet t = \mu(t)^\oplus(E_L) \ \wedge \ t^\bullet = \mu(t)^\oplus(E_R) \tag{2}$$

A Petri graph for a graph grammar $(\mathcal{R}, G_\mathcal{R})$ is a pair (P, ι) where $P = (G, N, \mu)$ is a Petri graph for $\mathcal{R}$ and $\iota : G_\mathcal{R} \to G$ is a graph morphism. The multiset $\iota^\oplus(E_{G_\mathcal{R}})$ is called the initial marking *of the Petri graph. A marking $m \in E_G{}^\oplus$ will be called* reachable (coverable) *in (P, ι) if it is reachable (coverable) in the underlying Petri net.*

Condition (2) requires that each transition in the net can be viewed as an "occurrence" of a rule in $\mathcal{R}$. More precisely, if $p_N(t) = (L, R, \alpha)$ and $\mu(t) : L \cup R \to G$ is the morphism associated to the transition, then $\mu(t)_{|L} : L \to G$ must be a match of the rule in G such that the image of the edges of L in G coincides with the pre-set of t. Observe that, due to the assumption on the rules (no multiple labels and no isolated node in the left-hand side) the morphism $\mu(t)_{|L}$ (if it exists) is completely determined by $^\bullet t$. Then, the result of applying the rule to the considered match must be already in graph G, and the corresponding edges must coincide with the post-set of t. This is formalized by the condition over the image through $\mu(t)$ of the edges of R (note that the set E_R is seen as a multiset and $\mu(t)$ as a multiset function to take care of multiplicities).

Every hypergraph G can be considered as a Petri graph $[G] = (G, N, \mu)$ for $\mathcal{R}$, by taking N as the net with $S_N = E_G$ and no transitions. Similarly, $G_\mathcal{R}$ can be seen as Petri graph for $(\mathcal{R}, G_\mathcal{R})$ by taking as $\iota : G_\mathcal{R} \to G_\mathcal{R}$ the identity.

We now introduce a merging operation on Petri graphs which constructs the quotient of a Petri graph through an equivalence induced by a suitable relation.

Definition 7 (consistent and closed relation on a Petri graph). *Let $P = (G, N, \mu)$ be a Petri graph and let $\frown$ be a relation on $V_G \cup E_G \cup T_N$ (assume the sets V_G, E_G, T_N to be disjoint). We say that $\frown$ is* consistent *when (i) if $x \frown x'$ then $x, x' \in X$ for some $X \in \{V_G, E_G, T_N\}$, (ii) for all $e, e' \in E_G$ if $e \frown e'$ then $l_G(e) = l_G(e')$ and (iii) for all $t, t' \in T_N$, if $t \frown t'$ then $p_N(t) = p_N(t')$.*

A consistent relation $\frown$ over P is called closed *if for all $t, t' \in T_N$, $e, e' \in E_G$*

$$p_N(t) = p_N(t') = (L, R, \alpha) \ \wedge \ (\forall e \in E_L : \mu(t)(e) \frown \mu(t')(e)) \ \Rightarrow \ t \frown t' \tag{3}$$

$$t \frown t' \Rightarrow \ \forall e \in E_L \cup E_R : \mu(t)(e) \frown \mu(t')(e) \tag{4}$$

$$e \frown e' \ \wedge \ c_G(e) = v_1 \dots v_m \ \wedge \ c_G(e') = v'_1 \dots v'_m \ \Rightarrow \ \forall 1 \leq i \leq m : v_i \frown v'_i \tag{5}$$

To ensure that the quotient of a Petri graph with respect to a relation is well-defined and irredundant, the relation must be closed. Hence the simple observation below is essential for defining the merging operation.

Fact. Given any consistent relation $\frown$ over a Petri graph P there exists a least equivalence relation $\approx$ including $\frown$ and closed.

Definition 8 (Petri graph merging). *Let $P = (G, N, \mu)$ be a Petri graph and let $\frown$ be a consistent relation over P. Then the merging of P w.r.t. $\frown$, denoted*

by $P/\!\!/_{\frown}$, is the Petri graph (G', N', μ') defined as follows. Let $\approx$ be the least equivalence relation extending $\frown$ and closed in the sense of Definition 7. Then

$$G' = (V_G/_{\approx}, E_G/_{\approx}, c_{G'}, l_{G'}),$$

where $c_{G'}([e]_{\approx}) = [v_1]_{\approx} \ldots [v_n]_{\approx}$ and $l_{G'}([e]_{\approx}) = l_G(e)$ whenever $e \in E_G$ and $c_G(e) = v_1 \ldots v_n$. Furthermore $N' = (E_{G'}, T_N/_{\approx}, {}^{\bullet}(), ()^{\bullet}, p_{N'})$, where ${}^{\bullet}[t]_{\approx} = \bigoplus_{e \in {}^{\bullet}t} [e]_{\approx}$, $[t]_{\approx}{}^{\bullet} = \bigoplus_{e \in t^{\bullet}} [e]_{\approx}$ and $p_{N'}([t]_{\approx}) = p_N(t)$ whenever $t \in T_N$. For each $t \in T_N$ the morphism $\mu'([t]_{\approx})$ is defined by $\mu([t]_{\approx})(x) = [\mu(t)(x)]_{\approx}$ for any graph item x in the rule $p_N(t)$.

Given a graph morphism $h : H \to G$ we will denote by $h/\!\!/_{\frown} : H \to G'$ the corresponding morphism, defined by $h/\!\!/_{\frown}(x) = [h(x)]_{\approx}$ for any $x \in V_H \cup E_H$.

The merging operation can be extended to sets of Petri graphs. Let $P_i = (G_i, N_i, \mu_i)$, with $i \in \{1, \ldots, n\}$, be Petri graphs and assume that the sets V_{G_i}, E_{G_i}, T_{N_i} are pairwise disjoint. Then the componentwise union $P = P_1 \cup \ldots \cup P_n$ is a Petri graph. A relation $\frown$ over $P_1, \ldots, P_n$ is called consistent (closed) if it is a consistent (closed) relation over P. Given a consistent relation over $P_1, \ldots, P_n$, we define the merging $\{P_1, \ldots, P_n\}/\!\!/_{\frown} = P/\!\!/_{\frown}$.

3 Algorithm Computing the Approximated Unfolding

In this section we describe an algorithm which computes the approximated unfolding of a graph grammar. Given a graph grammar, the algorithm produces a *finite* Petri graph such that every graph reachable in the grammar corresponds, in a sense formalized later, to a marking which is reachable in the Petri graph.

Let $(\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar. Its ordinary unfolding [17,2] is constructed inductively beginning from the start graph and then applying at each step in all possible ways the rules, without deleting the left-hand side, and recording each occurrence of a rule and each new graph item generated in the rewriting process. As a result one obtains an acyclic branching graph grammar describing the behaviour of $(\mathcal{R}, G_{\mathcal{R}})$. In particular every reachable graph embeds in (a concurrent subgraph of) the graph produced by the unfolding construction.

The unfolding is usually infinite, also in the case of finite-state systems. Here, to ensure that the our algorithm produces a finite structure, we consider—besides the *unfolding rule*, which extends the graph by simulating the application of a rule without deleting its left-hand side—a *folding rule*, which allows us to "merge" two occurrences of the left-hand side of a rule whenever they are, in a sense made precise later, one causally dependent on the other.

Definition 9 (folding operation). *Let $P = (G, N, \mu)$ be a Petri graph for a GTS $\mathcal{R}$. Let $r = (L, R, \alpha) \in \mathcal{R}$ be a rule and let $\varphi', \varphi : L \to G$ be matches of r in G. Let $\frown$ be the relation over P defined as follows: for every $e \in E_L$*

$$\varphi'(e) \frown \varphi(e).$$

The folding *of P at the matches φ', φ is the Petri graph $\mathsf{fold}(P, r, \varphi', \varphi) = P/\!\!/_{\frown}$. If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define $\mathsf{fold}((P, \iota), r, \varphi', \varphi) = (P/\!\!/_{\frown}, \iota/\!\!/_{\frown})$.*

To introduce the unfolding operation, we first need to fix some notation. If t is a transition and $r = (L, R, \alpha)$ is a rule we will write $P(t, r)$ to denote the Petri graph $(L \cup R, N, \mu)$ where $N = (E_{L \cup R}, \{t\}, {}^\bullet t = E_L, t^\bullet = E_R, p_N(t) = r)$ and $\mu(t) = id_{L \cup R}$. Whenever we can find a match of rule r in a given Petri graph, the unfolding operation extends the Petri graph by merging $P(t, r)$ at the match.

Definition 10 (unfolding operation). *Let $P = (G, N, \mu)$ be a Petri graph for a GTS $\mathcal{R}$. Let $r = (L, R, \alpha) \in \mathcal{R}$ be a rule and let $\varphi : L \to G$ be a match of r in G. Let $\frown$ be the relation over $\{P, P(t, r)\}$ defined as follows: for every $e \in E_L$*

$$\varphi(e) \frown e.$$

The unfolding *of P with rule r at match φ is the Petri graph $\mathsf{unf}(P, r, \varphi) = \{P, P(t, r)\}/\!\!/_\frown$. If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_\mathcal{R})$, in the same situation, we define $\mathsf{unf}((P, \iota), r, \varphi) = (\{P, P(t, r)\}/\!\!/_\frown, \iota/\!\!/_\frown)$.*

We can now describe the algorithm which produces the approximated unfolding of a given graph grammar. The algorithm generates a sequence of Petri graphs, beginning from the start graph and applying, non-deterministically, at each step, a folding or unfolding operation, until none of such steps is admitted.

Definition 11 (approximated unfolding). *Let $(\mathcal{R}, G_\mathcal{R})$ be a graph grammar. The algorithm generates a sequence $(P_i, \iota_i)_{i \in \mathbb{N}}$ of Petri graphs as follows.*

(Step 0) *Initialize $(P_0, \iota_0) = ([G_\mathcal{R}], id_{G_\mathcal{R}})$.*

(Step $i + 1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i. Choose non-deterministically one of the following actions*

⋆ *Folding: Find a rule $r = (L, R, \alpha)$ in $\mathcal{R}$ and two matches $\varphi', \varphi : L \to G_i$ of r such that*

- *$\varphi^\oplus(E_L)$ is a coverable marking in P_i;*
- *there exists a transition $t \in T_{N_i}$ such that*

$$p_{N_i}(t) = r \ \wedge \ {}^\bullet t = {\varphi'}^\oplus(E_L) \ \wedge \ \forall e \in \varphi^\oplus(E_L) : (e \in {}^\bullet t \ \vee \ t <_{N_i} e). \quad (6)$$

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{fold}((P_i, \iota_i), r, \varphi', \varphi)$.

⋆ *Unfolding: Find a rule $r = (L, R, \alpha)$ in $\mathcal{R}$ and a match $\varphi : L \to G_i$ such that*

- *$\varphi^\oplus(E_L)$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^\bullet t = \varphi^\oplus(E_L)$ and $p_{N_i}(t) = r$;*
- *there is no other match $\varphi' : L \to G_i$ satisfying condition (6).*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

If no folding or unfolding step can be performed, the algorithm terminates. The resulting Petri graph (P_i, ι_i) is called the approximated unfolding *of $(\mathcal{R}, G_\mathcal{R})$ and denoted by $\mathcal{U}(\mathcal{R}, G_\mathcal{R})$.*

Condition (6) basically states that we can fold two matches of a rule r whenever the first one has been already unfolded producing a transition t, and the second match depends on the first one, in the sense that any edge in the second match is already in the first one or causally depends on t. Roughly, the idea is that we should not unfold a left-hand side again, if we have already done the same unfolding step in its past, since this might lead to infinitely many steps. There are some similarities, to be further investigated, with the work in [10] where the sets of descendants and of normal forms of term rewriting systems are approximated by constructing an approximation automaton.

The coverability of a marking can be decided by computing the coverability tree of the net, as described in [16]. If this gets too costly, the condition of coverability can be relaxed or checked in an approximated way, a choice which does not compromise the result of correctness (see Proposition 12), but only reduces the "precision" of the algorithm: it will generate a worse approximation, where less properties of the given GTS can be proved.

4 Correctness, Termination, and Confluence of the Algorithm

We show that the algorithm described in the previous section is correct, namely that every reachable graph of a grammar is represented in the approximated unfolding produced by the algorithm. Furthermore the algorithm is terminating and confluent. Hence, by a classical result, its result is uniquely determined.

Correctness. We first show that the computed Petri graph is an appropriate approximation of the given graph grammar, in the sense that for any graph reachable in the graph grammar, there is a morphism into the approximated unfolding such that the image of its edge set corresponds to a reachable marking.

Proposition 12. *Let $(\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and assume that the algorithm computing the approximated unfolding terminates producing the Petri graph $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}}) = ((U, N, \mu), \iota)$.*

Then for every graph G with $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ there exists a morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$. Furthermore, if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{U}(\mathcal{R}, G_{\mathcal{R}})$.

Termination. The basic result towards the proof of termination shows that it is not possible to perform infinitely many unfolding steps, without having the folding condition satisfied at some stage. This property is independent of the graph structure and can be proved by considering only the causality structure of a Petri graph, as expressed by the underlying Petri net. More formally, we show that in any infinite Petri net, satisfying suitable acyclicity and well-foundedness requirement, there exists a pair of transitions t, t' (called a folding pair) such that the pre-set of t' is dependent on t in the sense of Condition (6) in Definition 11. Let us start formalizing the notion of folding pair.

Definition 13. *Let $N = (S, T, {}^\bullet(), ()^\bullet, p)$ be a Petri net. A* folding pair *in N is a pair of transitions $t, t' \in T$ such that $t \neq t'$, $p(t) = p(t')$ and for all $s \in {}^\bullet t'$ either $s \in {}^\bullet t$ or $t <_N s$.*

The next key lemma ensures that in any infinite net obtained by applying only unfolding steps there exists a folding pair.

Lemma 14. *Let $N = (S, T, {}^\bullet(), ()^\bullet, p)$ be an infinite irredundant Petri net, labelled over a finite set A, and satisfying the following conditions:*

- *for any $x \in S \cup T$ the set $\lfloor x \rfloor$ (the causes of x) is finite;*
- *the set $Min(N) = \{s \mid \lfloor s \rfloor = \emptyset\}$ is finite, i.e., only finitely many places have an empty set of causes;*
- *the relation $<_N$ is acyclic;*
- *the pre-set ${}^\bullet t$ of each transition is a set (rather than a proper multiset);*
- *for $t, t' \in T$ with $p(t) = p(t')$ it holds that $|{}^\bullet t| = |{}^\bullet t'|$.*

Then net N contains a folding pair.

Proof (Sketch). The core of the proof shows that if $Q \subseteq T$ is a set of transitions with the same action label a, then either there is a folding pair in Q or we can remove almost all elements of Q from N in a way that the resulting net remains infinite, i.e., there exists a set $Q' \subseteq Q$ such that $Q - Q'$ is finite and and the net obtained from N removing Q and all its causal consequences is infinite.

Then the result can be proved by induction on the number of labels that occur infinitely often in N. □

The above lemma ensures that in our algorithm a folding step will be eventually performed. We have yet to show termination of the algorithm.

Proposition 15. *The algorithm computing the approximated unfolding (see Definition 11) terminates for every graph grammar $(\mathcal{R}, G_\mathcal{R})$.*

Confluence. In order to prove that the algorithm produces a uniquely determined result, independently of the order in which folding and unfolding steps are applied, we show that the rewriting relation on Petri graphs induced by folding and unfolding steps is locally confluent. The following proposition only holds if we consider irredundant Petri nets.

Proposition 16. *Let us write $(P, \iota) \dashrightarrow (P', \iota')$ whenever (P, ι) can be transformed into (P'', ι'') by either a folding or an unfolding step applied under the corresponding condition (see the algorithm in Definition 11) and (P'', ι'') is isomorphic to (P', ι'), i.e., equal up to injective renaming of the edges, nodes and transitions.*

Let $(P, \iota) \dashrightarrow (P_i, \iota_i)$ for $i \in \{1, 2\}$. Then there is a Petri graph (P', ι') such that $(P_i, \iota_i) \dashrightarrow^* (P', \iota')$.

Since for a rewriting system local confluence and termination imply confluence we conclude the following result.

Proposition 17. *For any input $(\mathcal{R}, G_\mathcal{R})$ the algorithm computing the approximated unfolding terminates with a result $\mathcal{U}(\mathcal{R}, G_\mathcal{R})$ unique up to isomorphism.*

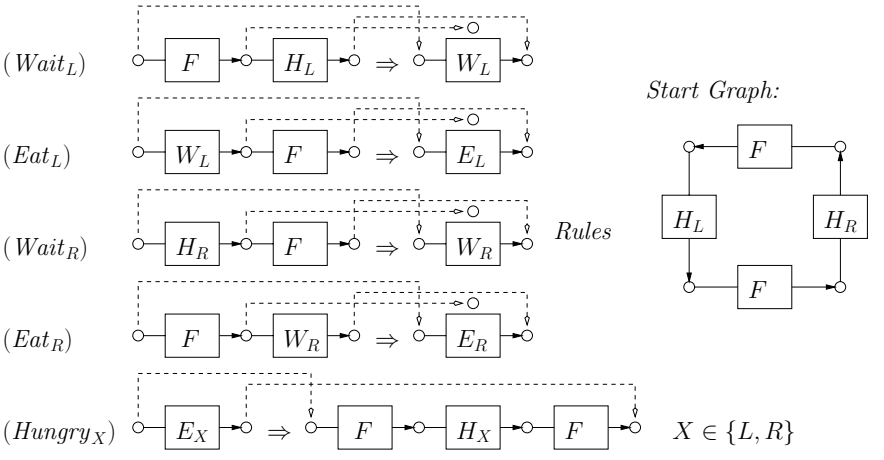


Fig. 1. A graph grammar modelling the dining philosophers (finite-state version).

5 The Approximated Unfolding at Work: Checking Absence of Deadlocks for Dining Philosophers

In order to illustrate our method, in this section we show how it can be applied to a well-known example, the dining philosophers system, which is presented in two versions, finite- and infinite-state.

Let us start with the classical finite-state version of the problem. Assume that sitting at the table are a left-handed philosopher and a right-handed philosopher with two forks between them. Our method is also applicable to instances of the problem with a greater number of philosophers. The restriction to two philosophers only avoids that the involved graphs become very large and hard to draw.

A philosopher, modelled by a binary edge, cycles through states H_X (hungry), W_X (waiting for the second fork), E_X (eating) where $X \in \{L, R\}$ depending on whether the philosopher is left- or right-handed. The thinking state is omitted. A fork is also represented by a binary edge labelled F. The system is described by the set of rewriting rules and by the start graph depicted in Fig. 1. A rule (L, R, α) is drawn in the form $L \Rightarrow R$, where edges are depicted by square boxes which are connected to a source node (the first node) and a target node (the second node). The mapping α is indicated by dashed arrows.

The algorithm in Definition 11 produces the Petri graph (a) in Fig. 2. Transitions are depicted by small rectangles and the connection to their pre-sets and post-sets is indicated by dashed arrows.

The algorithm terminates after six unfolding steps and four folding steps. Two unfolding steps which apply rules $(Wait_L)$ and (Eat_L), respectively, to edge H_L with the corresponding forks, give rise to edge E_L. Then a further unfolding step using rule $(Hungry_L)$ unfolds this edge into a graph consisting of two edges labelled F and one edge labelled H_L. But this graph consists of two

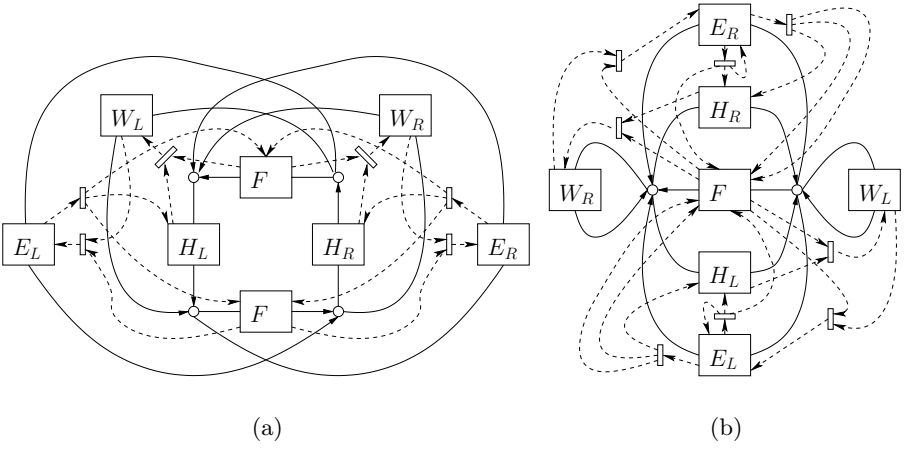


Fig. 2. Approximated unfoldings as Petri graphs: (a) dining philosophers, finite-state version; (b) dining philosophers, infinite-state version.

left-hand sides of previously applied rules and the edges are causally dependent on the corresponding transitions. Hence two folding steps can be applied, that merge the three edges (F, H_L and F) of the newly unfolded graph with the original edges from which they were derived. A symmetric reasoning applies for edge H_R.

We would like to prove that no deadlocks can occur in the system. First observe that any reachable graph is a cycle and, since an eating philosopher can always be reduced, a deadlocked state is necessarily a cycle including only hungry and waiting philosophers, where no forks are to the left of a left-handed hungry or a right-handed waiting philosopher and no forks are to the right of a right-handed hungry and a left-handed waiting philosopher. The absence of cycles is a property reflected by graph morphisms. Thus we can try to verify the absence of deadlocked states by analyzing cycles in the hypergraph associated to the approximated unfolding. To this aim we consider such graph as a finite-state automaton over the alphabet $\Sigma = \{F, H_X, W_X, E_X \mid X \in \{L, R\}\}$—with nodes as states and edges as transitions—and declare one of the four nodes as the initial and final state, thereby obtaining the languages L_{nw} (northwest node), L_{ne} (northeast node), L_{sw} (southwest node), L_{se} (southeast node). In this way we obtain all possible cycles of forks and philosophers as a regular language. By declaring, e.g., the northeast node as initial and final node we obtain the following language:

$$L_{ne} = (((FH_L + W_L)(E_R H_L)^* F + E_L)(W_R H_L (E_R H_L)^* F)^* H_R)^*.$$

An additional analysis of the Petri net would of course reveal that only a small finite subset of L_{ne} will ever occur, but here this is not needed for the analysis.

The language of all cycles allowing for the application of a rewrite rule is

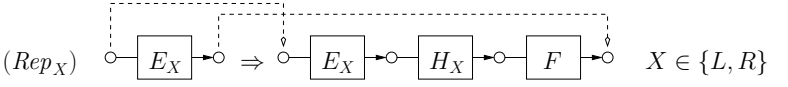
$$L_{lhs} = \Sigma^* E_L \Sigma^* + \Sigma^* E_R \Sigma^* + \Sigma^* F H_L \Sigma^* + H_L \Sigma^* F + \Sigma^* H_R F \Sigma^* +$$

$$F\Sigma^* H_R + \Sigma^* W_L F\Sigma^* + F\Sigma^* W_L + \Sigma^* F W_R \Sigma^* + W_R \Sigma^* F.$$

The language of all cycles which may occur but which do not allow the application of any rewriting rule can be now computed as $(L_{nw} \cup L_{ne} \cup L_{sw} \cup L_{se}) - L_{lhs} = \lambda$, i.e., the empty word. It is immediately clear that the circle of philosophers will never disappear entirely and thus we can conclude that no deadlocks will ever occur.

It is worth observing that if we forget about the graphical structure of the Petri graph, considering only the underlying Petri net, then we obtain a classical Petri net model of the dining philosophers. Therefore, in this case, the absence of deadlocks can be proved also by analyzing the Petri net underlying the approximated unfolding with classical Petri net techniques.

Now, in order to make things more interesting, we extend the example to an infinite-state system by adding a rule (Rep_X) which allows an eating philosopher to reproduce, creating another hungry philosopher with an adjacent fork.



Observe that we can reuse the unfolding of the finite-state case and continue by unfolding the edges E_R and E_L using the two new rules. A sequence of further unfolding and folding steps causes the causes the two pairs of opposite nodes in the square to collapse, ending up with Petri graph (b) in Fig. 2.

Again we would like to prove that no deadlocks can occur. By declaring the left-hand node as initial and final state, we obtain the following language:

$$L_{left} = (W_R^*((H_L + H_R)W_L^*(F + E_L + E_R))^*)^*$$

while using the right-hand node in the same role, we obtain the language:

$$L_{right} = (W_L^*((F + E_L + E_R)W_R^*(H_L + H_R))^*)^*.$$

The language of all cycles which may occur but which do not allow the application of a rewriting rule can be now computed as $(L_{left} \cup L_{right}) - L_{lhs} = W_L^* + W_R^*$. Then, an analysis of the Petri net underlying the approximated unfolding reveals that actually no marking which consists of tokens exclusively in W_L or of tokens exclusively in W_R is reachable from the initial marking which consists of two tokens on F and one token on H_L and H_R each. Hence the system will never reach a deadlock.

Observe that in this case the analysis of the underlying Petri net by itself is not sufficient. In fact the Petri net can deadlock: we start from the initial marking and after the firing of two transitions we obtain a marking with one token on W_R and one token on W_L, where no further firing is possible.

6 Conclusion

We have presented a static analysis technique for graph transformation systems which produces a finite structure, called Petri graph, combining hypergraphs

and Petri nets, which approximates the graphs which are reachable in the original grammar. Such a structure can be used to check safety properties, like the absence of deadlocks, in the original system.

An interesting question which has only been brushed in the paper, concerns the techniques which should be used to extract information from a computed Petri graph. It is certainly possible to reuse most of the well-established analysis techniques developed for Petri nets in the literature, such as coverability trees. However, as observed in the example, also the graphical structure underlying a Petri graph might play an essential role when establishing a property of the system. Since every graph reachable in the original grammar can be mapped to the approximated unfolding through a graph morphism, all properties which are reflected by graph morphisms can be checked on the approximated unfolding. We are currently investigating a syntactical characterization of such a class of properties. Other interesting issues are the use of methods from formal language theory (as hinted at in the example) and of model checking techniques.

Another question is the following: what can we do when we fail to prove a property? Obviously, it might still be the case that the considered property holds of the system, but this fact cannot be derived from the approximated unfolding where we have lost too much information by over-approximating. A partial solution could be to refine the description of the system, by computing a better approximation of the "complete" unfolding. This can be done by delaying folding steps and unfolding the Petri graph a bit further, "freezing" some parts of the approximated unfolding in order to avoid that a folding step leads to confuse them with other parts. A sequence of subsequently better approximations should converge to the whole, usually infinite, unfolding. In connection to this it would be interesting to determine which kind of systems can be "approximated" in an exact way—maybe by variations of the folding condition—one candidate being certainly Petri nets.

It is our aim to extend the proposed analysis technique to more general forms of graph rewriting, e.g., to the general double-pushout approach. In this case, since also edges might be preserved by a rewriting rule, the Petri net underlying a Petri graph cannot be simply an ordinary net, but it will be necessary to resort to contextual nets as in [1].

Acknowledgements: We are grateful to Javier Esparza for his insightful suggestions and to Alin Stefanescu and Stefan Schwoon who helped us with the finite-automaton tool used to compute the languages in Section 5. We are also grateful to anonymous referees for their valuable comments.

References

1. P. Baldan. *Modelling concurrent computations: from contextual Petri nets to graph grammars.* PhD thesis, Department of Computer Science, University of Pisa, 2000. Available as technical report n. TD-1/00.
2. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.

3. A. Corradini. Concurrent Graph and Term Graph Rewriting. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer Verlag, 1996.
4. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer Verlag, 1979.
5. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
6. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.
7. J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.
8. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.
9. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96*, volume 1055 of *LNCS*, pages 87–106. Springer-Verlag, 1966.
10. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proceedings 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 151–165. Springer Verlag, 1998.
11. M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems*. PhD thesis, Technische Universität Berlin, 2000.
12. B. König. A general framework for types in graph rewriting. In *Proc. of FST TCS 2000*, volume 1974 of *LNCS*, pages 373–384. Springer-Verlag, 2000.
13. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
14. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
15. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
16. W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
17. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
18. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
19. W. Vogler, A. Semenov, and A. Yakovlev. Unfolding and finite prefix for nets with read arcs. In *Proceedings of CONCUR'98*, volume 1466 of *LNCS*, pages 501–516. Springer-Verlag, 1998.

Local First Search – A New Paradigm for Partial Order Reductions

Peter Niebert[1], Michaela Huhn[2], Sarah Zennou[1], and Denis Lugiez[1]

[1] Laboratoire d'Informatique de Marseille
Université de Provence – CMI
39, rue Joliot-Curie / F-13453 Marseille Cedex 13
{niebert,zennou,lugiez}@cmi.univ-mrs.fr
[2] Institut für Software, TU Braunschweig, Gausstr. 11,
D-38023 Braunschweig, M.Huhn@tu-bs.de

Abstract. Partial order reductions are an approved heuristic method to cope with the state explosion problem, i.e; the combinatory explosion due to the interleaving representation of a parallel system. The partial order reductions work by providing sufficient criteria for building only a part of the full transition system on which the verification algorithms still compute the correct result for verifying *local properties*.
In this work, we present a new reduction method with a completely different justification and functioning: We show that under very realistic assumptions, local properties can be verified considering paths only corresponding to partial orders with very few maximal elements. Then we use this observation to derive our *local first search* algorithm. Our method can be understood as a hybrid between partial order reductions and the McMillan unfolding approach.
Experiments justify the practicality of the method.

1 Introduction

Model checking as an automatic method for proving simple system properties or finding witness executions of faulty systems suffers from the well known state explosion problem [CG87]: Typically, the number of states explored by naive algorithms is exponential in the size of the system description, so that often this automatic approach is limited to very small systems. However, the explosion of the number of states is typically due to a redundancy of the global state oriented interleaving semantics that can be circumvented for certain cases by heuristic methods, in particular: *Computing with symbolic representations* (e.g. ROBDDs or timed automata); *compositional and abstraction techniques*; *partial order methods*. The latter are aimed to reduce the state explosion which is due to parallelism. In this class, there are two prominent approaches:

Partial order reduction techniques (see e.g. [Val89,Pel93]), which try to exploit "diamond" properties to make savings in verification. They are based on a notion of equivalent executions and aim to cut redundant branches (and whole sub state spaces), where these cuts are justified by the existence of uncut

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 396–410, 2001.
© Springer-Verlag Berlin Heidelberg 2001

equivalent branches. Partial order reduction techniques have been applied with success notably to deadlock detection and model checking of certain (equivalence robust) linear time temporal properties without next operator(LTL-X).

The difficulty about partial order reduction methods as in [Pel93] is that for many systems it is too costly to compute optimal reductions (computing optimal reductions is as hard as solving the reachability problem, PSPACE-complete). Instead, sufficient criteria are used for correct reductions which in practice sometimes lead to relatively small savings.

Unfolding based methods (see e.g. [McM92,ERV96]): Rather than partially exploring an interleaving transition system, these methods directly construct partial order representations of executions with sharing. Instead of computing (successor) states, the unfolding approach is based on computing possible event extensions: An event is an instance of a transition and it depends on one or several immediate predecessor events. Using a notion of equivalence of events (based on the state corresponding to the first possible occurrence of that event) and an order among events, a *complete finite prefix* of the set of all events is defined and can be computed.

The complete finite prefix is a uniquely defined object and often quite small compared to the systems calculated by partial order reductions. However, the prefix method also suffers from several problems: (1) the computation of event extensions of the partially computed prefix is in general NP-complete in the size of the constructed part and (2) the prefix contains no states or transitions, just events with a precedence order, thus excluding the immediate application of interleaving based verification algorithms.

Problem (1) can be partially circumvented for certain applications, notably for the case of 1-safe Petri nets, where the pre-sets of transitions have small bounds (2 or 3). The current implementations of the unfolding procedure rely upon such restrictions.

Concerning problem (2), dedicated model checking algorithms for more involved properties have been proposed, e.g. a simple fragment of CTL with operators AG and EF only [Esp94] and LTL-X [Wal98,EH00], and some event based logics [HNW98]. Prefix based algorithms for such logics are much more involved than their counterparts on transition systems and indeed some of the published algorithms have turned out to contain subtle bugs or to be inefficient.

The origin of the present work was indeed an attempt to improve the efficiency of some of these algorithms.

In this work, we develop a partial order verification method which can be seen as a hybrid between classical partial order reduction methods unfolding based methods: We propose to build indeed a reduced transition systems with an orientation on the number of maximal events of corresponding partial order executions, a heuristic from the unfolding approach. We base the reduction on the two key observations of the experience with the unfolding approach: That its success depends on relatively local communication (bounded presets, bounded number of processes in synchronization) and that there are relatively few states corresponding to partial order executions with just one maximal element.

Exploiting structural properties, we show that all partial order executions with (for example) one maximal elements can be approximated adding one el-

ement after another in passing through intermediate partial order executions with a number of maximal elements logarithmic in the number of parallel components of the system only. It allows us to define a reduced transition system which passes through relatively local states (with a bounded number of maximal elements in the corresponding partial order). The proof of this structural property is based on observations linked to the Strahler number of trees [Str52] which are known in computer science in the context of optimal register allocation for the evaluation of binary expressions [FRV79].

We exploit this observation in order to prune transitions violating the bound in an extended transition system (which keeps track of additional information about the maximal elements). The resulting systems we compute contain a superset of states of the McMillan prefix, but with additional states and with a transition relation between them. They thus allow to apply interleaving based verification algorithms without further manipulation. Moreover, it is much easier to compute extensions in our approach and the total cost of construction is linear in the size of the computed transition system. The price is obviously that the computed structure can be much bigger than the prefix.

For reachability applications, instead of computing the system up to the bound, a heuristic search method constructing states with few maximal elements first is also proposed: *Local first search*. The idea behind the heuristic is that for many real systems, the theoretical bound is never reached.

The paper is structured as follows: In Section 2 we introduce an abstract model of parallel finite state systems in terms of Mazurkiewicz trace theory. In particular, we formalize the notion of parallelism and bounded communication in terms of a dependency relation of actions. In Section 3, we explore the structure of partial order executions and prove the logarithmic bound (Theorem 1). In Section 4, we show how the theoretical result of Section 3 can be used in model checking (the reduction technique with heuristic improvements). In Section 5, we report on current experimental results. Conclusions and future directions are outlined in Section 6.

2 Parallel Finite State Systems

The reduction and search technique we propose in this paper applies to a wide variety of system descriptions which involve parallelism. The idea is always that the single components of the system are relatively small, but that their parallel composition leads to a combinatoric explosion of states resulting from the product of the local state spaces. In order to explain the concepts of our method, we use a well known running example.

The dining philosophers. This example models processes communicating through shared variables as follows: n philosophers sit around a table with n forks, and each philosopher has a plate of spaghetti in front of him. Philosopher P_i either thinks (state q_i^t) or eats (state q_i^e). To eat, a philosopher takes the fork disposed at his right-hand side (action a_i^l) and the fork at his left-hand side (action a_i^l). To each fork i we associate a variable F_i indicating if it is free ($F_i = 1$) or not ($F_i =$

0). Once a philosopher has finished to eat, he puts back the right fork (action b_i^r) and the left fork (action b_i^l). Left-handed (resp. right-handed) philosophers always perform left (resp. right) actions first when ambidextrous philosophers have no required order. We assume that fork i is the right fork for philosopher i and fork $i+1$ is the left fork for philosopher i. Figure 1 describes the parallel composition of a ambidextrous philosopher with a right-handed philosopher. In this system the communication is achieved by the means of the shared variables F_i.

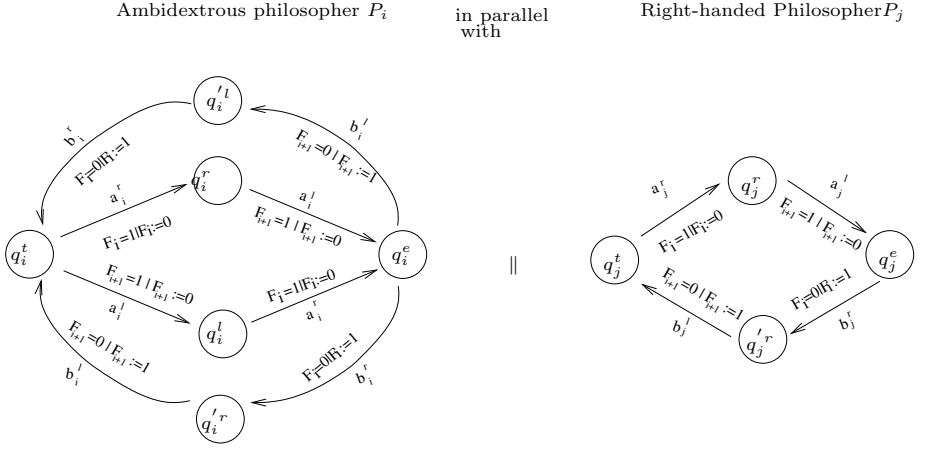


Fig. 1. The dining philosophers

For instance an execution of the n-philosophers may give rise to the sequence of actions $a_1^l a_3^r a_3^l a_1^r \ldots$ which we can describe as P_1 *takes its left fork,* P_3 *takes its right fork,* P_3 *takes its left fork,* P_1 *takes its right fork,* $\ldots$ An equivalent sequence corresponding to another interleaving is: $a_3^r a_3^l a_1^l a_1^r \ldots$. Both sequences result from an execution where the first philosopher takes its forks and starts to eat and the third one does the same concurrently. This shows that several (many) interleaved sequences represent the same concurrent execution.

Finite transition systems.

Definition 1. *Given a finite set of actions Σ, a finite state transition system is a triple $T = (S, \rightarrow, s_0)$ with S a finite set of states, $s_0 \in S$ the initial state, $\rightarrow \subseteq S \times \Sigma \times S$ a transition relation.*

From now on, we restrict our study to *deterministic* transition systems i.e. $\rightarrow$ is a function from $S \times \Sigma$ to S. By introducing new action names, we can transform a non-deterministic system into a deterministic one. This transformation doesn't modify the properties (mainly the reachability problem) we are interested in.

The *language L(T)* of execution sequences of a transition system T is the set of words $w = a_1 a_2 \ldots a_n$ of Σ^* such that there exists states $s_i \in S$, $i = 0, \ldots, n$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$.

A system of dining philosophers can be given a transition system semantics as follows: The states are tuples of local states for each philosopher; Transitions are the transitions of individual philosophers, just one at a time; Actions of taking forks depend on whether the forks are available or not.

Transition systems and the dependence relation. Looking at the example of the dining philosophers, one can see that some actions can be safely performed in any order without compromising the reachability property: For instance if P_1 performs a_1^l (take its left fork) then P_2 performs a_2^r (take its right fork) or if they do it the reverse order by performing a_2^r followed by a_1^l, we still reach the same state. This will be expressed by stating that a_1^l and a_2^r are independent.

A *dependence relation* D on Σ is a symmetric reflexive relation on Σ. The complement $I = \Sigma \times \Sigma \setminus D$ of a dependence relation is an (irreflexive and symmetric) relation called an *independence relation*. $\equiv_I$ denotes the least congruence on Σ^* such that $ab \equiv_D ba$ for all $a, b \in \Sigma$ such that $a\ I\ b$ (i.e. $a \not D\ b$). The equivalence classes $[w]_D$ of $\equiv_D$ are called *Mazurkiewicz traces* [DR95] and the pair (Σ, D) is called a partially commutative alphabet.

Definition 2. *The (deterministic) transition system T respects the dependence relation D iff*

- *for every $s \xrightarrow{a} s_1 \xrightarrow{b} s_2$ with $a\ I\ b$ there exists a uniquely defined s_1' with $s \xrightarrow{b} s_1' \xrightarrow{a} s_2$,*
- *for every $s \xrightarrow{a} s_1$ and $s \xrightarrow{b} s_1'$ with $a\ I\ b$ there exists s_2 with $s_1 \xrightarrow{b} s_2$ (and consequently $s_1' \xrightarrow{a} s_2$).*

For the philosophers, one can define a dependency relation as follows: All actions belonging to one philosopher i $(a_i^l, a_i^r, a_i^e, \dots)$ are mutually dependent, moreover actions concerning a fork between two philosophers (a_i^r, a_{i+1}^l) are mutually dependent.

Many examples of dependence relations in various modeling frameworks exist: In process algebras, dependency results from communication over shared channels; in Petri nets, transitions sharing places in the presets or postsets may be dependent.

In [Pel93], it is pointed out that dependency (in use for partial order reduction) need not have parallelism as only source. For instance, two operations "X:=X+1" and "X:=X+2" do also satisfy the commutativity properties and can hence be considered independent, although they touch the same variable. In contrast, "X:=X+1" and "X:=X*2" will normally not satisfy the diamond properties.

In the case of the ambidextrous philosopher, the two operations a_i^l and a_i^r also commute.

Closure of the language of actions. Transition systems respecting a dependence relation also satisfy a closure property under $\equiv_D$, more precisely if $w \in L(T)$ and $v \equiv_D w$ then $v \in L(T)$. This is the basis of *model checking with representative* which reduces $T = (S, \to, s_0)$ to some (simpler) $T' = (S', \to', s_0')$ such that each

class $[w]_D$ has a representative v in T'. This approach relies on the preservation of many interesting linear time and reachability properties under $\equiv_D$.

The character of a partially commutative alphabet. The following definition gives a precise meaning to informal concepts like the degree of parallelism and of communication in the framework of partially commutative alphabets.

Definition 3. *Let D be a dependence relation on Σ,*

- *we say that (Σ, D) has* parallel degree *m if m is the maximal number of pairwise independent elements of Σ,*
  (i.e. $m = max\{|A| \mid A \subseteq \Sigma$ and $a, b \in A, a \neq b \implies a\ I\ b\}$)
- *we say that the communication is* n-bounded *if n is the maximal number of pairwise independent elements which all depend on the same element, i.e.*
  $n = max\{|B| \mid B \subseteq \Sigma$ and
  $\forall b, b' \in B, b \neq b' \implies b\ I\ b'$ and
  $\exists c \in \Sigma$ s.t. $\forall b \in B,\ c\ D\ b\}$.

The pair (m, n) is called the character *of (Σ, D).*

The parallel degree of a system with n philosophers is n (for instance, all actions of taking the left fork are mutually independent). Intuitively, all philosophers could pick up their left fork simultaneously. The communication among the philosophers is 2-bounded: For instance, the operations a_1^l and a_2^r are mutually independent, but they both depend on a_2^l.

It is a generally observable fact that in many formalisms for concurrent systems the resulting models have a character (m, n) where n tends to be small compared to m. For instance, many process algebras restrict communication to pairs of dual *send* and *receive* actions, leading to a 2-bounded alphabet. The dependency relations resulting from Petri-nets are bounded by the number of presets and postsets of transitions, which are often very small compared to the size of the entire net.

3 The Structure of Partial Order Executions

It is a well known fact of Mazurkiewicz trace theory [DR95] that there is a one-to-one correspondence between equivalence classes $[w]_D$ defined by a dependence relation D and classes of finite labeled partial-orders $(E, \leq, \lambda)$ such that

(1) For any $e, f \in E$ with $\lambda(e)\ D\ \lambda(f)$ we have $e \leq f$ or $f \leq e$.
(2) $\leq$ is equal to the transitive closure of $\leq \cap \{(e, f) \mid \lambda(e)\ D\ \lambda(f)\}$.

In principle, a word can be seen as a total order and the partial order corresponding to an equivalence class $[w]_D$ has all equivalent words $w' \equiv_D w$ as linearisations. These partial orders can thus be seen as an abstract representation of executions, where the fact that two elements are unordered means that they could have occurred in any order (or in parallel).

In this section, we state some properties of partial order executions on the basis of the character of the underlying partially commutative alphabet. These properties will constitute the basis of our method for analyzing concurrent systems.

The character of a partial order. The *immediate successor* relation $\prec$ is defined by $e \prec f$ iff $e < f$ and $\forall g, e \leq g \leq f \implies g = e$ *or* $g = f$. We say that f is an immediate successor of e and that e is an immediate predecessor of f. We define the character of $(E, \leq)$ in a way similar to the definition of the character of a partially commutative alphabet.

Definition 4. *Let $(E, \leq)$ be a partial finite order.*
Let m be the maximal number of pairwise incomparable elements of $(E, \leq)$, i.e. $m = max\{|A| \mid A \subseteq E \wedge \forall e, f \in A \,.\, e \leq f \Rightarrow e = f\}$. Let n denotes the maximal number of immediate predecessors of an element of E, i.e. $n = max\{|A| \mid A \subseteq E \wedge \exists f \in E \forall e \in A \,.\, e \prec f\}$.
The pair (m, n) is called the character *of $(E, \leq)$.*

The following proposition relates execution sequences and partial orders via their respective characters.

Proposition 1. *Let (Σ, D) be a partially commutative alphabet of character (m, n) and let $(E, \leq, \lambda)$ be a labeled partial order corresponding to an equivalence class $[w]_D$ (i.e. $(E, \leq, \lambda)$ satisfies properties (1) and (2) from above).*

Then $(E, \leq)$ has character (m', n') with $m' \leq m$ and $n' \leq n$. Conversely, there exists a word $v \in \Sigma^*$ such that the partial order corresponding to $[v]_D$ has character (m, n).

Proof. Assume a set $M \subseteq E$ with $|M| > m$. Supposing that for all e, f we have $\lambda(e)\ I\ \lambda(f)$ then this gives a set of $|M|$ mutually independent letters, a contradiction to the character of the partially commutative alphabet. Hence, there exist $e, f \in M$, $e \neq f$, such that $\lambda(e)\ D\ \lambda(f)$. Hence, either $e \leq f$ or $f \leq e$ and M is not an anti chain. Hence, the maximal anti chain in M has size $m' \leq m$.

Assume now an element f and a set $\{e_1, ..., e_{n'}\}$ of distinct immediate predecessors of f. Hence $\lambda(e_i)\ D\ \lambda(f)$ for all i, because $e \prec f$ implies $\lambda(e)\ D\ \lambda(f)$. The latter follows from the fact that a set of generators (via transitive closure) of a partial order must contain the immediate predecessors (these cannot be generated) and the condition (2) on D respecting partial orders, the set of dependently labelled ordered pairs is a set of generators.

Moreover, two immediate predecessors of some element in a partial order cannot be ordered (hence are independently labeled). Hence, an element f and a set of predecessors E as in Definition 4 immediately give rise to a letter a and a set of letters B as required in Definition 3. It follows that $n' \leq n$.

For the converse direction (the existence of a word v with the partial order corresponding to $[v]_D$ having character (m, n)) construct $v = v_1 v_2$: Take $v_1 = a_1 \ldots a_m$ for a maximally sized set of mutually independent letters $\{a_1, \ldots, a_m\}$ and $v_2 = b_1 \ldots b_n c$ with $B = \{b_1, \ldots, b_n\}$ a maximally sized set of mutually independent letters all depending of a letter c.

The main theorem. The character of a partial order can be used to enumerate efficiently the partial order, as stated by the following theorem.

Theorem 1. *Let $(E, \leq)$ be a finite, nonempty partial order of character (m, n). Furthermore, let $(E, \leq)$ have $k \leq n$ maximal elements.*

Then there exists an enumeration $(e_1, \ldots, e_{|E|})$ of $E = \{e_1, \ldots, e_{|E|}\}$ such that for every $e, f \in E$ if $e \leq f$ then e occurs in the sequence before f and every $(E = \{e_1, \ldots, e_i\}, \leq \cap E \times E)$ has at most 1 maximal element if $n = 1$, and at most $\lfloor (n-1) log_n(m) + 1 \rfloor$ maximal elements if $n > 1$.

Proof. The proof is by induction on triples $(|E|, m, k)$ in the lexicographical order. The cases $|E| = 1$, $m = 1$ or $k = 1$ are obvious. As for the rest, we restrict ourselves to the simpler case of $n = 2$ (and as consequence $k = 2$) in which the formula simplifies to $\lfloor log_2(m) + 1 \rfloor$, but which shows well the reasoning applied: Let e_1, e_2 be the maximal elements of E and let $E_i := E \setminus \{e \mid \exists j \neq i \,.\, e \leq e_j\}$. Note that E_1 and E_2 are disjoint and moreover $E_1 \times E_2 \cap (\leq \cup \geq) = \emptyset$, that is, elements of E_1 and elements of E_2 are mutually unordered. As a consequence, the union of an antichain in E_1 and an antichain in E_2 is an antichain in E, hence has $\leq m$ elements.
We can conclude that the maximal size antichains of either E_1 or E_2 have at most $\frac{m}{2}$ elements. Let us say that this is the case for E_2, i.e. $(E_2, \leq \cap E_i \times E_i)$ has character (m', n') with $n' \leq n$ and $m' \leq \lfloor \frac{m}{2} \rfloor$ and by induction we obtain the bound $\lfloor log_2(m') + 1 \rfloor \leq \lfloor log_2(m) - 1 + 1 \rfloor$ for the enumeration of E_2. Also by induction (on $|E|$), we obtain the bound $\lfloor log_2(m) + 1 \rfloor$ for the enumeration of $E \setminus E_2$. If we assemble the two enumerations – first enumerating $E \setminus E_2$ and then E_2 – this gives us as bound the maximum for the two parts of the enumeration. Note that while enumerating E_2 we will always have e_1 as additional maximal element, leading to the bound $\lfloor log_2(m) - 1 + 1 \rfloor + 1 = \lfloor log_2(m) + 1 \rfloor$ as desired.

For the more general case of $n > 2$, a separation in to pairwise unordered sets $E_1, \ldots, E_n$ is necessary and the analysis inequations involving the logarithm gets more complicated, whence the factor $(n - 1)$ in the formula.

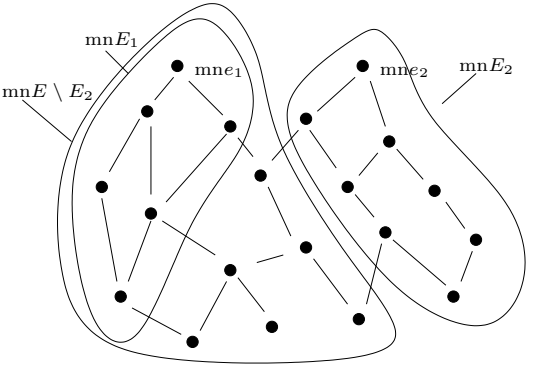


Fig. 2. A partial order with character $(5, 2)$ and two maximal elements

Remark 1. There is a strong link between the reasoning bound calculated in Theorem 1 and Strahler numbers of trees [Str52], that – while invented for the classification of rivers – occur in many areas of computer science. For the case of binary trees (forests), our bound corresponds to the bound of the maximal Strahler number for any tree with less than m leaves (vertices). An application similar to ours in reasoning concerns results on register requirements for the evaluation of binary (arithmetic) expressions [FRV79]: The requirement for registers is limited by the Strahler number of the expression.

The difference in reasoning in our setting is that m is the limited maximal size of anti chains, not the number of elements. The best way to see the link between Theorem 1 with Strahler numbers is to consider the partitioning of E into $E \setminus E_2$ and E_2 or into $E \setminus E_1$ and E_1 in the proof as a non-deterministic recursive procedure of turning the partial order into a tree. The bound we search is then a bound for the minimal Strahler number for any tree obtainable in this way.

We remark that the bound $\lfloor (n-1)\log_n(m) + 1 \rfloor$ is sharp[1] by giving an example where all enumerations require at least $\lfloor (n-1)\log_n(m) + 1 \rfloor$ maximal elements.

Example 1. Let $T(n, p)$ be the complete n-ary tree of height p. It defines a partial order of character $(m = n^p, n)$ since the maximal anti-chain is the set of the n^p leaves and each element but the leaves has n immediate successors. From now, we say *enumeration* for an *enumeration* satisfying the conditions of Theorem 1. Then any enumeration of this partial order requires at least $(n-1)p+1$ maximal elements. The proof is by induction on p.

4 Application to Model Checking

In this section, we explain how the results of the previous section can be used to obtain valid and efficient model checking techniques for parallel finite state systems.

Local and distributed properties. We propose a reduction method aimed on two applications: Reachability analysis and linear time model checking. In either case, the method is made for *local properties* as introduced as follows.

Definition 5. *Let $T = (S, \rightarrow, s_0)$ be a deterministic transition system over Σ, respecting a dependency relation D. For a given set $P \subseteq S$ (called* property*), the set of* visible actions *$V_P \subseteq \Sigma$ is the set of all labels $a \in \Sigma$ such that there exist $s_1, s_2 \in S$ with $(s_1, a, s_2) \in \rightarrow$ and either $s_1 \in P$ and $s_2 \notin P$ or $s_2 \in P$ and $s_1 \notin P$. For a set $\mathcal{P}$ of properties, we define $V_{\mathcal{P}} = \bigcup\{V_P \mid P \in \mathcal{P}\}$.*

A set of properties $\mathcal{P}$ has parallel degree *n iff $(V_{\mathcal{P}}, D\,|_{V_{\mathcal{P}} \times V_{\mathcal{P}}})$ has parallel degree n. A set of properties is called* local *iff it has parallel degree* 1.

[1] this doesn't imply that a better bound can't be given for certain cases

The idea of visible actions [Pel93] is that of actions that may affect a property of a set of properties of interest. The naming of local (sets of) properties is due to the typical case of properties of one process in a network: These are properties that depend only on the local state of the process in question, and this state only changes by transitions involving this process, thus mutually dependent transitions.
For the philosophers, typical local properties are: *philosopher i is eating*, *the fork between philosophers i and $i+1$ is free*. It is easy to see that any two transitions affecting any of these properties are mutually dependent. A typical property of parallel degree 2 is *philosopher* 1 *and philosopher* 3 *are eating*. A clearly non-local property is *deadlock* (the set of states without any transition), as can arise for the philosophers all picking up the left fork simultaneously. It has parallel degree n.

Proposition 2. *For a D respecting transition system $T = (S, \rightarrow, s_0)$, a property P of parallel degree n is reachable (i.e. there exists a state $s \in P$ reachable from s_0 iff there exists an execution sequence $a_1 \dots a_k$ leading to a state $s' \in P$) such that the partial order corresponding to $[a_1 \dots a_k]_D$ has at most n maximal elements, all of which are labelled with visible actions.*

For the proof, consider a minimal partial order of actions leading to such a property: It cannot contain a maximal element that is not a visible action!

Combining Proposition 2 with Theorem 1, we thus only have to construct "all" partial order executions with a number of elements below the theoretical bound stated in the theorem. However, there infinitely many partial order executions. In order to obtain a finite search space, we propose to take two contradictory steps: First, we blow up the state space by augmenting the states with additional (and potentially very costly) bookkeeping information, then we apply reductions to this blown up state space. As we will show, the reduction effect of the second step can over compensate the inflating effect of the first step in some cases, leading on the whole to substantial reductions in comparison to the original transition system.

Label tracking. We start with the following observation: For two execution sequences w_1, w_2 leading to the same state s and exposing the same set $M \subseteq \Sigma$ of labels of maximal elements in the corresponding partial orders, the same transitions (s, a, s') are possible (since these only depend on s) and the sets M_1' and M_2' of labels of maximal elements in the extended partial orders are the same: $h(a)$ is added and all elements in M_i' in dependence with $h(a)$ are removed.
Instead of representing all of the (partial order) history of executions, it is thus sufficient to keep track of the set of labels marking the maximal elements.

Definition 6 (maximal label tracking transition system). *Let $T = (S, \rightarrow, s_0)$ be a D respecting transition system over Σ. Let furthermore (Γ, D') be a second partially commutative alphabet and $h : \Sigma \rightarrow \Gamma$ a homomorphism with $a\ D\ b$ iff $h(a)\ D'\ h(b)$. The* maximal label tracking transition system (MLTTS) *$ML(T, \Gamma) = (S', \rightarrow', s_0')$ is a transition system defined as follows: $S' = S \times 2^{\Gamma}$;*

$s'_0 = (s_0, \emptyset)$; $(s_1, M_1) \stackrel{a}{\rightarrow}' (s_2, M_2)$ *iff* $s_1 \stackrel{a}{\rightarrow} s_2$ *and* $M_2 = \{h(a)\} \cup \{m \in M_1 \mid h(a)\ I'\ m\}$.

The interpretation of a property P *of* T *on* $ML(T, \Gamma)$ *is given by* $P' = \{(s, M) \mid s \in P\}$.

Often, $ML(T, \Gamma)$ will have several copies of some state s of T with different label sets M corresponding to different ways of reaching the same state with structurally different partial order executions, thus there is a substantial overhead in the $ML(T, \Gamma)$. However, it is a practical observation that there are typically few copies of the same state with very small sets[2] M. More importantly, very often the number of states s occurring with very small sets M in the accessible part of $ML(T, \Gamma)$ is small compared to $|S|$, which explains the success of McMillan's unfolding approach.

Combining the results of the previous section with Proposition 2, we obtain:

Theorem 2. *Let* $T = (S, \rightarrow, s_0)$ *be a* D *respecting transition system over* Σ, $ML(T, \Gamma)$ *a corresponding MLTTS, and* (Σ, D) *having character* (m, n). *Let furthermore* $P \subseteq S$ *be a property of parallel degree* $\leq n$. *Then* P *is reachable in* T *iff its interpretation* P' *is reachable in* $ML(T, \Gamma)$ *passing only by intermediate states* (s, M) *with* $|M| \leq \lfloor (n-1) log_n m + 1 \rfloor$

Theorem 2 thus yields an easy to realize partial order reduction scheme, for instance for reachability:

$B := \lfloor (n-1) \log_n m + 1 \rfloor$;
Explore:=$(s_0, \emptyset)$; *Visited*:=$\emptyset$;
repeat
  choose (s, M) from *Explore*;
  for each transition $s \stackrel{a}{\rightarrow} s'$
  do
    if $s' \models P$ **then** return(s')
    else $M' := M \setminus \{b \mid a\ D\ b\} \cup \{a\}$;
      if $|M'| \leq B$ and $(s', M') \notin$ *Explore*$\cup$*Visited*
      then add (s', M') to *Explore*
      fi
    fi
  od
  remove (s, M) from *Explore*, add (s, M) to *Visited*;
until *Explore*=$\emptyset$

Heuristic improvement: Local first search. Theorems 1 and 2 give only theoretical bounds for the worst case. To prove that a certain property is not reachable, the MLTTS has to be constructed up to this limit. However, if we are merely interested in quickly finding a state satisfying the property, we can deduce a heuristic from the bound: There exist paths with small label sets in the MLTTS

[2] The use of labels in a set Γ rather than Σ also reduces the number of label sets.

leading to local properties. The algorithm above can be specialized to the *Local first search* strategy as follows: *choose* $(s, M) \in$ *Explore with* $|M|$ *minimal*.
In practice, this strategy is realized in combination with a basic strategy such as depth first search (insertion to and extraction from set *Explore* as stack operations) or breadth first search (queue operations): For instance, a *local then depth first search strategy* is obtained by organizing *Explore* by priority stacks, one stack for each cardinality of label sets.

Further heuristic improvements of the algorithm are possible, for instance, we can exploit the fact that for a pair of states (s, M_1), (s, M_2) such that $M_1 \subseteq M_2$ in the MLTTS, any sequence started from (s, M_2) can be mimicked by a corresponding sequence starting from (s, M_1) preserving the label set inclusion. In other words: (s, M_2) need not be explored.

Beyond reachability. In principle, extensions of Proposition 2 and Theorem 2 are possible for a number of applications beyond reachability:

- Model checking for linear temporal properties without next-operator such that the set of visible actions has a small parallel degree can be done on the MLTTS with label sets satisfying the bound $\lfloor (n-1) log_n m + 2 \rfloor$ (c.f. [Pel93, Wal98]).
- Model checking of event based logics [Pen97,HNW98] can also be performed on the MLTTS with bound $\lfloor (n-1) log_n m + 2 \rfloor$ (which then represents the bound guaranteeing relative reachability among partial order executions with one maximal element).

Due to practical problems indicated in the next section, such extensions are currently of theoretical value only and will be explored in the future.

5 Experimentation

In order to practically evaluate the model checking approach shown in the previous section, we have built a first prototype implementation that allows to search local properties in the MLTTS. The prototype is currently implemented in Objective CAML using mostly functional data structures (with the exception of hash tables) and thus leaves great room for improved runtime efficiency. We used it to measure reductions in terms of numbers of states and did not compare execution times.

In particular, we wanted to explore whether synthetic benchmarks, on which McMillan's unfolding approach is known to work well, can be favorably handled by our method: We thus consider the dining philosophers and an asynchronous token buffer.
Both cases concern scalable examples with a parameter m (number of philosophers, number of buffer cells) which is at the same time the parallel degree of these systems and the sizes of their state spaces grow exponentially with m. Also in both cases, the distributed alphabet is 2-bounded.

Asynchronous buffer

cells (m)	1	2	3	4	5	6	7	8	12	15	32
states with $\|M\| = 1$	0	3	6	10	15	21	28	36	78	120	528
explored states $\|M\| \leq 2$	1	4	8	18	36	66	111	174	666	1371	14886
states without reduction	2	4	8	16	32	64	128	256	4096	32768	2^{32}
explored states $\|M\| \leq \log_2 m + 1$	1	4	8	19	52	132	310	1116	35335	241906	–

Lefthanded philosophers

philosophers (m)	1	2	3	4	5	6	7	8	12	16
states with $\|M\| = 1$	1	6	21	44	75	114	161	216	516	944
explored states $\|M\| \leq 2$	2	8	37	129	343	738	1363	2270	9758	25918
states without reduction	2	8	26	80	242	728	2186	6560	531440	$3^{16} - 1$
explored states $\|M\| \leq \log_2 m + 1$	2	8	37	202	1006	4195	13981	206421	–	–

Fig. 3. Experimental results

The findings for these two examples can be summarized as follows:

- The number of states with one label in the MLTTS is growing moderately ($O(m^2)$) and corresponds approximatively to the number of events in the McMillan unfolding.
- The number of states with two and three labels is still growing moderately and in both cases all states with one label can be found with bound 2 instead of $log_2 m + 1$ (a practical finding, which we theoretically confirmed). Hence, we have a reduction from exponential to cubic $O(m^3)$ for these synthetic benchmarks. And the local first search heuristic is justified by finding that the worst case bound is nowhere reached.
- With growing bound, the number of states in the MLTTS quickly explodes and soon surpasses the number of states in the original transition system, i.e. the overhead of label sets surpasses the reduction.
  The explanation we see is that the number of reachable states with a bounded number of labels quickly approaches the number of all states for the systems we explored, while the overhead of incomparable label sets is rising.

It should be noted that classical partial order reductions give even better results for the asynchronous token buffer (linear), but give much weaker reductions for the philosophers (exponential): This is due to the sensitivity of *confusion* of partial order reductions. The asynchronous token buffer is choice free, hence allows strong reduction, whereas for the philosophers, each operation of one philosopher could potentially lead to a blocking of another philosopher, so that typical heuristics for the computation of *ample sets* (see [Pel93]) basically disallows reductions for pickup-operations. This leads to an exponential growth in the set of explored states.

As an intermediate conclusion, local first search seems to be a very interesting search strategy for local properties. However, for proving the non-reachability of a property, which implies calculating the MLTTS up to the theoretical bound, the method without further improvements cannot be recommended due to the exploding overhead.

In the future, we will explore additional heuristics to combine with bound: Currently, the search picks sets of concurrent transitions up to the bound in a completely blind way and we believe that further structural insights into the relation of partial orders and the structure of the alphabet can help for further reduction.

6 Conclusions

We have introduced a new way of looking at partial order reductions, which is inspired by the McMillan prefix construction. Apart of mathematical elegance and simplicity, it has shown significant reductions for academic scalable examples and it holds a good promise for practical applications.

The method is currently based on a logarithmic bound on the number of maximal elements in partial order executions explored for verification. In a sense, the idea of the reduction is to focus the search on local progress rather than searching in all directions at once as is the case for the traditional interleaving approach. We believe that this focusing can still be strengthened: Currently, our method just uses the number of maximal elements as criterion based on a bound linked to the distributed alphabet of the system. For instance, we will explore whether further reductions can be achieved by taking the positions of the maximal elements in dependency graph of the alphabet into account.

A drawback of our method is the need to introduce additional overhead into the transition system first, before reductions can be applied. This can result in systems that are bigger than the interleaved transition systems. However, there are problem classes where it seems likely that the bookkeeping information does not lead to multiple instances of the same state. For instance, applying local first search to scheduling problems might add an interesting heuristic for reducing the search space without a blowup in the inverse direction: Schedules can be modeled as partial order executions and partial schedules as their prefixes. The idea would be to explore partial schedules with few maximal tasks in unfinished jobs first. On the other hand, the conflict structure of scheduling problems typically excludes any reduction using classical partial order methods. We intend to explore the potential of local first search in the scheduling context.

We believe that our approach also has a didactic benefit: It shows a link between the unfolding method and interleaving systems and it gives a measure explaining the enormous state space compression found in unfoldings.

Acknowledgements. We thank the following people for discussions on model checking applications of McMillan's Method, which have led to the direction we took in this work: Frank Wallner, Javier Esparza and Rom Langeraak. The first two authors wish to thank Volker Strehl (Univ. Erlangen) for an inspiring course on average case complexity, where they first learned about Strahler numbers, only to forget about them and to rediscover them at the center of their research a decade later.

References

[CG87] E.M. Clarke and O. Grumberg, *Avoiding the state explosion problem in temporal logic model checking algorithms*, Sixth Annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 294–303.

[DR95] V. Diekert and G. Rozenberg (eds.), *The book of traces*, World Scientific, 1995.

[EH00] J. Esparza and K. Heljanko, *A new unfolding approach to LTL model checking*, ICALP, LNCS, vol. 1835, 2000, pp. 475–486.

[ERV96] J. Esparza, S. Römer, and W. Vogler, *An improvement of McMillan's unfolding algorithm*, TACAS (T. Margaria and B. Steffen, eds.), LNCS, vol. 1055, 1996, pp. 87–106.

[Esp94] J. Esparza, *Model checking using net unfoldings*, Science of Computer Programming **23** (1994), 151–195.

[FRV79] P. Flajolet, J.C. Raoult, and J. Vuillemin, *The number of registers required for evaluating arithmetic expressions*, Theoretical Computer Science **9** (1979), 99–125.

[HNW98] M. Huhn, P. Niebert, and F. Wallner, *Verification on local states*, TACAS, LNCS, Springer-Verlag, 1998.

[McM92] K.L. McMillan, *Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits*, Computer Aided Verification (CAV), 1992, pp. 164–174.

[Pel93] D. Peled, *All from one, one for all: On model checking using representatives*, International Conference on Computer Aided Verification (CAV), LNCS, vol. 697, 1993, pp. 409–423.

[Pen97] W. Penczek, *Model checking for a subclass of event structures*, TACAS (Ed. Brinksma, ed.), LNCS, 1997.

[Str52] A.N. Strahler, *Hypsometric (area-altitude) analysis of erosonal topology*, Bull. Geol. Soc. of America **63** (1952), 1117–1142.

[Val89] A. Valmari, *Stubborn sets for reduced state space generation*, 10th International Conference on Application and Theory of Petri Nets, vol. 2, 1989, pp. 1–22.

[Wal98] F. Wallner, *Model checkin LTL using net unfoldings*, CAV, LNCS, vol. 1427, 1998, pp. 207–218.

Extending Memory Consistency of Finite Prefixes to Infinite Computations*

Marcelo Glusman and Shmuel Katz

Department of Computer Science
The Technion, Haifa, Israel
{marce, katz}@cs.technion.ac.il

Abstract. Infinite computations are widely used to model arbitrarily long computations of infinite-state systems. Certain properties have both a finitary version, applying only to finite prefixes of computations, and an infinitary version. It is tempting to verify these properties for finite computations only, and then conclude that the infinitary version of the property holds too. This generalization is sound for safety properties, but to verify non-safety properties "by prefixes", one must justify the generalization step. This paper studies how this can be done for sequential consistency of shared memory protocols. In the related literature, this generalization is sometimes done informally, if at all. We define, independently of any specific shared memory algorithm, sufficient conditions so that sequential consistency can be verified by finite prefixes. These conditions are expected to be satisfied by any reasonable shared memory system, regardless of the consistency model.

1 Introduction

Infinite computations are today a widely accepted formalism, which arises from the modelling of systems that do not necessarily terminate. Non-termination of reactive systems is not a fault, but rather a desired feature. Most real reactive systems are expected to continue running correctly for as long as needed, and to allow (constantly or at least frequently enough) the initiation of some sort of orderly shutdown procedure. If we don't concern ourselves with the shutdown behavior, then we may choose to consider infinite computations. Certainly, if a system can react correctly forever, it will also be able to do it for any finite period of time.

However, we know that no reactive system will actually run *forever* – it will eventually be stopped. Moreover, finite computations seem to be simpler objects to reason about than infinite computations. We may use induction to prove properties for any finitely sized computation. We may define well-founded measures based on the number of occurrences of certain events along a finite computation, and use them to prove equivalence to other - possibly more convenient [5] - finite computations. Infinite computations are the limits of converging sequences

* This work was partially supported by the Fund for the Support of Research at the Technion.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 411–425, 2001.
© Springer-Verlag Berlin Heidelberg 2001

of finite computations. Not all properties proved for all the elements of such a sequence will also hold for the limit. For example, if a given event occurs as the last event of every finite computation that is part of a converging sequence, in the limit this event might not occur at all (e.g.: the infinite sequence of finite words $\{a^i b\}, i \in \mathbb{N}$ converges to the infinite word a^ω).

We may then ask ourselves - why should we specify and prove properties about infinite computations at all? It should be enough to prove them for arbitrarily long finite computations. However, the conditions under which a property verified for finite computations can be extended to infinite computations are nevertheless of interest. First, the accepted abstract notion of liveness for reactive systems is defined for infinite computations – given a liveness property, *any* finite computation can be extended into an infinite one satisfying it[13]. If we only consider finite behaviors, then instead of specifying abstract liveness properties we need to specify bounds to the time it should take the system to react. Instead of requiring that something should *eventually* happen, we should say that it should happen *within a specified bound* if the system is allowed to run enough time. Without infinite computations, the specification becomes less abstract.

Another reason to consider infinite computations is related to the significance of proofs of non-safety properties, when the proofs are based on a property-preserving equivalence between computations [5]. Consider a finite computation in which an event creates a conflict with subsequent events. We may prove that this computation is equivalent to another finite computation in which the problematic event appears at the end, where it has no subsequent events to conflict with. Intuitively speaking, our proof would show how the system may "postpone" this event until the end, where it doesn't conflict with other events. Of course, this cannot be done in the infinite case, nor is it something that should be done: if the system is supposed to work properly, it must be able to do so without postponing anything until it is shut down. For the infinite computations to be proved correct based on our proof for finite prefixes, this proof should not rely on finiteness in such a way. The conditions we consider do not affect the soundness of a proof that "every finite computation is correct", but rather they change completely the relevance of such a result.

A good example is provided by shared memory consistency conditions like sequential consistency [11], where the equivalence to a particular computation (e.g., one displaying a serial memory's behavior) is the essence of the property itself. Requiring equivalence to a serial behavior for every finite prefix of a computation (where the local history for any processor is the same in any two equivalent behaviors) seems natural, but is not enough. As we will show, it is possible for a clearly *faulty* memory system to display an infinite behavior which is not equivalent to any serial memory behavior, while every one of its prefixes has an equivalent serial behavior. Therefore, in this paper we propose a set of conditions and we prove that they are sufficient to conclude sequential consistency for all infinite behaviors, if all their finite prefixes are proved sequentially consistent. Intuitively, these conditions should hold for every memory system

satisfying a minimal connectedness requirement, and with a reasonable level of independence between write/read operations done to different locations.

In Section 2 we discuss the level of abstraction appropriate for our purposes. In Section 3 we define verification by prefixes, prove formally that Sequential Consistency is not a safety property and describe our approach for verifying it by prefixes. Some general modelling assumptions are presented and justified in Section 4, and in Section 5 an abstract condition is proved sufficient for extending sequential consistency of finite prefixes to infinite behaviors. Section 6 shows how to prove that this abstract sufficient condition holds for a real system. Finally, in Section 7 we show three examples and in Section 8 we consider how to apply the same method if we relax one of our modelling assumptions (the write-before-read assumption discussed in Subsection 3).

Related Work: Sequential Consistency for the Lazy Caching algorithm has been verified directly for infinite computations in [6,8,10]. In works like [12,9,14, 7] only finite computations are considered, though [7] handles liveness by adding extensive notation involving the failures-divergence model for traces. In [1,3] and others, first finite computations are proved sequentially consistent under various simplifying assumptions, and then the result is extended to infinite computations. This generalization is done in varying degrees of detail, completeness and formalization, and in a way specific for the Lazy Caching algorithm. In our work, we propose and formally prove the conditions that justify such a generalization, independently of any specific implementation of a sequentially consistent shared memory.

2 Shared Memory Systems and Memory Consistency

2.1 Concrete and Abstract Models of a System's Interface

The observed events of a shared memory system are those occurring at its interface with the processors. This allows a specification of memory consistency to be independent of the details of a given algorithm. The model used to describe the Lazy Caching algorithm [1], for example, includes the description of a handshake between the processors and the memory, with two atomic events for every read or write operation, namely: a request (by a processor) and a response (by the memory). This is also the level of abstraction used in [2]. It allows us to specify "service" liveness properties, i.e., liveness of the processor/memory handshake: every request eventually gets a response (if not overwritten).

In a more abstract model of Lazy Caching and sequential consistency [4], the observed behaviors include only two kinds of events:
- $W_i(x, v)$: processor i writes value v into location x.
- $R_i(x, v)$: processor i gets value v from location x.

Requests and responses are not distinguishable, and any handshake that may prevent overwriting of requests is hidden. This level of abstraction (also used in [15]) is usually preferred for the specification of the consistency model, so we will adopt it here. However, in an implementation reads and writes might

not be atomic. A refinement mapping may be involved in the proof, connecting implementation events to abstract atomic reads and writes.

2.2 Abstract Definition of Sequential Consistency

We now define the sequential consistency property for abstract write/read behavior, independently of any specific shared memory system.

Let n be a fixed number of processors, $Addr$ any address space (a set of memory locations), and $Data$ any set of data values. Let $\Sigma_{n,Addr,Data} = \{R, W\} \times \{1..n\} \times Addr \times Data$ be the set of possible read/write interface events. (We use Σ for short, when the context is clear). The set $BEH = \Sigma^{\infty} = \Sigma^{+} \cup \Sigma^{\omega}$ includes all (nonempty) finite and infinite sequences of read and write interface events.

We assume that $v_0 \in Data$ is the initial value of all memory locations.

Definition. *A behavior $b \in BEH$ is* serial *iff every Read has the value written by the last Write (to the same memory location) preceding it, or v_0 if the location was not written to before the Read.*

This is also known as read/write consistency. Let $Ser = \{b : BEH | b \text{ is serial}\}$. Notation: We will use sets as predicates (e.g., $Ser(b) = b \in Ser$).

Definition. *Processor p_i's* local history *in behavior b (for $i \in 1..n$) is the projection of b to the set of events occurring at processor p_i's interface.*

$$b \restriction i = b \restriction (\{R, W\} \times \{i\} \times Addr \times Data)$$

Definition. *Two behaviors $a, b \in BEH$ are "*sc-equivalent*" iff for every processor p_i, p_i has the same local history in a as in b.*

$$a \equiv_{sc} b \leftrightarrow \forall i : 1..n \cdot a \restriction i = b \restriction i$$

Definition. *A behavior $b \in BEH$ is* sequentially consistent *iff b is sc-equivalent to some serial behavior.*

$$SC = \{b : BEH \mid \exists s \in BEH : b \equiv_{sc} s \wedge Ser(s)\}$$

This means that, from each processor's point of view, the actual memory system that generated behavior b could be a serial shared memory. Moreover, the serial behavior which is consistent with the given one is the same for all the processors. Weaker consistency conditions (like processor consistency) allow each processor to have its own view of the serial behavior, e.g., perceive a different write ordering.

The relation $\equiv_{sc}$ is an equivalence relation, and it preserves the SC property:

$$\forall a, b \in BEH : (a \equiv_{sc} b \wedge SC(a)) \rightarrow SC(b)$$

2.3 Concrete Shared Memory Systems and Sequential Consistency

Let $M_{n,Addr,Data}$ be a concrete shared memory system (M for short, where the context is clear). Let $Comps(M)$ denote the set of legal computations of M,

defined by a given algorithm/protocol. $Comps(M)$ is a prefix-closed set of finite and infinite sequences of events: both interface events (from Σ) and internal events. We choose to ignore internal events, so that our work can be applied to any implementation of shared memory. Let $Obs : Comps(M) \to BEH$ be the mapping that hides internal events:

$$\forall c \in Comps(M) : Obs(c) = c \restriction \Sigma$$

Definition. $Beh(M) = range(Obs)$ *is the set of* observable behaviors *of* M.

Definition. *(universal path quantifier* $\mathbf{A}$*) Let* Π *be a property (a predicate of possibly infinite behaviors).* $M \vDash \mathbf{A}\Pi \leftrightarrow \forall b \in Beh(M) : \Pi(b)$

Definition. *A shared memory system* M implements serial memory *iff all its behaviors are serial.* ($Beh(M) \subseteq Ser$, *or* $M \vDash \mathbf{A}Ser$)

Definition. *A shared memory system* M implements sequential consistency *iff all its behaviors are sequentially consistent.* ($Beh(M) \subseteq SC$, *or* $M \vDash \mathbf{A}SC$)

3 Verification by Prefixes and Sequential Consistency

Let ϕ be any predicate of finite behaviors (a *finitary* property).

Definition (universal prefix quantifier "a"[1]). *A (possibly infinite) behavior* b *satisfies* $\mathbf{a}\phi$ *iff every finite prefix of* b *satisfies* ϕ.[2]

Given a finitary property ϕ such that $\mathbf{A}(\mathbf{a}\phi \to \Pi)$ and any single behavior $b \in Beh(M)$, we can "verify $b \vDash \Pi$ by finite prefixes" by proving $b \vDash \mathbf{a}\phi$. If we do the same for all behaviors (i.e., prove $M \vDash \mathbf{A}\mathbf{a}\phi$), we "verify $M \vDash \mathbf{A}\Pi$ by finite prefixes". Moreover, if we find a condition that implies $\mathbf{A}\mathbf{a}\phi \to \mathbf{A}\Pi$ (which is weaker than $\mathbf{A}(\mathbf{a}\phi \to \Pi)$), then for any system satisfying this condition we can verify $\mathbf{A}\Pi$ by prefixes (by proving $\mathbf{A}\mathbf{a}\phi$), even when we didn't prove that $\mathbf{a}\phi \to \Pi$ holds for every individual behavior.

Definition. Π *is a* safety *property iff* $\Pi = \mathbf{a}\phi$ *for some finitary* ϕ. *[13]*

Obviously, a safety property $\Pi = \mathbf{a}\phi$ can be verified by prefixes, for single behaviors or for all of them.

Is SC a safety property? In [1,3] an example is shown suggesting that SC implies a liveness property. We now use the same example to formally prove the following lemma:

Lemma 1 *SC is not a safety property.*

[1] this is the same as the operator A, or A_f for finite behaviors, as defined in [13]

[2] For the *special case* that ϕ asserts that "at the end of the finite prefix the LTL *past formula* φ holds", the property $\mathbf{a}\phi$ is the same as $\Box\varphi$, the canonical representation of safety formulas in LTL.

Proof: Consider the following infinite behavior b: $(v \neq v_0)$

$$W_i(x,v)R_j(x,v_0)R_j(x,v_0)R_j(x,v_0)R_j(x,v_0)\cdots$$

That is, a Write of v to x by processor p_i is followed by infinitely many Reads of the initial value v_0 from x by processor p_j. Behavior b is not sequentially consistent. No matter how we reorder its events, there always remain infinitely many reads that occur after the write but get the old value of x. Now consider the finite prefixes of b. A prefix b_k of b with length k has the form:

$$W_i(x,v)R_j(x,v_0)\cdots R_j(x,v_0) \qquad (k-1 \text{ reads})$$

For all k, b_k can be extended into an infinite behavior:

$$W_i(x,v)R_j(x,v_0)\cdots R_j(x,v_0)R_j(x,v)R_j(x,v)\cdots$$

which is SC since it is sc-equivalent to the infinite *serial* behavior:

$$R_j(x,v_0)\cdots R_j(x,v_0)W_i(x,v)R_j(x,v)R_j(x,v)\cdots$$

Assume now that a finitary property ϕ such that $SC = \mathbf{a}\phi$ exists. Every b_k is a prefix of some SC behavior, so every b_k satisfies ϕ. Therefore, b satisfies $\mathbf{a}\phi$, a contradiction to the fact that it is not SC. □

Can non-safety properties be verified by prefixes? If we identified a finitary property ϕ and a property Δ such that $SC = \Delta \wedge \mathbf{a}\phi$, then we could split the proof of $\mathbf{A}SC$ into two proofs: $\mathbf{A}\Delta$ and $\mathbf{A}\phi$. Of course, to prove that $SC = \Delta \wedge \mathbf{a}\phi$ we have to show that Δ doesn't hold in *all* the possible "bad" scenarios (i.e., those satisfying $\mathbf{a}\phi$ but not SC), which should then be precisely characterized. In fact, there are other behaviors besides the one in the proof of Lemma 1 that are problematic: (assume all v_i are different)

$$W_i(x,v_1)R_j(x,v_0)W_i(x,v_2)R_j(x,v_0)W_i(x,v_3)R_j(x,v_0)\cdots$$

A property like $\Delta =$"every written value, if not overwritten, is eventually seen by all the processors" would rule out the first bad scenario, but not the second one. In this paper we do not follow this approach.

In [1] it is claimed that sequential consistency is a safety property, and the Lazy Caching algorithm is then verified by proving that all its finite computations are SC. Nevertheless, their conclusion that all infinite computations of the Lazy Caching algorithm are also SC is actually justified based on extra liveness assumptions. Their justification is not complete – they show why the first problematic scenario shown above is not possible in the algorithm, but they don't prove that this is the *only* "bad" case.

In this paper we follow the alternative approach for verification of $\mathbf{A}SC$ that was mentioned before: Instead of a Ψ and a ϕ such that $\mathbf{A}\Psi \rightarrow \mathbf{A}(\mathbf{a}\phi \rightarrow SC)$, we will find a Ψ and ϕ such that $\mathbf{A}\Psi \rightarrow (\mathbf{A}\mathbf{a}\phi \rightarrow \mathbf{A}SC)$.

A natural choice for ϕ, which corresponds to the intuitive notion of verifying sequential consistency by prefixes, is to take SC itself, applied to finite prefixes. For clarity, let SC denote the finitary version of SC (both have the same definition, but SC only applies to finite behaviors). As illustrated by our previous examples, $\mathbf{a}$SC does not imply SC: those "bad" behaviors can be displayed by an incorrect (e.g., completely disconnected) memory system, yet they still satisfy $\mathbf{a}$SC. On the other hand, we will also show (in Section 3) that for certain

SC behaviors **a**SC doesn't hold: there are sequentially consistent behaviors with prefixes which are not SC.

4 Some Modelling Assumptions

In this section we make (and justify) some assumptions about the system being verified, which should be natural for reasonable shared memory systems. These assumptions are common in the literature on shared memory systems.

Abstracting out from a specific address space: The algorithmic idea behind shared memory protocols is usually not sensitive to the actual size of the address space. We will consider verification of shared memory protocols that are parameterized on the address space. No assumptions are made on this set of memory locations, so the proof will be correct for any concrete address space.

Abstracting out from data value constraints: As with the set of memory locations, the idea behind concrete shared memory designs (for read/write objects) is usually not sensitive to the actual size of a memory location. This should enable us to verify a version of the design in which the set of possible data values is infinite.[3]

Abstracting out from repeated written values: Shared memory protocols are usually *data independent* – the way data items are "shuffled around" does not depend on their specific values. If we consistently rename the data values so they become unique, the result is as sequentially consistent as the original behavior. Knowing that written values are unique makes it easier to relate a read event with its corresponding write.

Definition. *A computation is called* unambiguous *iff for every memory location, all the written values are different from* v_0 *and different among themselves.*

Note that if the data domain were finite, its size would bound the number of possible writes with different values to a single location. The previous abstraction allows us to make the following *unique writes* assumption:

For every behavior $b \in Beh(M)$ there is an unambiguous behavior $b' \in Beh(M)$ such that $SC(b') \rightarrow SC(b)$.

Thus it is sufficient to verify $\mathbf{A}SC$ only for unambiguous behaviors.

Value-Closed and Write-Before-Read Behaviors:

Definition. *A (finite or infinite) behavior* $b \in Beh(M)$ *is called* value-closed *(VC) iff every value read by a Read event in* b *is either the initial value (*v_0*), or a value written by* some *Write event in* b*.*

[3] A proof based on the assumption that the data domain is infinite cannot be "instantiated" for an implementation with a finite data domain. However, any legal behavior of an implementation with bounded data is also exhibited by the infinite (unbounded data) version. If we prove $\mathbf{A}SC$ for the infinite version, we can conclude the same for the finite one, by inclusion. This reasoning cannot be used to prove the *existence* of certain behaviors in the finite version.

Definition. *A (finite or infinite) behavior $b \in Beh(M)$ is called* write-before-read *(WBR) iff every value read by a Read event in b is either v_0, or a value written by a* previous *Write event in b.*

Lemma 2 *The following implications hold for all finite or infinite behaviors:*

$$\begin{array}{ccccccc} & & Ser & \leftrightarrow & \mathbf{a}Ser & \rightarrow & \mathbf{a}\text{SC} \\ & \swarrow & \downarrow & & & & \downarrow \\ SC & & WBR & \leftrightarrow & \mathbf{a}\,WBR & \leftrightarrow & \mathbf{a}\,VC \\ & \searrow & \downarrow & & & & \\ & & VC & & & & \end{array}$$

Proof: We only prove $\mathbf{a}\,VC \rightarrow \mathbf{a}\,WBR$, the rest is immediate from the definitions. Assume b doesn't satisfy $\mathbf{a}\,WBR$: some prefix of b has a Read event not preceded by its corresponding Write. The shortest such prefix is not VC, therefore b doesn't satisfy $\mathbf{a}\,VC$. □

For memory systems with atomic Write operations, it is easy to see (if they don't invent values or foresee the future) that every behavior satisfies the *WBR* property. For example: the behavior (for $v \neq v_0$): "$R_j(x,v)W_i(x,v)$" is not expected. However, when Writes are *not* implemented as atomic operations, a newly written value may be returned to a Read operation before the writing processor gets the Write's response. The implementor might choose a refinement mapping in which the concrete response event for the Read operation is mapped to the abstract $R_j(x,v)$ event, and the concrete response event for the Write operation is mapped to the abstract $W_i(x,v)$ event. Such an implementation could exhibit the non-*WBR* behavior described above. Note that *WBR* is not a necessary condition for sequential consistency: the events may still be rearranged into a serial sequence.

Lemma 2 implies that non-*WBR* behaviors never satisfy **a**SC. Even though **Aa**SC $\rightarrow$ **A***SC* holds (vacuously) for non-*WBR* systems, we cannot use this fact to verify their sequential consistency by prefixes. Therefore, we now temporarily restrict ourselves to *WBR* systems. In Section 8 we generalize the results to non-*WBR* systems.

5 An Abstract Sufficient Condition for Aasc $\rightarrow$ A*SC*

For "bad scenarios" like those seen in Section 3, we can prove that any finite prefix of a non-*SC* behavior is SC by "postponing" an event (and those events that follow it locally) until after all the conflicting events from other processors occur. At the end of this section we define a condition that will not be satisfied by such behaviors.

Definition (*FP* behaviors). *Behavior b is a* finite-postponement *(FP) behavior iff for every event e in b, there is a natural number k_e such that for every $c \equiv_{sc} b$ with a serial prefix c_{k_e} of length at least k_e, e appears in c_{k_e}.*

In other words, for every event e in b, the set of events that may precede e in some sc-equivalent behavior's serial prefix is finite.

Theorem 1 *For every FP behavior,* $\mathbf{a}\text{SC} \rightarrow SC$.

Proof: Let b be an *FP* behavior satisfying $\mathbf{a}$SC. For every natural number k, the prefix b_k of b of length k has at least one serial sc-equivalent finite behavior s_k (of the same length k). Let $sb(k)$ denote the behavior built by replacing b_k by s_k in b. For every k, $sb(k)$ is sc-equivalent to b, and has a serial prefix of length k. For $S_b = \{sb(k), k \in \mathbb{N}\}$, build a tree with the behaviors in S_b, where shared prefixes appear as a shared path in the tree. The branching degree of this tree is finite, since at each node (which represents a finite prefix of a behavior), only the next event in the local histories of finitely many processors can be added to any behavior (because all the behaviors in S_b are sc-equivalent). We prune this tree leaving only the serial parts of the behaviors: branches are cut at all nodes where there is a violation of read/write consistency. Since for every k the tree has a behavior which is serial up to k, the remaining tree is still infinite. By Koenig's Tree Lemma, it has an infinite path - let's call it s. The path s is serial since the pruned tree doesn't include any violation of read/write consistency, and for every k, there is a behavior in S_b that shares with s a prefix of length k. To conclude $SC(b)$, we must prove that $s \equiv_{sc} b$. First we prove they have the same events. Clearly, every event in s is also an event of some behavior in S_b, so it also appears in b. Let e be an event in b. Since b is *FP*, there is a natural number k_e such that for every behavior sc-equivalent to b with a serial prefix of length at least k_e, e appears in that prefix. As seen above, there is a behavior s_e in S_b that shares with s a prefix of length k_e. The event e must appear in that serial prefix, so e appears in s. We only have left to prove that the local histories of s and b are the same. Let e_1 and e_2 be any two events occurring at the same processor's interface. If e_1 precedes e_2 in s, then e_1 precedes e_2 in some behavior in S_b, which is sc-equivalent to b. Therefore, the same ordering occurs in b. □

If every behavior of a system were *FP*(i.e., if $\mathbf{A}FP$), then this theorem means we could prove $\mathbf{A}SC$ by just verifying $\mathbf{Aa}$SC. However, for most interesting systems this is not the case. Non-*FP* behaviors arise, for example, if processors execute completely independent programs, without sharing any memory location.

We now extend the result of Theorem 1 to systems in which every behavior b has an *FP representative*: another behavior containing the local histories of b, and additional Write/Read events that make it *FP*. For the *FP* representative not to disturb the events from b, new (i.e., unused in b) memory locations will be needed. This is not a problem, since we are verifying systems $M(Addr)$ that are parametric on the set of memory locations $Addr$.

Definition. *For all b in* $Beh(M(Addr))$*:*

$$\Psi(b) = \exists Addr' \exists b' \in Beh(M(Addr')) : FP(b') \wedge (SC(b') \rightarrow SC(b))$$

$\mathbf{A}\Psi$ is a sufficient condition for verification of sequential consistency by prefixes:

Theorem 2 $\mathbf{A}\Psi \rightarrow (\mathbf{Aa}\text{SC} \rightarrow \mathbf{A}SC)$

Proof: Assume $M(Addr) \vDash \mathbf{A}\Psi$. Let $b \in Beh(M(Addr))$. We have $\Psi(b)$: there exists an $Addr$' and FP behavior b' in $Beh(M(Addr'))$ such that $SC(b') \rightarrow SC(b)$. Assume $M(Addr) \vDash \mathbf{Aa}\text{SC}$. Since M is parametric on $Addr$, no assumptions were made about $Addr$ when proving $\mathbf{Aa}\text{SC}$, so it also holds for $M(Addr')$. By Theorem 1, $SC(b')$ holds, which in turn implies $SC(b)$. Since b was arbitrary, $M(Addr)$ satisfies $\mathbf{A}SC$. □

6 Concrete Conditions Implying $\mathbf{A}\Psi$

The condition $\mathbf{A}\Psi$ is too abstract; we now formulate more concrete conditions which imply $\mathbf{A}\Psi$. It is expected that the new conditions are easy to verify for any reasonable shared memory system, which will therefore satisfy $\mathbf{Aa}\text{SC} \rightarrow \mathbf{A}SC$.

Liveness condition Γ (Eventual Influence): For every two processors p_i, p_j, memory location x and data value v_1, if p_j reads from location x repeatedly, and p_i repeatedly writes to x values different from v_1, then it is not the case that p_j always gets the same value v_1. In LTL:

$$\Gamma = \forall i, j \in 1..n, x \in Addr, v_1 \in Data : \forall_f v \in Data :$$
$$\Box[\Box\Diamond W_i(x,v) \wedge \Box(W_i(x,v) \rightarrow v \neq v_1) \wedge \Box\Diamond R_j(x,v) \rightarrow \Diamond(R_j(x,v) \wedge v \neq v_1)]$$

Here, $\forall_f$ is a temporal (flexible) quantifier – v can be different at each step. This "eventual influence" liveness property is a natural assumption for any reasonable shared memory system, regardless of the memory consistency model it is supposed to implement. It provides for minimal (even not reliable) connectivity among the processors, through shared memory locations.

Definition (should-precede). *Event* e_1 should-precede *event* e_2 *in behavior* b *iff for every behavior* d *sc-equivalent to* b*, if* e_2 *appears in a serial prefix of* d *then* e_1 *precedes* e_2 *in* d*.*

Lemma 3 *If* e_1 *precedes* e_2 *in* b *and both occur at the same processor's interface, then* e_1 *should-precede* e_2 *in* b*.*

Proof: Trivial, since sc-equivalence preserves local histories. □

Lemma 4 *For a given behavior* b*, the relation* should-precede *in* b *is transitive.*

The proof is left to the reader.

Definition. *We say that* $p_i \rightrightarrows_b p_j$ *iff in behavior* b*, processor* p_i *has infinitely many events that should-precede infinitely many respective events in* p_j*.*

Lemma 5 *For every behavior $b \vDash \Gamma$ and processors p_i, p_j: if p_i writes infinitely many unique values to some location x and p_j reads infinitely often from x, then $p_i \rightrightarrows_b p_j$.*

Proof: From the definition of Γ, p_j must read infinitely many unique values that were written by p_i, so there are infinitely many $W_i(x,v), R_j(x,v)$ pairs. For each such pair, $W_i(x,v)$ should-precede $R_j(x,v)$ in b. Therefore, $p_i \rightrightarrows_b p_j$. □

Definition. *A behavior $b \in Beh(M)$ is a* heartbeat *(HB) behavior iff for every $i, j \in 1..n$, processor p_i writes infinitely many unique values to location hb_i, and processor p_j performs infinitely many reads from location hb_i.*

Lemma 6 *For every HB behavior b, if $b \vDash \Gamma$, then for all i, j :$p_i \rightrightarrows_b p_j$*

Proof: In b, every p_i writes infinitely many unique values to location hb_i from which every p_j reads infinitely often. By Lemma 5, $p_i \rightrightarrows_b p_j$. [4] □

Theorem 3 *If b is an HB behavior and $b \vDash \Gamma$ then b is FP.*

Proof: Let e_i be an event occurring at processor p_i in b, and p_j some other processor. By Lemma 6, $p_i \rightrightarrows_b p_j$. By the definition of $\rightrightarrows_b$, for some event e_i' of p_i that occurs after e_i, e_i' should-precede some event e_j in p_j. By Lemmas 3 and 4, e_i should-precede e_j and also any later events at p_j. Therefore, only finitely many events at p_j (all of which precede e_j) may precede e_i in any serial prefix of an sc-equivalent behavior. □

$$\begin{array}{lccccc} p_i: & e_i & \to & e_i' & \to \cdots & \\ & & & & \searrow & \\ p_j: & \cdots & \to \cdots & \to & e_j & \to \cdots \end{array}$$

Let us add n new memory locations to the original address space. For any given $Addr$, let $Addr^{hb} = Addr \cup \{hb_i | i : i..n\}$. For any given $M = M(Addr)$, let $M^{hb} = M(Addr^{hb})$. Obviously, every behavior b in $Beh(M)$ also belongs to $Beh(M^{hb})$.

Definition (*HB* Assumption). *For every behavior $b \in Beh(M)$ there is an HB behavior $b' \in Beh(M^{hb})$ such that $b' \restriction Addr \equiv_{sc} b$.*

In other words, it should be possible to add heartbeat operations to every processor (using the new locations hb_i) without modifying the original behavior's local histories.

Theorem 4 *If a shared memory protocol M satisfies all the modelling assumptions from Section 4, the HB assumption and $\mathbf{A}\Gamma$, then M satisfies $\mathbf{A}\Psi$.*

[4] Here we proved *directly* that for *HB* behaviors satisfying Γ, the directed graph $(\{p_i : i \in 1..n\}, \rightrightarrows_b)$ is a clique. The relation $\rightrightarrows_b$ is transitive, so this lemma will also hold for any other definition of *HB* describing a strongly connected graph.

Proof: Let M^{hb} be defined as before. The *HB* assumption implies that for every behavior $b \in Beh(M)$ there is an *HB* behavior $b' \in Beh(M^{hb})$ such that $b' \restriction Addr \equiv_{sc} b$. The $\mathbf{A}\Gamma$ assumption that was made about M also holds for M^{hb}, since we didn't assume anything about the address space. By Theorem 3, b' is *FP*. Assume now that b' is *SC* : there is a serial behavior $s' \equiv_{sc} b'$. The projection of a serial behavior to part of the memory locations is also a serial behavior, so $s' \restriction Addr$ is serial. Clearly, $s' \restriction Addr \equiv_{sc} b' \restriction Addr \equiv_{sc} b$, so b is *SC* too. We showed that for every $b \in Beh(M(Addr))$ there is an $Addr$' and an *FP* behavior b' in $M(Addr')$ such that $SC(b') \rightarrow SC(b)$, i.e., $M(Addr) \models \mathbf{A}\Psi$. □

7 Examples

7.1 Lazy Caching

The Lazy Caching algorithm is described in [1,4]. We will refer to the version in [4] (summarized in Appendix A), since it has atomic Read and Write events, and show why it satisfies all the conditions described above. First of all, its description is parametric on the address space and the data domain. Thus, we can assume an infinite data space and consider only its unambiguous behaviors. It is easy to see that the algorithm has only *WBR* behaviors: Read data values come from cached values, which originate from *earlier* Write events and propagate through Out and In queues.

Let's now see why the property $\mathbf{A}\Gamma$ (eventual influence) holds. Assume processor p_i writes unique values infinitely often to location x, and processor p_j reads from x infinitely often. The Out and In queues are reliable and the Memory Write (MW) and Cache Update (CU) actions, which remove items from those queues, enjoy a fairness assumption. If the value v is written into p_i's Out queue, it will eventually be propagated to the main memory and p_j's cache. The location x in the cache might be invalidated (CI) before p_j reads v from x, but we know that p_j succeeds to read from x infinitely often, therefore eventually it will read some value: v if the cache was updated from memory, or a newer value if the cache was updated by a new Memory Write action. In any case, p_j will stop reading the value that was in x before v was written.

Now we prove that every behavior of the Lazy Caching algorithm can be extended to an *HB* behavior if n new locations (hb_i) are added. The queues and caches are not bounded in size. The addition of heartbeat Write events at some processor, however, may affect the enabledness of Read events (if its Out queue is empty and its In queue has no "starred" entries). Read events will be postponed until the new heartbeat value reaches the processor's cache. By the time this happens, the cache might have been updated by values written before the added heartbeat, so a delayed Read operation might return a different value. Let's now see how this can be avoided. Recall that this is not a modification of the algorithm: we must only show that a suitable *HB* behavior exists.

Let b be some behavior, and let p_i be a processor that writes infinitely often in b. We can insert a $W_i(hb_i, w)$ event just after every $W_i(x, v)$ in the original

behavior (with a new value w each time), and let the (hb_i, w) pair "follow" the (x, v) pair along the data path: immediately after a $MW_i(x, v)$ there will be a $MW_i(hb_i, w)$, and so on. We also insert a $R_j(hb_i, z)$ event at every processor p_j as soon as any $CU_j(hb_i, z)$ event occurs. No Read event at p_i will be delayed by the presence of the (hb_i, w) item in p_i's Out queue, (or later by a $(hb_i, w, *)$ item in p_i's In queue), since it "follows" a previous (x, v) item (or $(x, v, *)$, respectively). The other processors' local histories (projected to original events) are also not affected. Let's now consider a processor p_i that does infinitely many Reads but only finitely many Writes in b. Eventually, its Out queue will be empty and its In queue will not contain "starred" items. After that, cache updates and Reads at this processor are independent from other events, so they can be done earlier –while preserving the ordering among them– as soon as a new item is added to the In queue. This will not modify the local history of this processor. The moved Reads will get the same values that appear in main memory. The addition of heartbeat writes at this processor may cause some delay to a Read event, but it will not affect its value.

We conclude that Lazy Caching satisfies the conditions that justify $\mathbf{Aa}\text{SC} \rightarrow \mathbf{A}SC$. In [1,12,9] it was verified that it satisfies $\mathbf{Aa}\text{SC}$, so their results can be extended to $\mathbf{A}SC$.

7.2 Weak Lazy Caching

Let's now consider a modified version of Lazy Caching: the Read operation's enabledness will only depend on the cache contents, and not on the contents of the Out or In queues. A processor with a nonempty Out queue or with "starred" items in its In queue is now allowed to Read values stored in the cache.

The Weak Lazy Caching algorithm satisfies all the conditions to justify $\mathbf{Aa}\text{SC} \rightarrow \mathbf{A}SC$. This is easier to check here than in the original Lazy Caching algorithm, since writes can be added to hb_i locations at any moment, without ever affecting any event from the original behavior. However, this weak algorithm is not a correct implementation of sequential consistency! This shows that the family of systems satisfying the needed conditions is not restricted to sequentially consistent algorithms. To belong to this family, an algorithm must be "connected" (to satisfy the "eventual influence" liveness property $\mathbf{A}\Gamma$) and allow enough independence between the operations done at different locations (to satisfy the HB assumption).

Fortunately, for the weak algorithm we simply cannot prove $\mathbf{Aa}\text{SC}$. This version of the algorithm allows a behavior b where processor p_i writes a value to location x and later reads an older value from x. Assume p_i is the only processor writing to x. This local violation of write/read consistency will appear in any sc-equivalent behavior, so a prefix of b containing this violation is not SC.

7.3 Really-Lazy Caching

We now modify the original Lazy Caching by replacing the fairness of the Memory Write operations with the following mechanism: the first item in an Out

queue is dequeued (by a MW) only when a new Write is done. This means that if there are only finitely many Writes in a behavior, then the last Write can remain in the Out queue forever.

This algorithm can in principle display the behavior shown in the proof of Lemma 1. We could prove **a**SC for any finite behavior, by postponing the last Write until after the last conflicting Read. On the other hand, this is not a correct implementation – $\mathbf{A}SC$ doesn't hold!

Again, fortunately, any attempt to superimpose a heartbeat protocol on such a behavior would cause the last written value to be taken from the Out queue when the next heartbeat is written. This means that the values read by the original behavior will be affected by the addition of heartbeats. In this example, the *HB* assumption does not hold.

8 Extension to Systems with Non-*WBR* Behaviors

Non-*VC* behaviors cannot be *SC* (see Lemma 2). Therefore, we assume that all infinite behaviors are value-closed (*VC*). Every reasonable memory system lets Read events return only values that are written at some Write event.

Definition (Condition EQ-wbr). *For every infinite VC behavior* $b \in Beh(M)$ *and for every* $k \in \mathbb{N}$*,* b *must be sc-equivalent to some behavior* $d^k \in Beh(M)$ *with a* WBR *prefix of length* k*.*

This condition can be verified inductively (for any behavior b of a given system M) by taking the first $R_i(x, v)$ that precedes its matching $W_j(x, v)$ and showing that the corresponding Write could have occurred before the Read in some sc-equivalent behavior, while keeping the ordering of the reads – thus creating a longer WBR prefix.

Theorem 5 *Let* $\text{SC}_w = (\text{WBR} \rightarrow \text{SC})$*. For any VC system* M *satisfying the EQ-*WBR *condition and* $\mathbf{A}\Psi$*,* $\mathbf{Aa}\text{SC}_w \rightarrow \mathbf{A}SC$*.*

Proof: Assume $\mathbf{Aa}\text{SC}_w$. By $\mathbf{A}\Psi$, it is enough to prove that *FP* behaviors are *SC*. Let $b \in Beh(M)$ be *FP*. By EQ-WBR, for every $k \in \mathbb{N}$ there is a behavior $d^k \in Beh(M)$ sc-equivalent to b with a WBR prefix of length k. This prefix satisfies SC_w, so it is SC. Therefore, the *FP* behavior b is sc-equivalent to behaviors with serial prefixes of all lengths. The rest of the proof is as for Theorem 1. □

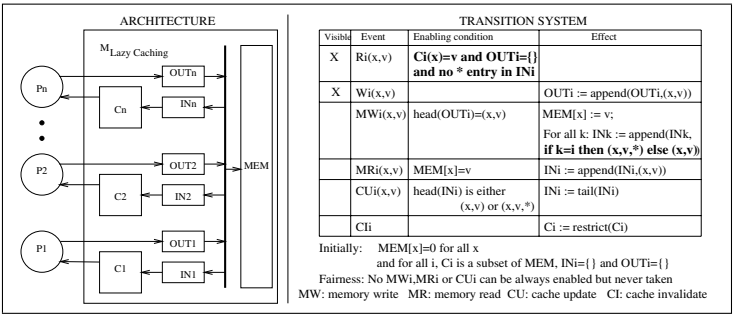
The proof of Theorem 3 still holds: if behavior b is not *WBR*, heartbeat Writes will still precede their Reads in a *serial* prefix of any behavior sc-equivalent to b. The proofs of Theorems 2 and 4 also do not rely on the *WBR* assumption.

Acknowledgement: We thank Shaz Qadeer for the useful comments about the modelling assumptions used in [14], some of which we adopted here.

References

1. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
2. Hagit Attiya and Jennifer Welch. *Distributed Computing*. McGraw-Hill Publishing Company, UK., 1998.
3. Ed Brinksma. Cache consistency by design. *Distributed Computing*, 12:61–74, 1999.
4. Rob Gerth. Sequential consistency and the lazy caching algorithm. *Distributed Computing*, 12:57–59, 1999.
5. Marcelo Glusman and Shmuel Katz. Mechanizing proofs of computation equivalence. In *Proceedings of 11th International Conference on Computer-Aided Verification, CAV'99*, volume 1633 of *LNCS*, pages 354–367. Springer-Verlag, 1999.
6. Susanne Graf. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing*, 12:75–90, 1999.
7. Wil Janssen, Mannes Poel, and Job Zwiers. The compositional approach to sequential consistency and lazy caching. *Distributed Computing*, 12:105–127, 1999.
8. Bengt Jonsson, Amir Pnueli, and Camilla Rump. Proving refinement using transduction. *Distributed Computing*, 12:129–149, 1999.
9. Shmuel Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, pages 59–78. American Mathematical Society, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29.
10. Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. *Distributed Computing*, 12:151–174, 1999.
11. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
12. Gavin Lowe and Jim Davies. Using CSP to verify sequential consistency. *Distributed Computing*, 12:91–103, 1999.
13. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 377–408, New York, NY, 1990. ACM Press.
14. Shaz Qadeer. On the verification of memory models of shared-memory multiprocessors. In *Workshop on Shared Memory Protocol Verification*, October 2000.
15. Andrew Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.

APPENDIX A: The Lazy Caching algorithm (adapted from [4])



TRANSITION SYSTEM

Visible	Event	Enabling condition	Effect
X	Ri(x,v)	**Ci(x)=v and OUTi={} and no * entry in INi**	
X	Wi(x,v)		OUTi := append(OUTi,(x,v))
	MWi(x,v)	head(OUTi)=(x,v)	MEM[x] := v; For all k: INk := append(INk, **if k=i then (x,v,*) else (x,v)**)
	MRi(x,v)	MEM[x]=v	INi := append(INi,(x,v))
	CUi(x,v)	head(INi) is either (x,v) or (x,v,*)	INi := tail(INi)
	CIi		Ci := restrict(Ci)

Initially: MEM[x]=0 for all x
and for all i, Ci is a subset of MEM, INi={} and OUTi={}
Fairness: No MWi,MRi or CUi can be always enabled but never taken
MW: memory write MR: memory read CU: cache update CI: cache invalidate

Abstraction-Based Model Checking Using Modal Transition Systems

Patrice Godefroid[1], Michael Huth[2], and Radha Jagadeesan*[3]

[1] Bell Laboratories, Lucent Technologies, god@bell-labs.com
[2] Computing and Information Sciences, Kansas State University, huth@cis.ksu.edu
[3] Department of Computer Science, Loyola University of Chicago, radha@cs.luc.edu

Abstract. We present a framework for automatic program abstraction that can be used for model checking any formula of the modal mu-calculus. Unlike traditional *conservative* abstractions which can only prove universal properties, our framework can both prove and disprove any formula including arbitrarily nested path quantifiers. We discuss algorithms for automatically generating an abstract *Modal Transition System* (MTS) by adapting existing predicate and cartesian abstraction techniques. We show that model checking arbitrary formulas using abstract MTSs can be done at the same computational cost as model checking universal formulas using conservative abstractions.

1 Introduction

There are essentially two approaches for extending the applicability of model checking to programs written in general-purpose programming languages such as C or Java. The first approach consists of adapting existing model-checking techniques into a form of systematic testing that is applicable to processes executing arbitrary code (e.g., [16]); although sound, this approach is inherently incomplete for large systems. The second approach consists of automatically extracting a model out of a program by a static analysis of its code, and of analyzing this model using existing model-checking techniques (e.g., [1,9]); although automatic abstraction can be complete, this approach is generally unsound since abstraction usually introduces unrealistic behaviors that may yield to spurious errors being reported when analyzing the model.

In this paper, we study the latter approach and show how automatic abstraction can be performed in such a way that it yields verification results whose completeness and soundness can be both guaranteed. We also show how automatic abstraction can be applied to check arbitrary formulas of the modal mu-calculus [22], thus including negation and arbitrarily nested path quantifiers. Maybe surprisingly, both extensions can be implemented in combination with existing abstraction techniques without incurring any significant computational overhead. Our algorithms could be used to extend the scope of existing tools for (conservative) automatic abstraction such as SLAM [1] and Bandera [9],

* Supported by NSF CCR-9901071.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 426–440, 2001.
© Springer-Verlag Berlin Heidelberg 2001

which currently support the verification of universal properties only [7]. Our algorithms to construct abstract transition systems can also be used in the context of the verification of arbitrary modal mu-calculus formulas with methods based on theorem-proving [30].

Allowing the specification of arbitrary formulas with nested path quantifiers makes it possible to express more elaborate properties of the temporal behavior of a reactive program, such as "for all possible input values, there exists an execution path of the system that allows the user to restart the service". Unfortunately, the verification of such properties necessitates the relation between the concrete program and an abstract program to be more constraining than a simulation relation [26,25]. Although bisimulation [28,27] over Labeled Transition Systems (LTSs) reflects all such general properties [18], it is persuasively argued in [24,23] to be ill-suited for our context: as an equivalence relation, it confines the choice of an abstraction to the implementation's equivalence class, which is too limiting to allow for compact abstractions.

For this reason, we use Modal Transition Systems (MTS) [24,23] for representing abstract systems in order to allow their specifications to be partially defined. MTSs are LTSs with two kinds of transitions, termed `may` and `must` transitions, satisfying the consistency condition that every `must`-transition is also a `may`-transition. A MTS can be "refined" by preserving at least all `must`-transitions (and maybe adding some) while eliminating some `may`-transitions. Since this refinement preorder on MTSs preserves all properties expressible in the modal mu-calculus, we can verify any such properties on the source (concrete) program by verifying these on any abstract MTS that is refined by this concrete program; conversely, if there exists a behavior of the abstract MTS that refutes the property, the existence of a refuting behavior of the concrete program is also immediately guaranteed.

An alternative representation for abstract systems is the *partial Kripke structure* [4,5]. Partial Kripke structures are Kripke structures whose states are labeled with atomic propositions that can have any of three possible truth values: `true`, `false` or `unknown`. Partial Kripke structures are closely related to MTSs since the transition relation of a MTS can be viewed as a function associating each transition with one of three possible values: `must`-transitions correspond to the value `true`, `may`-transitions that are not `must`-transition are mapped to `unknown`, and absent transitions render `false`. It can be shown that any partial Kripke structure can be translated into an equivalent MTS, and vice versa. This correspondence makes it possible to apply the results of [4,5] (in particular, model-checking algorithms and complexity bounds) to the context of MTSs. Conversely, the abstraction techniques developed in this paper can be adapted to the context of partial Kripke structures.

A crucial aspect of our model-checking framework is that it not only permits the abstraction of complete programs, but also the refinement of *partially* specified abstract programs by more concrete abstract programs, to adequately accommodate the incremental process of building more detailed abstractions by successive approximations, as used in SLAM or Bandera for instance.

We develop an expressive and flexible relational calculus for the sound specification of MTSs as abstractions. This calculus adapts the definitions of [12,13] to *partially* specified systems and is complete in the sense that it can specify every refinement of MTSs. In particular, any abstract interpretation of data values extends to a relational abstraction expressible in the calculus. In this calculus, we specify two standard abstractions of abstract interpretation [10], namely predicate abstraction [17,14,32] and cartesian abstraction [3,1] (also known as "independent attribute analysis"), and describe their implementations:

- When applied to `may`-transition relations only, the specifications and implementations we present coincide with traditional "conservative" abstraction.
- We discuss how these specifications can be implemented using standard tools (automatic theorem proving for quantifier-free first-order logic and BDDs), except for the use of Ternary Decision Diagrams (TDDs) [31] for cartesian abstraction. We show that the computational cost of constructing a `must`-transition relation is the same as that of constructing a `may`-transition relation.
- We show that our implementations are sound and (relatively) complete[1] with respect to their specifications in our calculus. Moreover, they conveniently model approximations in calls to a theorem prover as under-approximations of `must`-transitions and over-approximations of `may`-transitions.
- We prove that abstraction refinement is incremental for MTSs built using *cartesian* abstraction.

Predicate abstraction [17,14,32] is based on a set of predicates, $\Phi \stackrel{\text{def}}{=} \{\phi_1, \ldots, \phi_n\}$, typically quantifier-free formulas of first-order logic (e.g. `(x == y+1) || (x < y-5)`). An abstract state is induced by n-ary conjunctions, called *monomials*, with each predicate ϕ_i contributing either ϕ_i or $\neg\phi_i$. This abstraction identifies concrete states that satisfy the same predicates in Φ.

Given a set of states represented by a formula of quantifier-free first-order logic ψ, the set ψ' of abstract `may`-successors states is defined as the disjunction of all monomials η such that $\texttt{post}(\psi) \wedge \eta$ is satisfiable [17,14].[2] Computing ψ' can be done using automatic theorem proving for quantifier-free formulas, and [14] shows how to use a representation based on BDDs [6] at a propositional level to compactly represent the construction of ψ' as a disjunction of conjunctions. We can compute `must`-transitions by dualizing, in a logical sense, the above construction: for ψ as above, we show that the set of `must`-successors is the disjunction of all monomials η such that $\psi \wedge \tilde{\texttt{pre}}(\neg\eta)$ is unsatisfiable.[3]

Unfortunately, this approach is not incremental: adding a new predicate ϕ_{n+1} to Φ may not yield a refinement of the abstraction, and hence the entire abstraction may need to be recomputed. This shortcoming can be eliminated at the expense of enlarging the abstract state space: states are now built as disjunctions of abstract states from predicate abstraction. Using disjunctions can yield

[1] Which are perforce relative to the completeness of the underlying theorem prover.
[2] $\texttt{post}(\psi)$ is the set of immediate successor states of states satisfying ψ.
[3] $\tilde{\texttt{pre}}(\neg\eta)$ is the weakest precondition of states satisfying $\neg\eta$.

a must-component that is more precise than the one obtained from predicate abstraction, but can also be much more expensive: for n predicates, an abstraction using disjunctions can have 2^{2^n} states. This tradeoff between cost and precision is discussed in [8].

This limitation motivates the next layer of approximation: cartesian abstraction, which can be used on top of predicate abstraction in order to approximate sets of n-tuples by n-tuples of sets. We modify the work of [14] to synthesize abstract states and abstract may-successors for this composite abstraction, replacing BDDs by TDDs [31]. Then we construct must-transitions by dualizing, in the logical sense, the construction of may-transitions using cartesian abstraction.

We complete this framework with an algorithm for model checking any modal mu-calculus formula on an abstract MTS. Following [4,5], any (three-valued) model-checking problem on MTSs can be reduced to two traditional (two-valued) model-checking problems on regular LTSs.

The rest of the paper is organized as follows. Section 2 discusses background material on MTSs. Section 3 formally develops a relational calculus of abstractions and proves a basic result that permits the methods of analysis of this paper. In Section 4, we apply these methods to predicate and cartesian abstraction and prove that cartesian abstraction allows for incremental refinement. Section 5 discusses three-valued model-checking for MTSs, and Section 6 concludes.

2 Background: Abstract Modal Transition Systems

MTSs [24,23] are defined from labeled transition systems.

Definition 1 (Labeled transition systems). *A* labeled transition system *[27] (LTS) is a tuple* $\mathcal{K} = (\Sigma_K, \mathtt{Act}, \longrightarrow)$*, where* Σ_K *is a set of states,* Act *is a set of action symbols, and* $\longrightarrow \subseteq \Sigma_K \times \mathtt{Act} \times \Sigma_K$ *is a transition relation. We call* $\mathcal{K}$ finitely-branching *if for each* $s \in \Sigma_K$*, the set* $\{s' \in \Sigma_K \mid \exists \alpha \in \mathtt{Act}\colon (s, \alpha, s') \in \longrightarrow\}$ *is finite.*

A strategy to reason about a complex program represented by an LTS $\mathcal{C}$ consists of (i) generating from $\mathcal{C}$ an abstract LTS $\mathcal{A}$, (ii) checking whether $\mathcal{A}$ satisfies a behavioral property ϕ, and (iii) transferring those results to the original program $\mathcal{C}$. For (i) and (iii), standard practice [10,7] is to construct some $\mathcal{A}$ such that the initial states of $\mathcal{C}$ and $\mathcal{A}$ are related by a *simulation*.

Definition 2 (Simulation). *A relation* $\rho \subseteq \Sigma_C \times \Sigma_A$ *is a* simulation *[26] iff for any* $c\,\rho\,a$ *and* $c \to^\alpha c'$ *there is some* $a' \in \Sigma_A$ *such that* $a \to^\alpha a'$ *and* $c'\,\rho\,a'$*.*

The temporal logic $L_\forall$ whose abstract syntax is

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid (\forall\alpha)\phi \mid \nu Z.\phi \tag{1}$$

with $\alpha \in \mathtt{Act}$, variables $Z \in \mathtt{Var}$ for the greatest fixed point $\nu Z.\phi$, and usual semantics, expresses *universal properties* [29]. We assume here the semantics

of (closed) formulas over LTSs is defined as sets of states. For instance, the semantics of $(\forall\alpha)\phi$ is:

$$[\![(\forall\alpha)\phi]\!] \stackrel{\text{def}}{=} \{s \in \Sigma_K \mid \text{for all } s' \in \Sigma_K,\ s \rightarrow^{\alpha} s' \text{ implies } s' \in [\![\phi]\!]\}.$$

A simulation relation $c\,\rho\,a$ ensures that $a \in [\![\phi]\!]$ (read "a satisfies ϕ") implies $c \in [\![\phi]\!]$. Thus, we may verify any universal property $\phi \in L_\forall$ (such as "For all paths, nothing bad will happen") at c by (i) computing an abstract model $\mathcal{A}$, (ii) establishing a simulation ρ satisfying $c\,\rho\,a$, and (iii) verifying ϕ at a. Unfortunately, a negative check $a \notin [\![\phi]\!]$ does not imply anything about the truth or falsity of $c \notin [\![\phi]\!]$. At most, debugging information obtained from such a negative check may be used to construct a more concrete version of $\mathcal{A}$ (a refinement), hoping that this more precise model either renders a positive check or that refined debugging information eventually "applies" to $\mathcal{C}$ as well.

In this paper, we argue that a better approach consists of using MTSs instead of LTSs for representing abstractions of LTSs.

Definition 3 (MTS). *A* MTS *[24] is a pair* $\mathcal{K} = (\mathcal{K}^{\text{must}}, \mathcal{K}^{\text{may}})$*, where* $\mathcal{K}^{\text{must}} = (\Sigma_K, \texttt{Act}, \rightarrow_{\text{must}})$ *and* $\mathcal{K}^{\text{may}} = (\Sigma_K, \texttt{Act}, \rightarrow_{\text{may}})$ *are LTSs such that* $\rightarrow_{\text{must}} \subseteq \rightarrow_{\text{may}}$*.*

An LTS is simply a MTS $\mathcal{K}$ where $\mathcal{K}^{\text{must}}$ equals $\mathcal{K}^{\text{may}}$. The intuition behind the inclusion above is that transitions that are necessarily true ($\mathcal{K}^{\text{must}}$) are also possibly true ($\mathcal{K}^{\text{may}}$). Reasoning about the existence of transitions of MTSs can be viewed as reasoning with a three-valued logic with truth values `true`, `false`, and `unknown` [4]: transitions that are necessarily true are `true`, transitions that are possibly true but not necessarily true are `unknown`, and transitions that are not possibly true are `false`.

Definition 4 (Refinement [24]). *An MTS* $\mathcal{A}_1$ *is a* refinement *of an MTS* $\mathcal{A}_2$ *if there exists a relation* $\rho \subseteq \Sigma_{A_1} \times \Sigma_{A_2}$ *such that (i)* ρ *is a simulation from* $\mathcal{A}_1^{\text{may}}$ *to* $\mathcal{A}_2^{\text{may}}$ *and (ii)* ρ *is a simulation from* $\mathcal{A}_2^{\text{must}}$ *to* $\mathcal{A}_1^{\text{must}}$*. In that case, we also say that* $\mathcal{A}_2$ *is an* abstraction *of* $\mathcal{A}_1$*. We write* $\prec$ *for the greatest refinement relation between MTSs.*

MTSs can be used to both verify and refute *any* property of the full modal mu-calculus, which is defined as follows [22]:

$$\phi ::= \texttt{tt} \mid Z \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid (\exists\alpha)\phi \mid \mu Z.\phi \tag{2}$$

where $\alpha \in \texttt{Act}$, and $Z \in \texttt{Var}$ (variable for the least fixed point $\mu Z.\phi$).

Definition 5 (Semantics of modal logic [19]). *For a MTS* $\mathcal{K}$ *and any modal mu-calculus formula* ϕ*, we define a semantics* $[\![\phi]\!]_\sigma \in \mathcal{P}(\Sigma_K) \times \mathcal{P}(\Sigma_K)$*, where* $\mathcal{P}(\Sigma_K)$ *is the powerset of* Σ_K*, ordered by set inclusion,* $\sigma{:}\texttt{Var} \rightarrow \mathcal{P}(\Sigma_K) \times \mathcal{P}(\Sigma_K)$ *is an environment, and* $[\![\phi]\!]_\sigma^{\text{nec}}$ *and* $[\![\phi]\!]_\sigma^{\text{pos}}$ *are the projection of* $[\![\phi]\!]_\sigma$ *to its first and second component, respectively:*

1. $[\![\texttt{tt}]\!]_\sigma \stackrel{\text{def}}{=} \langle \Sigma_K,\ \Sigma_K \rangle$*;*

2. $[\![\neg\phi]\!]_\sigma \stackrel{\text{def}}{=} \langle \Sigma_K \setminus [\![\phi]\!]^{\text{pos}}_\sigma,\ \Sigma_K \setminus [\![\phi]\!]^{\text{nec}}_\sigma \rangle$;
3. $[\![\phi_1 \wedge \phi_2]\!]_\sigma \stackrel{\text{def}}{=} \langle [\![\phi_1]\!]^{\text{nec}}_\sigma \cap [\![\phi_2]\!]^{\text{nec}}_\sigma,\ [\![\phi_1]\!]^{\text{pos}}_\sigma \cap [\![\phi_2]\!]^{\text{pos}}_\sigma \rangle$;
4. $[\![(\exists\alpha)\phi]\!]_\sigma \stackrel{\text{def}}{=} \langle \{s \in \Sigma_K \mid \text{for some } s',\ s \rightarrow^a_{\text{must}} s' \text{ and } s' \in [\![\phi]\!]^{\text{nec}}_\sigma\},$
$\{s \in \Sigma_K \mid \text{for some } s',\ s \rightarrow^a_{\text{may}} s' \text{ and } s' \in [\![\phi]\!]^{\text{pos}}_\sigma\} \rangle$.

The treatment of negation is due to P. Kelb [20] and allows for verifying ($s \in [\![\phi]\!]^{\text{nec}}$) and refuting ($s \in [\![\neg\phi]\!]^{\text{nec}}$) property ϕ at state s. For brevity, we did not present the standard least-fixed point semantics of $\mu Z.\phi$ (e.g., see [19]).

Theorem 1 (Soundness and consistency of semantics [19]). *For any MTSs, formulas ϕ, ψ of the modal mu-calculus, and environments σ:*

1. $[\![\phi]\!]^{\text{nec}}_\sigma \subseteq [\![\phi]\!]^{\text{pos}}_\sigma$;
2. $[\![\phi \wedge \neg\phi]\!]^{\text{nec}}_\sigma = \emptyset$; *and* $[\![\phi \vee \neg\phi]\!]^{\text{pos}}_\sigma = \Sigma_K$; *that is, the semantics is consistent for* $[\![\ ]\!]^{\text{nec}}$ *and "complete" for* $[\![\ ]\!]^{\text{pos}}$;
3. *if* $c \prec a$, *then* $a \in [\![\phi]\!]^{\text{nec}}_\sigma$ *implies* $c \in [\![\phi]\!]^{\text{nec}}_\sigma$; *and* $c \in [\![\phi]\!]^{\text{pos}}_\sigma$ *implies* $a \in [\![\phi]\!]^{\text{pos}}_\sigma$; *that is, verification and refutation of ϕ are sound;*
4. *For LTSs,* $[\![\phi]\!]^{\text{nec}}_\sigma = [\![\phi]\!]^{\text{pos}}_\sigma$ *and corresponds to the standard semantics for labeled transition systems.*

The semantics $[\![\phi]\!]^{\text{nec}}$ (without negation and fixed points) is the one given by Larsen [23]; it produces a logical characterization of refinement for finitely branching[4] MTSs [23]. Since $s \notin [\![\phi]\!]^{\text{pos}}$ iff $s \in [\![\neg\phi]\!]^{\text{nec}}$, this logical characterization can be extended to the full mu-calculus (including negation). We thus obtain that $c \prec a$ iff for all ϕ of the modal mu-calculus, $[a \in [\![\phi]\!]^{\text{nec}} \Rightarrow c \in [\![\phi]\!]^{\text{nec}}]$.

3 A Relational Calculus for Abstract MTSs

In [12], abstract interpretation frameworks are systematically defined through description relations $\rho\colon \Sigma_C \times \Sigma_A$ with suitable properties. We provide a general calculus for specifying abstract MTSs based on such relations.

Definition 6 (Relational abstraction). *Let $\mathcal{A}_1 = (\mathcal{A}_1^{\text{must}}, \mathcal{A}_1^{\text{may}})$ be an MTS. Given a set Σ_{A_2} of abstract states and a total relation*[5] *$\rho\colon \Sigma_{A_1} \times \Sigma_{A_2}$, we define $\mathcal{A}_2 = (\Sigma_{A_2}, \texttt{Act}, \stackrel{\text{must}}{\longrightarrow}, \stackrel{\text{may}}{\longrightarrow})$ as follows:*

- $a_2 \rightarrow^\alpha_{\text{must}} a_2'$ *iff for all* $a_1 \in \Sigma_{A_1}$ *with* $a_1\,\rho\,a_2$ *there exists* $a_1' \in \Sigma_{A_1}$ *such that* $a_1'\,\rho\,a_2'$ *and* $a_1 \rightarrow^\alpha_{\text{must}} a_1'$;
- $a_2 \rightarrow^\alpha_{\text{may}} a_2'$ *iff there exist* $a_1 \in \Sigma_{A_1}$ *and* $a_1' \in \Sigma_{A_1}$ *such that* $a_1\,\rho\,a_2$, $a_1'\,\rho\,a_2'$, *and* $a_1 \rightarrow^\alpha_{\text{may}} a_1'$.

This definition is a tool to specify abstract MTSs. Its two components are similar to the universal abstraction $\alpha^{\forall\exists}$ and the (dual) existential abstraction $\alpha^{\exists\exists}$ defined in [11], and to the relations $R^{\forall\exists}$ and $R^{\exists\exists}$ in [12].

[4] `may`-components and `must`-components are finitely-branching.

[5] That is, $(\forall a_1 \in \Sigma_{A_1} \exists a_2 \in \Sigma_{A_2}\colon a_1\,\rho\,a_2) \wedge (\forall a_2 \in \Sigma_{A_2} \exists a_1 \in \Sigma_{A_1}\colon a_1\,\rho\,a_2)$.

Lemma 1. *Given $\mathcal{A}_1$ and $\mathcal{A}_2$ as above, $\mathcal{A}_2$ is a MTS and ρ is a refinement.*

Totality is a natural condition in applications and Definition 6 can express stepwise abstractions, products, sums, etc; moreover, it translates to other frameworks — e.g. the one based on Galois connections [10] — in the manner described in [12].

The specification in Definition 6 is also complete: given an MTS $\mathcal{A}_1$, any abstraction $\mathcal{A}_2$ of $\mathcal{A}_1$ via a total refinement relation $\prec$ can be constructed using Definition 6 by choosing ρ as $\prec$. The following example illustrates Definition 6.

Example 1. Let $\mathcal{A}_1$ be a complete MTS ($\mathcal{A}_1^{\text{may}}$ equals $\mathcal{A}_1^{\text{must}}$) whose infinite state space is given by all possible valuations of three integer variables `x`, `y`, and `z`. Any state c is of the form $\{\mathtt{x} \mapsto i, \mathtt{y} \mapsto j, \mathtt{z} \mapsto k\}$, for some integers i, j, k. Let us assume that transitions of $\mathcal{A}_1$ are those induced by the single assignment statement `x = z`, e.g. there is a transition from state c above to state $c' = \{\mathtt{x} \mapsto k, \mathtt{y} \mapsto j, \mathtt{z} \mapsto k\}$.

The predicates $\phi_1 \stackrel{\text{def}}{=} \mathtt{odd(x)}$, $\phi_2 \stackrel{\text{def}}{=} (\mathtt{y} > \mathtt{0})$, and $\phi_3 \stackrel{\text{def}}{=} (\mathtt{z} < \mathtt{0})$ induce an equivalence relation on the states of $\mathcal{A}_1$: two states are equivalent if they agree on all three predicates. Let Σ_{A_2} be the set of all sets of equivalence classes of states of $\mathcal{A}_1$. Therefore, states of $\mathcal{A}_2$ are representable as boolean formulas built from the ϕ_i's. By Definition 6, there is a `may`-transition from a to a' in Σ_{A_2} iff there are $c \in a$ and $c' \in a'$ such that c has a transition to c' in $\mathcal{A}_1$. Dually, there is a `must`-transition from a to a' iff, *for all* $c \in a$, there exists $c' \in a'$ such that c has a transition to c' in $\mathcal{A}_1$.

For instance, there is (i) a `may`-transition from the state $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the states $\phi_1 \wedge \phi_2 \wedge \phi_3$ and $\neg\phi_1 \wedge \phi_2 \wedge \phi_3$; (ii) a `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the disjunction of monomials $(\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\neg\phi_1 \wedge \phi_2 \wedge \phi_3)$; but (iii) *no* `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to any monomial, e.g. $\phi_1 \wedge \phi_2 \wedge \phi_3$ or $\neg\phi_1 \wedge \phi_2 \wedge \phi_3$.

4 Implementation of Relationally Specified MTSs

In this section, we consider in turn predicate abstraction (also called "boolean abstraction") and cartesian abstraction. When applied to `may`-transition relations only, the specifications and implementations we present coincide with traditional "conservative" abstractions. We implement these specifications with standard tools (automatic theorem-proving for quantifier-free first-order logic and BDDs), except for the use of TDDs. We show that the computational cost of constructing a `must`-transition relation is the same as that of constructing a `may`-transition relation. We then show that our implementations are sound and complete (relatively to the completeness of the underlying theorem prover) with respect to their specifications. Moreover, they conveniently model approximations in calls to a theorem prover as under-approximations of `must`-transitions and over-approximations of `may`-transitions. Importantly, we prove that abstraction refinement is incremental for MTSs built using cartesian abstraction.

Notation. For any predicate η on a set Σ_S of states, for any label $\alpha \in \mathtt{Act}$, the post operator [10] and weakest precondition [15] are defined as

$$\mathtt{post}_\alpha(\eta) \stackrel{\text{def}}{=} \{s' \in \Sigma_S \mid \exists s \in \Sigma_S\colon s \models \eta,\ s \rightarrow^\alpha s'\}$$
$$\tilde{\mathtt{pre}}_\alpha(\eta) \stackrel{\text{def}}{=} \{s \in \Sigma_S \mid \forall s' \in \Sigma_S\colon s \rightarrow^\alpha s' \text{ implies } s' \models \eta\}.$$

These operators satisfy several interesting relationships [10]. Here we only use the property that, for any predicates η, ψ on states, $\mathtt{post}_\alpha(\psi) \wedge \eta$ is satisfiable if and only if $\psi \wedge \neg\tilde{\mathtt{pre}}_\alpha(\neg\eta)$ is satisfiable.

Methodological assumptions. We assume that an abstract program is built by converting each program statement from a transformer operating on concrete states to a transformer operating on abstract states, as illustrated in Example 1. For notational convenience, we focus in what follows on the abstraction of a single program statement, and hence consider MTSs, $\mathtt{post}$, and $\tilde{\mathtt{pre}}$ without explicit action labels. For a given program statement and a quantifier-free formula η, we assume that $\tilde{\mathtt{pre}}(\eta)$ is *quantifier-free* as well. This is the case for usual programming language constructs [17] and enables the use of decision procedures as implemented in tools such as SVC [14].

Predicate Abstraction. Predicate abstraction [17,14,32] collapses an infinite-state LTS into a finite-state MTS defined by choosing finitely many quantifier-free formulas of first-order logic.

Specification. The abstract states in the predicate abstraction are built out of monomials over predicates. Each abstract state corresponds to a set of concrete states that satisfy the same set of predicates. Formally, given a finite set of quantifier-free formulas of first-order logic, $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, and a "bit-vector" $b \in \{0,1\}^n$, we write $\langle b, \Phi\rangle$ for a monomial whose ith conjunct is ϕ_i if $b_i = 1$, and $\neg\phi_i$ otherwise.

Definition 7 (Predicate abstraction). *Given an LTS $\mathcal{S}$ and a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic, we derive a finite-state abstract MTS $\mathcal{B}_\Phi$ following Definition 6 ($\mathcal{A}_1$ is $\mathcal{S}$ and $\mathcal{A}_2$ is $\mathcal{B}_\Phi$) in such a way that:*

- $\rho \stackrel{\text{def}}{=} \rho^\Phi_{\mathrm{b}} \subseteq \Sigma_S \times \{0,1\}^n$, *where* $s\,\rho^\Phi_{\mathrm{b}}\,b$ *iff* $s \models \langle b, \Phi\rangle$; *and*
- $\Sigma_{\mathcal{B}_\Phi} \stackrel{\text{def}}{=} \{b \in \{0,1\}^n \mid s\,\rho^\Phi_{\mathrm{b}}\,b \text{ for some } s \in \Sigma_S\}$, *which makes* ρ^Φ_{b} *total.*

Implementing $\mathtt{may}$*-successors of a predicate abstraction.* Current tool-supported predicate-abstraction frameworks [17,14,32,9,2,1] can be viewed as constructing an abstraction of the $\mathtt{may}$-component of $\mathcal{B}_\Phi$ defined above. We now review how to compute the set of abstract $\mathtt{may}$-successors for a single program statement.

Following [14], we use BDDs over boolean variables $x_1, x_2, \ldots, x_n$ as representations of such sets. If ψ is a boolean combination of $\{\phi_i \mid m \le i \le n\}$, we compute in (3) below a BDD, denoted by $H^{\mathrm{may}}(\psi, \mathrm{true}, m)$, for representing the set of $\mathtt{may}$-successors of ψ. The definition of H^{may} is essentially the definition of

H in [14] where $\texttt{post}(\psi) \wedge \eta$ is replaced by $\psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta)$ (to facilitate dualizing this construction later).

$$H^{\text{may}}(\psi,\eta,i) \stackrel{\text{def}}{=} \begin{cases} (x_i \wedge H^{\text{may}}(\psi, \eta \wedge \phi_i, i+1)) \\ \vee(\neg x_i \wedge H^{\text{may}}(\psi, \eta \wedge \neg\phi_i, i+1)) & \text{if } 0 < i \leq n, \\ 1 \quad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta) \text{ is satisfiable,} \\ 0 \quad \text{if } i = n+1 \text{ and } \psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta) \text{ is unsatisfiable.} \end{cases} \tag{3}$$

The BDD in (3) can be computed using standard BDD operations [6] plus the optimizations discussed in [14], while the satisfiability checks can be computed by calling a theorem prover. Unwinding the recursion in the definition above, it is clear that the set of may-successors of ψ computed by H^{may} is:

$$\texttt{next}(\psi)^{\text{may}}_{\text{b}} \stackrel{\text{def}}{=} \{b' \in \Sigma_B \mid \psi \wedge \neg\tilde{\texttt{pre}}(\neg\langle b', \Phi\rangle) \text{ is satisfiable } \}. \tag{4}$$

Implementing must*-successors of a predicate abstraction.* The logical duality of may versus must is captured by replacing the satisfiability check of $\psi \wedge \neg\tilde{\texttt{pre}}(\neg\eta)$ in (3) by the unsatisfiability check of $\psi \wedge \tilde{\texttt{pre}}(\neg\eta)$ in the following equation (5).

$$H^{\text{must}}(\psi,\eta,i) \stackrel{\text{def}}{=} \begin{cases} (x_i \wedge H^{\text{must}}(\psi, \eta \wedge \phi_i, i+1)) \\ \vee(\neg x_i \wedge H^{\text{must}}(\psi, \eta \wedge \neg\phi_i, i+1)) & \text{if } 0 < i \leq n, \\ 1 \quad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\texttt{pre}}(\neg\eta) \text{ is unsatisfiable,} \\ 0 \quad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\texttt{pre}}(\neg\eta) \text{ is satisfiable.} \end{cases} \tag{5}$$

Thus, the set of must-successors of ψ represented by the BDD $H^{\text{must}}(\psi, \text{true}, m)$ is:

$$\texttt{next}(\psi)^{\text{must}}_{\text{b}} \stackrel{\text{def}}{=} \{b' \in \Sigma_B \mid \psi \wedge \tilde{\texttt{pre}}(\neg\langle b', \Phi\rangle) \text{ is unsatisfiable } \}. \tag{6}$$

We now show that the BDDs computed by H^{may} and H^{must} represent exactly the transitions specified in Definition 7.

Theorem 2 (Soundness and completeness).

- $b \rightarrow_{\text{may}} b'$ *in* $\mathcal{B}_\Phi$ *iff* $b' \in \texttt{next}(\langle b, \Phi\rangle)^{\text{may}}_{\text{b}}$*;*
- $b \rightarrow_{\text{must}} b'$ *in* $\mathcal{B}_\Phi$ *iff* $b' \in \texttt{next}(\langle b, \Phi\rangle)^{\text{must}}_{\text{b}}$*.*

Proof. The proof follows from the direct application of the definitions and is omitted here due to space constraints.

Cost. In the worst case, the computation of $H^{\text{may}}(\langle b, \Phi\rangle, \text{true}, m)$ makes $O(2^n)$ calls to the theorem prover. Similarly, the computation of $H^{\text{must}}(\langle b, \Phi\rangle, \text{true}, m)$ makes at most $O(2^n)$ calls to the theorem prover. Hence, *the complexity of computing* H^{must} *is the same as the complexity of computing* H^{may}*.*

Note that optimizations for computing H^{may} discussed in [14] can also be used when computing H^{must}. Our algorithm also accommodates the complexity of theorem-proving by allowing the sound over-approximation of H^{may} and

under-approximation of H^{must} as follows: in both cases, simply convert the absence of an answer, when truncating the computation performed by the satisfiability checker, into "satisfiable".

Predicate-Cartesian Abstraction. Unfortunately, predicate abstraction of MTSs is not incremental: adding a new predicate ϕ_{n+1} to Φ may not yield a refinement of the abstraction, and hence the entire abstraction may need to be recomputed. This is illustrated by the following example.

Example 2. Revisiting Example 1, if $\Phi = \{\phi_2, \phi_3\}$, then $\mathcal{B}_\Phi$ has four states, each with a `must`-transition to itself. However, adding the predicate ϕ_1 to Φ, there are no `must`-transitions from the abstract state $\phi_1 \wedge \phi_2 \wedge \phi_3$ (111) in $\mathcal{B}_{\{\phi_1\} \cup \Phi}$. This is quite unfortunate: the information about variable `y` is lost even though `y` is absent from the assignment `x = z`. But in Example 1 we saw that there *is* a `must`-transition from $\phi_1 \wedge \phi_2 \wedge \phi_3$ to the *disjunction* $(\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\neg\phi_1 \wedge \phi_2 \wedge \phi_3)$ that correctly captures the "absence of effect" on `y`.

Computing `must`-transitions with abstract states of the above kind can be expensive: given n predicates, there are a possible 2^{2^n} such states. This motivates our next topic: cartesian abstraction.

Specification. The basic idea behind cartesian abstraction is to approximate sets of tuples by a tuple of sets. For instance, a set $\{\langle 0,1\rangle, \langle 1,1\rangle\}$ is represented by $\{\langle \star, 1\rangle\}$, where $\star$ is used as a wildcard for different values (such as 0 and 1 in this example). Formally, given a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic and a "tri-vector" $c \in \{0, 1, \star\}^n$, we write $\langle c, \Phi\rangle$ for a monomial whose ith conjunct is ϕ_i if $c_i = 1$, $\neg\phi_i$ if $c_i = 0$, and `true` otherwise. Abstract states in $\mathcal{C}_\Phi$ are built out of "tri-vectors" of length n.

Definition 8 (Predicate-cartesian abstraction). *Given an LTS $\mathcal{S}$ and a finite set $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ of quantifier-free formulas of first-order logic, we derive a finite-state abstract MTS $\mathcal{C}_\Phi$ following Definition 6 ($\mathcal{A}_1$ is $\mathcal{S}$ and $\mathcal{A}_2$ is $\mathcal{C}_\Phi$) in such a way that:*

- $\rho \stackrel{\text{def}}{=} \rho_{\text{c}}^\Phi \circ \rho_{\text{b}}^\Phi \subseteq \Sigma_S \times \{0, 1, \star\}^n$, *where* $b \, \rho_{\text{c}}^\Phi \, c$ *iff* $\forall 1 \leq i \leq n : [c_i \neq \star \;\Rightarrow\; b_i = c_i]$; *and*
- $\Sigma_{\mathcal{C}_\Phi} \stackrel{\text{def}}{=} \{c \in \{0, 1, \star\}^n \mid b \, \rho_{\text{c}}^\Phi \, c$ *for some* $b \in \Sigma_{\mathcal{B}_\Phi}\}$, *which makes* ρ_{c}^Φ *total.*

The symbol $\star$ means "don't care" in the above definition. It is easy to show that, by construction, we have:

$$s \, (\rho_{\text{c}}^\Phi \circ \rho_{\text{b}}^\Phi) \, c \qquad \text{iff} \qquad s \models \langle c, \Phi\rangle. \tag{7}$$

Note that the abstract MTS $\mathcal{C}'_\Phi$ obtained by abstracting $\mathcal{B}_\Phi$ (whose states are vectors of n-bits) with ρ_{c}^Φ is typically less precise than $\mathcal{C}_\Phi$. For instance, in the case of Examples 1 and 2 again, $\mathcal{C}'_\Phi$ would not contain any `must`-transition from state 111 (i.e., $\phi_1 \wedge \phi_2 \wedge \phi_3$), while $\mathcal{C}_\Phi$ does contain a `must`-transition from 111 to state $\star 11$.

The MTS $\mathcal{C}_\Phi$ supports an approximate union operation, defined using the pointwise application of Kleene's alignment operator [21]: $c \cup c' \stackrel{\text{def}}{=} c''$, where $c''_i = c_i$ if $c_i = c'_i$ and $\star$ otherwise. This operation thus approximates disjunctions (sets) of monomials by tri-vectors. As previously mentioned, cartesian abstraction allows for incremental abstraction refinement:

Theorem 3 (Incremental refinement). *For $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ and $\Psi = \Phi \cup \{\phi_{n+1}, \phi_{n+2}, \ldots, \phi_{n+m}\}$, the MTS $\mathcal{C}_\Psi$ is a refinement of the MTS $\mathcal{C}_\Phi$.*

Proof. The refinement $\rho \subseteq \Sigma_{C_\Psi} \times \Sigma_{C_\Phi}$ is given by $\{(c', c) \mid c \text{ is a prefix of } c'\}$.

Implementing `may`*-successors of a predicate-cartesian abstraction.* Instead of representing the abstract post-operator with a BDD as in the predicate abstraction case, we now use a Ternary Decision Diagram [31], writing $[x/v]$ for the replacement of variable x with value $v \in \{0, 1, \star\}$:

$$G^{\text{may}}(\psi, \eta, i) \stackrel{\text{def}}{=} \begin{cases} ([x_i/1] \wedge G^{\text{may}}(\psi, \eta \wedge \phi_i, i+1)) \vee ([x_i/0] \wedge G^{\text{may}}(\psi, \eta \wedge \neg\phi_i, i+1)) \\ \vee ([x_i/\star] \wedge G^{\text{may}}(\psi, \eta, i+1)) \qquad \text{if } 0 < i \leq n, \\ 1 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \neg \tilde{\mathtt{pre}}(\neg\eta) \text{ is satisfiable,} \\ 0 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \neg \tilde{\mathtt{pre}}(\neg\eta) \text{ is unsatisfiable.} \end{cases} \tag{8}$$

The function G^{may} essentially computes the abstract post-operator of SLAM [1]. Unwinding the recursion in the above definition, the set of `may`-successors of ψ represented by the TDD $G^{\text{may}}(\psi, \text{true}, m)$ is:

$$\mathtt{next}(\psi)^{\text{may}}_{\text{c}} \stackrel{\text{def}}{=} \{c' \in \Sigma_C \mid \psi \wedge \neg \tilde{\mathtt{pre}}(\neg\langle c', \Phi\rangle) \text{ is satisfiable } \}. \tag{9}$$

Implementing `must`*-successors of a predicate-cartesian abstraction.* The TDD $G^{\text{must}}(\psi, \text{true}, m)$, defined below, represents the set of `must`-successors of ψ. Similar to our presentation of predicate abstraction, we are capturing the logical duality of `may` versus `must` by replacing the satisfiability check of $\psi \wedge \neg \tilde{\mathtt{pre}}(\neg\eta)$ by the unsatisfiability check of $\psi \wedge \tilde{\mathtt{pre}}(\neg\eta)$ in the following equation:

$$G^{\text{must}}(\psi, \eta, i) \stackrel{\text{def}}{=} \begin{cases} ([x_i/1] \wedge G^{\text{must}}(\psi, \eta \wedge \phi_i, i+1)) \vee ([x_i/0] \wedge G^{\text{must}}(\psi, \eta \wedge \neg\phi_i, i+1)) \\ \vee ([x_i/\star] \wedge G^{\text{must}}(\psi, \eta, i+1)) \qquad \text{if } 0 < i \leq n, \\ 1 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\mathtt{pre}}(\neg\eta) \text{ is unsatisfiable,} \\ 0 \qquad \text{if } i = n+1 \text{ and } \psi \wedge \tilde{\mathtt{pre}}(\neg\eta) \text{ is satisfiable.} \end{cases} \tag{10}$$

Again, by unwinding the recursion, the set of `must`-successors of ψ represented by the TDD $G^{\text{must}}(\psi, \text{true}, m)$ is thus defined by:

$$\mathtt{next}(\psi)^{\text{must}}_{\text{c}} \stackrel{\text{def}}{=} \{c' \in \Sigma_C \mid \psi \wedge \tilde{\mathtt{pre}}(\neg\langle c', \Phi\rangle) \text{ is unsatisfiable } \}. \tag{11}$$

The following theorem states that the TDDs computed by G^{may} and G^{must} represent exactly the transitions specified in Definition 8.

Theorem 4 (Soundness and completeness).

- $c \to_{\text{may}} c'$ *in* $\mathcal{C}_\Phi$ *iff* $c' \in \mathtt{next}(\langle c, \Phi\rangle)^{\text{may}}_{\text{c}}$;
- $c \to_{\text{must}} c'$ *in* $\mathcal{C}_\Phi$ *iff* $c' \in \mathtt{next}(\langle c, \Phi\rangle)^{\text{must}}_{\text{c}}$.

Proof. Similar to the proof of Theorem 2.

Cost. In the worst case, the computation of $G^{\text{may}}(\langle c, \Phi\rangle, \text{true}, m)$ makes $O(3^n)$ calls to the theorem prover. Similarly, the computation of $G^{\text{must}}(\langle c, \Phi\rangle, \text{true}, m)$ makes at most $O(3^n)$ calls to the theorem prover. Therefore, *the complexity of computing* G^{must} *is the same as the complexity of computing* G^{may}.

Note that the heuristics discussed in [1] for approximating the calculation of $G^{\text{may}}(\langle c, \Phi\rangle, \text{true}, m)$ by restricting the expansion of the recursion to a fixed depth (rather than n) can be applied when computing $G^{\text{must}}(\langle c, \Phi\rangle, \text{true}, m)$ as well. Again, the absence of answers from the theorem prover for satisfiability checks can be interpreted as "satisfiable" to yield a sound over-approximation of G^{may} and under-approximation of G^{must}.

5 Three-Valued Model Checking on MTSs

The automatic-abstraction algorithms of the previous section can be used to generate a MTS $\mathcal{A}_2$ which, by construction, is guaranteed to be an abstraction (as defined in Definition 4) of a given, possibly initial and concrete, system $\mathcal{A}_1$. By Theorem 1, we can check a modal mu-calculus formula ϕ on $\mathcal{A}_1$ by analyzing $\mathcal{A}_2$ instead, resulting in three possible answers: either (i) ϕ is necessarily true on $\mathcal{A}_2$ — its initial state is contained in $[\![\phi]\!]^{\text{nec}}$ — and hence ϕ holds for $\mathcal{A}_1$ (the answer is `true`), or (ii) ϕ is only possibly true on $\mathcal{A}_2$ — its initial state is contained in $[\![\phi]\!]^{\text{pos}}$ only — and whether ϕ holds on $\mathcal{A}_1$ is unknown (the answer is `unknown`), or (iii) ϕ is not possibly true on $\mathcal{A}_2$ — its initial state is not contained in $[\![\phi]\!]^{\text{pos}}$ — and ϕ does not hold on $\mathcal{A}_1$ (the answer is `false`). We are thus left with a three-valued model-checking problem on MTSs which, following [5], can be reduced to two model-checking problems on LTSs as follows.

First, we rewrite formula ϕ to a formula ϕ^+ in positive normal form defined over all the clauses of (1) plus (2) by pushing all negations in ϕ inwards so that they apply only to `tt` or `ff` in ϕ^+. This is done using the classic rewrite rules: $\neg\neg\phi = \phi$, $\neg(\phi_1 \wedge \phi_2) = (\neg\phi_1) \vee (\neg\phi_2)$, $\neg((\exists\alpha)\phi) = (\forall\alpha)(\neg\phi)$, and $\neg(\mu Z.\phi) = \nu Z.(\neg\phi)$. Then, we translate ϕ^+ into a formula $T(\phi)$ by applying the following translation rules: for all $\alpha \in \texttt{Act}$, replace all occurrences of $(\forall\alpha)$ in ϕ^+ by $(\forall\alpha_\forall)$ and replace all occurrences of $(\exists\alpha)$ in ϕ^+ by $(\exists\alpha_\exists)$.

Second, from the MTS $\mathcal{A}_2 = (\Sigma_{A_2}, \texttt{Act}, \rightarrow_{\text{must}}, \rightarrow_{\text{may}})$, we define two LTSs $\mathcal{A}_2^{\text{pess}}$ and $\mathcal{A}_2^{\text{opt}}$, representing respectively the *pessimistic* and *optimistic* interpretations of $\mathcal{A}_2$ (see [5]). These two LTSs are defined over the set

$$\texttt{Act}^{\text{c}} \stackrel{\text{def}}{=} \{\alpha_\forall \mid \alpha \in \texttt{Act}\} \cup \{\alpha_\exists \mid \alpha \in \texttt{Act}\} \tag{12}$$

of action symbols. Precisely, we define $\mathcal{A}_2^{\text{pess}} = (\Sigma_{A_2}, \texttt{Act}^{\text{c}}, \rightarrow_{\text{pess}})$ with

$$(s, \alpha_\forall, s') \in \rightarrow_{\text{pess}} \text{ if } (s, \alpha, s') \in \rightarrow_{\text{may}} \tag{13}$$

$$(s, \alpha_\exists, s') \in \rightarrow_{\text{pess}} \text{ if } (s, \alpha, s') \in \rightarrow_{\text{must}} \tag{14}$$

and we define $\mathcal{A}_2^{\text{opt}} = (\Sigma_{A_2}, \texttt{Act}^{\text{c}}, \rightarrow_{\text{opt}})$ with

$$(s, \alpha_\forall, s') \in \rightarrow_{\text{opt}} \text{ if } (s, \alpha, s') \in \rightarrow_{\text{must}} \tag{15}$$

$$(s, \alpha_\exists, s') \in \rightarrow_{\text{opt}} \text{ if } (s, \alpha, s') \in \rightarrow_{\text{may}}. \tag{16}$$

Finally, we model-check the modal mu-calculus formula $T(\phi)$ on the LTSs $\mathcal{A}_2^{\text{pess}}$ and $\mathcal{A}_2^{\text{opt}}$, and combine the results as specified in the following theorem.

Theorem 5 (Correctness of model checking algorithm). *Given a MTS $\mathcal{A}_2$ and a modal mu-calculus formula ϕ, let $T(\phi)$, $\mathcal{A}_2^{\text{pess}}$, and $\mathcal{A}_2^{\text{opt}}$ be the formula and the two LTSs (respectively) as defined above. For any state $s \in \Sigma_{A_2}$, we then have*

1. $s \in [\![\phi]\!]^{\text{nec}}$ *iff* $(\mathcal{A}_2^{\text{pess}}, s) \models T(\phi)$
2. $s \in [\![\phi]\!]^{\text{pos}}$ *iff* $(\mathcal{A}_2^{\text{opt}}, s) \models T(\phi)$.

Proof. By induction on the structure of ϕ.

The previous theorem is similar to Theorem 3 of [5]. It reduces three-valued model checking of MTSs to two traditional (two-valued) model-checking problems on regular LTSs, namely $(\mathcal{A}_2^{\text{pess}}, s) \models T(\phi)$ and $(\mathcal{A}_2^{\text{opt}}, s) \models T(\phi)$. Since the transformations performed to obtain $T(\phi)$, $(\mathcal{A}_2^{\text{pess}}, s)$, and $(\mathcal{A}_2^{\text{opt}}, s)$ can be done in constant space and time linear in the size of the formula and MTS respectively, three-valued model checking on MTSs has the same time and space complexity as two-valued model checking on LTSs. Moreover, the problem can be solved in practice using existing model-checking tools for LTSs, with all the optimizations that these tools may already implement. In particular, if the refined system $\mathcal{A}_1$ is concrete and composed of multiple concurrent LTSs or of recursive procedures (LTSs extended with a "call-stack"), the abstraction algorithms of the previous section will preserve the architecture of $\mathcal{A}_1$ when generating $\mathcal{A}_2$, and existing tools for model-checking concurrent or pushdown systems can be applied to $\mathcal{A}_2^{\text{pess}}$ and $\mathcal{A}_2^{\text{opt}}$.

6 Conclusions

We developed a framework for automatic program abstraction based on modal transition systems. This framework can be used for model-checking any formula of the modal mu-calculus, and is also applicable to the verification of concurrent and pushdown systems. It uses cartesian abstraction, implemented with TDDs and quantifier-free first-order-logic theorem-proving, to extend existing predicate-abstraction techniques to the verification of formulas containing arbitrarily nested path quantifiers. Cartesian abstraction has no significant cost overhead and is compatible with the standard incremental refinement process for adding more predicates.

Acknowledgments. We wish to thank Glenn Bruns and David Schmidt for inspiring discussions and helpful comments.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In T. Margaria and W. Yi, editors, *Proceedings of TACAS'2001*, volume 2031 of *LNCS*, pages 268–283, Genova, Italy, April 2001. Springer Verlag.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the Seventh International SPIN Workshop (SPIN 2000)*, volume 1885, pages 113–130. Springer Verlag, 2000.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions on infinite state systems compositionally and automatically. In A. J. Hu and M. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427, pages 319–331, Vancouver, Canada, 1998. Springer Verlag.
4. G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287. Springer Verlag, July 1999.
5. G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *Proceedings of CONCUR'2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer Verlag, August 2000.
6. R. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
7. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
8. R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS'95: Proc. 2d. Static Analysis Symposium*, Lecture Notes in Computer Science 983, pages 51–63. Springer, 1995.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd Intl' Conference on Software Engineering*, June 2000.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
11. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.
12. D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
13. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.
14. S. Das, D. L. Dill, and S. Park. Experience with Predicate Astraction. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th International Conference on Computer-Aided Verification*, pages 160–172, Trento, Italy, July 1999. Springer Verlag.
15. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
16. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

17. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Grumberg O., editor, *Conference on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.
18. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.
19. M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In D. Sands, editor, *Proceedings of the European Symposium on Programming (ESOP'2001)*, volume 2028 of *LNCS*, pages 155–169, Genova, Italy, April 2001. Springer Verlag.
20. P. Kelb. Model checking and abstraction: a framework preserving both truth and failure information. Technical Report OFFIS, University of Oldenburg, Germany, 1994.
21. S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
22. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
23. K. G. Larsen. Modal Specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 232–246. Springer Verlag, June 12–14 1989. International Workshop, Grenoble, France.
24. K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Third Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.
25. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11–44, January 1995.
26. R. Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489, London, United Kingdom, 1971. British Computer Society.
27. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
28. D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *In Proc. of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, 1989.
29. A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1985.
30. H. Saidi and N. Shankar. Abstract and model check while you prove. In *Proc. of the 11th Conference on Computer-Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 443–454. Springer, 1999.
31. T. Sasao. Ternary Decision Diagrams — Survey. In *Proceedings of the 27th International Symposium on Multi-valued Logic*, pages 241–250. IEEE, 1997.
32. W. Visser, S. J. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proc. of Formal Methods in Software Practice (FMSP'00)*, pages 3–12, Portland, Oregon, August 2000.

Efficient Multiple-Valued Model-Checking Using Lattice Representations

Marsha Chechik, Benet Devereux, Steve Easterbrook, Albert Y.C. Lai, and Victor Petrovykh

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: {chechik,benet,sme,trebla,victor}@cs.toronto.edu

Abstract. Multiple-valued logics can be effectively used to reason about incomplete and/or inconsistent systems, e.g. during early software requirements or as the systems evolve. We specify multiple-valued logics using finite lattices. In this paper, we use lattice representation theory to cast the multiple-valued model-checking problem in terms of symbolic operations on classical sets of states, provided the lattices are distributive. This allows us to partially reuse existing symbolic model-checking technology and improve efficiency over previous implementations that were based on multiple-valued decision diagrams.

1 Introduction

Multiple-valued logics provide an interesting alternative to classical boolean logic for modeling and reasoning about systems. By allowing additional truth values, they support the explicit modeling of uncertainty and disagreement. For these reasons, they have been explored for a variety of applications in databases [19], knowledge representation [20], machine learning [24], and circuit design [22].

A number of specific multi-valued logics have been proposed and studied. For example, Łukasiewicz [23] first introduced a three-valued logic to allow for propositions whose truth values are 'unknown', while Belnap [1] proposed a four-valued logic that also introduces the value 'both' (i.e. "true *and* false"), to handle inconsistent assertions in database systems. Each of these logics can be generalized to allow for different levels of uncertainty or disagreement. In practice, it is useful to be able to choose different multi-valued logics for different modeling tasks.

The motivations that led to the development of these logics clearly apply to the modeling of software behaviour, especially the exploratory modeling used in the early stage of requirements engineering and architectural design.

- We need to allow for *uncertainty* – for example, we may not yet know whether some behaviours should be possible;
- We need to allow for *disagreement* – for example, different stakeholders may disagree about how the systems should behave;
- We need to represent *relative importance* – for example, in the case where some behaviours are essential and others may or may not be implemented.

For reasoning about dynamic properties of systems, existing modal logics can be extended to the multi-valued case. Fitting [17, 18] suggests two different approaches

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 441–455, 2001.
© Springer-Verlag Berlin Heidelberg 2001

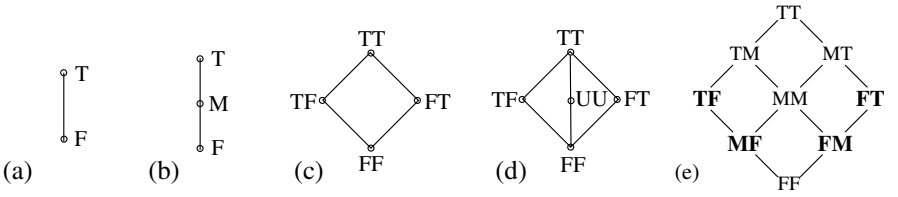


Fig. 1. (a) $\mathbf{2}$, the classical logic lattice; (b) $\mathbf{3}$, a 3-valued logic lattice, representing uncertainty; (c) $\mathbf{2x2}$, a 4-valued logic lattice, representing disagreement; (d) the 5-valued lattice, representing disagreement with "unknowns"; (e) a lattice $\mathbf{3X3}$.

for doing this: the first extends the interpretation of atomic formulae in each world to be multi-valued; the second also allows multi-valued accessibility relations between worlds. The latter approach is more general, and our work on logic in this paper is somewhat similar to his. Three-valued logic has received most interest. It has been shown to be useful for analyzing programs using abstract interpretation [9, 25], and for analyzing partial models [3, 4]. Bruns and Godefroid also proved that automata-theoretic model-checking on 3-valued predicates reduces to classical model-checking. In our recent work [8], we showed that this result extends to many-valued logics, where the values form a total order (e.g., "True", "Likely", "Undecided", "Unlikely", "False"). Work has also been done on deciding a more general class of multi-valued logics. In particular, the work of Hähnle and others [21, 27] has led to the development of several theorem-provers for first-order multi-valued logics.

This paper deals with symbolic model-checking for multi-valued logics. In our earlier work [10, 7], we identified a useful family of multiple-valued logics, known as the quasi-boolean logics [2] (or *de Morgan* logics [15]). A quasi-boolean logic has truth values that form a finite lattice, with a negation operator chosen such that the negation of each value is its image under horizontal symmetry. Conjunction and disjunction are defined as meet and join in the lattice, respectively. A few quasi-boolean logics are represented in Figure 1. Classical logic, referred to as $\mathbf{2}$, is given in Figure 1(a). The lattice in Figure 1(c) represents possible disagreement between two views: TT and FF represent unanimous agreement on true or false; TF and FT, two possible types of disagreement. The lattice in Figure 1(d) represents disagreement when some information is not known by either view. This sometimes happens when merging two partial views, represented by state machines. Suppose that a state s is present in one state machine, but a variable v is not. Suppose further that the variable v is present in the other state machine, but the state s is not. Then neither state machine can give the value for variable v in state s. Such situation requires introduction of a new logic value, "unknown", represented in Figure 1(d) by value UU [16].

We are interested in building a model-checker that takes as its input a particular quasi-boolean logic, represented as a lattice of truth values $(\mathcal{L}, \sqsubseteq)$, a state machine model, represented as a multiple-valued Kripke structure, and a temporal logic property expressed in CTL enriched with multiple-valued semantics, and returns the truth value that the property has in the initial state(s) and a counter-example, if applicable. Success of symbolic model-checking in a given domain (probabilistic, multiple-valued, timed, etc.) depends on being able to create efficient algorithms for manipulating the sets of states in which a property holds. We need to calculate union, intersection, complement, and backwards reachability (for computing predecessors). For model checking

in a given multiple-valued logic, we can treat these as operations over multiple-valued sets: sets whose membership functions are multiple-valued.

We have explored a number of choices for efficient computation of these operations over multiple-valued sets in our previous work:

- We can use MDDs, the multiple-valued extension of binary decision diagrams [5], as a canonical representation of the multiple-valued membership function. A multiple-valued model-checker based on MDDs is described in [7].
- We can represent the multiple-valued set as a collection of classical sets, one for each value of the logic. This approach has been taken in [10], using a Multiple-Valued Binary-Terminal Decision Diagram (MBTDD) [13] to represent each set.

Clearly, multi-valued logics with a finite number of values do not add expressive power to the modeling. However, in many cases they produce a significantly smaller state space then the alternative approach of introducing additional boolean predicates. Still, the above approaches suffer from a fairly high running time ($O(|\mathcal{L}|^{3n} \times h \times |p|$, where h is the height of the lattice representing the logic, $|\mathcal{L}|$ is the number of logic values, $|p|$ is the size of the formula under analysis, and n is the number of atomic propositions in the system), and we are interested in further optimizations of the model-checker.

In this paper, we describe a third alternative. We represent the multiple-valued sets as a collection of classical sets, but exploit a result of lattice theory to reduce the number of such sets that we need to represent. In a distributive lattice, any element of the lattice can be represented uniquely as the join of a subset of the join-irreducible elements of the lattice. This provides us with a compact encoding of the multiple-valued set membership function. This encoding supports efficient computation of the necessary set operations, and hence significantly reduces the running time of our model-checker.

The paper is organized as follows: Section 2 introduces multiple-valued sets and multiple-valued set operations, whereas Section 3 shows how to represent these using join-irreducible elements of the lattice describing the logic. We review semantics of multiple-valued CTL and multiple-valued Kripke structures, and present the model-checking algorithm in Section 4. We compare the performance of this model-checker with our previous multiple-valued model-checkers [7, 10] in Section 5 and conclude in Section 6.

2 Multiple-Valued Sets

2.1 Quasi-Boolean Logics

Our approach to modeling makes use of an entire family of multi-valued logics. Rather than giving a complete axiomatization for each logic, we simply give a semantics by defining conjunction, disjunction and negation on the truth values of the logic, and restrict ourselves to logics where these operations are well-defined, and satisfy commutativity, associativity etc. Such properties can be easily guaranteed if we require that the truth values of the logic form a lattice. The partial order operation $a \sqsubseteq b$ means "b is at least as true as a". Conjunction and disjunction of L are defined as $\sqcap$ and $\sqcup$ (meet and join) operations of $(\mathcal{L}, \sqsubseteq)$, respectively. In defining negation, we decided that involution ($\neg\neg a = a$) is essential, whereas the laws of non-contradiction ($\neg a \wedge a = \bot$) and

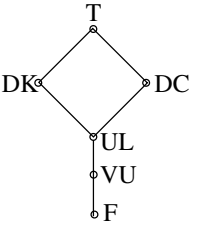


Fig. 2. A finite distributive non-quasi-boolean lattice.

excluded middle ($\neg a \vee a = \top$) are not. The resulting family of multiple-valued logics is known as quasi-boolean logics [2]. A *quasi-boolean logic* L has truth values that form a finite quasi-boolean lattice $(\mathcal{L}, \sqsubseteq)$.

Definition 1. *A finite lattice* $(\mathcal{L}, \sqsubseteq)$ *is* quasi-boolean *[2] if there exists a unary operator* $\neg$ *defined for it, with the following properties (*a, b *are elements of (*$\mathcal{L}$*,* $\sqsubseteq$*)):*

$$\neg(a \sqcap b) = \neg a \sqcup \neg b \quad \text{(De Morgan)} \qquad \neg\neg a = a \quad (\neg \text{ involution})$$
$$\neg(a \sqcup b) = \neg a \sqcap \neg b \qquad a \sqsubseteq b \Leftrightarrow \neg a \sqsupseteq \neg b \quad (\neg \text{ antimonotonic})$$

Thus, negation is defined as the $\neg$ operator of $(\mathcal{L}, \sqsubseteq)$.

In this paper, we additionally assume that $(\mathcal{L}, \sqsubseteq)$ is distributive, i.e.,

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c) \qquad a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c) \quad \text{(distributivity)}$$

Many lattices that we encounter in practice are distributive. For example, lattices in Figure 1(a),(b),(c),(e) are distributive. However, the lattice $\mathbf{5}$ in Figure 1(d) is not distributive[1]. Furthermore, not all finite distributive lattices are quasi-boolean. Consider an example in Figure 2. This lattice also represents levels of disagreement: T, DK ("don't know"), DC ("don't care"), UL ("unlikely"), VL ("very unlikely") and F. However, values UL and VL do not have counter-parts satisfying our definition of quasi-complement.

2.2 Definition of Multiple-Valued Sets

We now define the concept of multiple-valued sets over quasi-boolean lattices and give their operations. In classical set theory, a set is defined by a boolean predicate, also called a *membership function*. Typically, it is written using *set comprehension notation*: a predicate P defines the set $S = \{x \mid P(x)\}$. For instance, if $P = \lambda x \in \mathbb{Z} : 0 \leq x \leq 10$, then S is the set of all integers between 0 and 10 inclusive. If, instead of using a boolean membership function, we allow the membership function to range over a lattice, we obtain a *multiple-valued* set theory in which it is possible to make statements like "element x is more in set S than element y". We call the resulting sets *mv-sets*.

Given a lattice $L = (\mathcal{L}, \sqsubseteq)$, we define a multiple-valued set theory relative to it. For an mv-set S, and a candidate element x, we use $S(x)$ to denote the membership degree of x in S. In the classical case, this amounts to representing a set by its characteristic function.

We can use the 4-valued lattice representing disagreement (Figure 1(c)) to represent disagreement over membership of the set of years of the second millennium, as shown in

[1] We address handling of these lattices in Section 6.

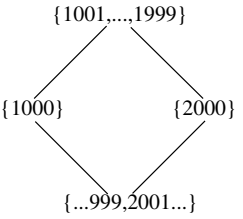


Fig. 3. The 4-valued set of years in the second millennium.

Figure 3. In order to graphically depict an mv-set, we use the structure of its underlying lattice, but replace each element with the set of elements which take on that value in the mv-set. "Pedants" insist that a millennium begins on the year ending in 1, so to them 1000 is not in the set, but 2000 is. However, "partiers" wanted to celebrate the start of the third millennium in 2000, and so claimed that 1000 is in the second millennium, and 2000 is not. Let (v_1,v_2) represent the tuple of answers that Pedants and Partiers give to a question of membership of a given element in the second millennium. Then TT is the value for all elements of $\{1001, 1002, ..., 1999\}$, TF is the value for 2000 (Pedants say "yes", whereas Partiers say "no"), FT is the value for 1000 (Pedants say "no", whereas Partiers say "yes"), and they agree that no other years belong to the second millennium.

We extend some standard set operations to the multiple-valued case by lifting the lattice meet and join operations, as follows:

$$\begin{aligned}
(S \cap_L S')(x) &\triangleq (S(x) \sqcap S'(x)) && \text{(Multiple-valued intersection)}\\
(S \cup_L S')(x) &\triangleq (S(x) \sqcup S'(x)) && \text{(Multiple-valued union)}\\
S = S' &\triangleq \forall x : (S(x) = S'(x)) && \text{(Extensional equality)}
\end{aligned}$$

It is easy to verify that if L is $\mathbf{2}$, the two-valued lattice representing classical logic (Figure 1(a)), then the membership function is a boolean predicate, and the set operations are simply standard set membership, union, and intersection. If L is the interval $[0, 1]$, ordered by the usual $\leq$ relation on real numbers, then we obtain fuzzy set theory [29]. The fuzzy-set lattice is *infinite*, but complete.

Hence forward, we will restrict ourselves to *finite and distributive* quasi-boolean lattices. These are complete, so we can extend the notion of *set complement* to the multiple-valued case, by defining it in terms of the quasi-complement of L, and denoting it with a bar:

$$\overline{S}(x) \triangleq \neg(S(x)) \quad \text{(Multiple-valued complement)}$$

We then obtain the expected properties:

$$\begin{aligned}
\overline{S \cup_L S'} &= \overline{S} \cap_L \overline{S'} && \text{(De Morgan 1)}\\
\overline{S \cap_L S'} &= \overline{S} \cup_L \overline{S'} && \text{(De Morgan 2)}\\
S \cap_L (T \cup_L V) &= (S \cap_L T) \cup_L (S \cap_L V) && \text{(Distributivity 1)}\\
S \cup_L (T \cap_L V) &= (S \cup_L T) \cap_L (S \cup_L V) && \text{(Distributivity 2)}
\end{aligned}$$

Since our sets, either classical or multiple-valued, are subsets of some U, we denote the *classical complement* of a set S as $U \setminus S$. For model-checking, U will be the set of all states of a model.

We now define some additional concepts for mv-sets that are not needed in the classical set theory:

Definition 2. *Support, Core, ℓ-cut, and ℓ-clip of a multi-valued set, S:*

$$\begin{aligned}
\sigma(S) &\triangleq \{x \mid S(x) \neq \bot\} && \text{(Support)}\\
\mathcal{C}(S) &\triangleq \{x \mid S(x) = \top\} && \text{(Core)}\\
\Uparrow_\ell(S) &\triangleq \{x \mid \ell \sqsubseteq S(x)\} && (\ell\text{-cut})\\
\Downarrow_\ell(S) &\triangleq \{x \mid S(x) \sqsubseteq \ell\} && (\ell\text{-clip})
\end{aligned}$$

All of these operations create classical sets. Support, cut, and core are standard concepts from fuzzy set theory [29]; clip is a new operation, which we shall need to prove some later result. Following the conventions of fuzzy set theory, we identify an explicit universe of discourse U, rather than use an undefinable set of all possible entities. Mv-sets can be thought of as functions from elements of U to the underlying lattice; therefore, $U = \Uparrow_\bot(S)$.

Consider the example 4-valued set of Figure 3. By inspection, we can see that $\mathcal{C}(S) = \{1001, \ldots, 1999\}$ (all the years agreed on by both groups), $\sigma(S) = \{1000, \ldots, 2000\}$ (all the years considered by either group), and $\Uparrow_{TF}(S) = \{1001, \ldots, 2000\}$, the second millennium according to the Pedants.

2.3 Multiple-Valued Relations

Now we extend the concept of degrees of membership in an mv-set to degrees of relatedness of two entities. This concept, formalized by *multiple-valued relations*, allows us to define multiple-valued transitions in a state machine models.

Definition 3. *A multiple-valued relation $\mathcal{R}$ on two sets S and T is an mv-subset of $S \times T$. The* forward image *of an mv-subset Q of S under $\mathcal{R}$ is*

$$F(\mathcal{R}, Q) \triangleq \lambda t. \bigsqcup_{s \in S} (Q(s) \sqcap \mathcal{R}(s,t))$$

and the backward image $B(\mathcal{R}, Q)$ *of an mv-subset Q of T is*

$$B(\mathcal{R}, Q) \triangleq \lambda s. \bigsqcup_{t \in T} (Q(t) \sqcap \mathcal{R}(s,t))$$

Given an mv-subset of S, its forward image under the relation $\mathcal{R}$ is an mv-subset of T; likewise, an mv-subset of T has an mv-subset of S as a backward image.

3 Efficient Representations of Multiple-Valued Sets

Having defined mv-sets, we now address an efficient implementation of the mv-set operations when the underlying lattice is finite and distributive. We represent each mv-set compactly using the cuts of the join-irreducible elements in the lattice. In this section we review the concept of join-irreducibility and show that the join-irreducible encoding allows for efficient implementation of the mv-set operations and efficient recovery of the full mv-set.

3.1 Join-Irreducibility

Definition 4. *[12] An element j in $\mathcal{L}$ is* join-irreducible *iff $j \neq \bot$ and for any x and y in $\mathcal{L}$, $j = x \sqcup y$ implies $j = x$ or $j = y$. Dually, an element m is* meet-irreducible *iff $m \neq \top$ and for any x and y, $m = x \sqcap y$ implies $m = x$ or $m = y$.*

In other words, j cannot be further decomposed into the join of other elements in the lattice, and m cannot be further decomposed into the meet of other elements in the lattice, just as prime numbers cannot be further factored into the product of other natural numbers. For example, the join-irreducible elements of the lattice **3x3** in Figure 1(e), shown in bold, are {MF, TF, FM, FT}. We denote the sets of all join-irreducible and meet-irreducible elements of a lattice L by $\mathcal{J}(L)$ and $\mathcal{M}(L)$, respectively.

We will use join-irreducibility to provide us with a kind of "prime factorization" for mv-sets. Every lattice element of a finite lattice can be defined as a subset of join-irreducible elements:

Theorem 1. *[12] Let ℓ be any element in a lattice $(\mathcal{L}, \sqsubseteq)$. Then $\ell = \bigsqcup\{\, j \in \mathcal{J}(L) \mid j \sqsubseteq \ell \,\}$.*

For example, in the lattice shown in Figure 1(e), $\mathrm{TT} = \bigsqcup\{\mathrm{MF}, \mathrm{TF}, \mathrm{FM}, \mathrm{FT}\}$, $\mathrm{TM} = \bigsqcup\{\mathrm{MF}, \mathrm{TF}, \mathrm{FM}\}$, $\mathrm{FF} = \bigsqcup \emptyset$, and so on.

Lemma 1. *[12] Let j, x, and y be elements of a distributive lattice $(\mathcal{L}, \sqsubseteq)$, with j being join-irreducible. Then $j \sqsubseteq x \sqcup y$ iff $j \sqsubseteq x$ or $j \sqsubseteq y$.*

Lemma 2. *Let ℓ be any element of a quasi-boolean lattice $(\mathcal{L}, \sqsubseteq)$. Then*

$$\{\, \neg x \mid x \in \mathcal{L}, \ell \sqsubseteq x \,\} = \{\, x \mid x \in \mathcal{L}, x \sqsubseteq \neg \ell \,\}$$

The above two lemmas describe properties of join-irreducible elements that we exploit in our representation of mv-sets. Note that Lemma 1 only applies to distributive lattices, and Lemma 2 uses the $\neg$ operator that we defined for quasi-boolean lattices. From here on we will assume all our lattices are distributive quasi-boolean lattices. All proofs are omitted in this version of the paper due to space limitations. For proofs of Lemma 2 and other non-trivial results, please refer to the extended version of this paper available at `http://www.cs.toronto.edu/~chechik/publications.html`.

Definition 5. *Let X be a subset of $\mathcal{L}$. Then*

$$\max(X) \triangleq \begin{cases} \sqcup X & \text{if } \sqcup X \in X \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \min(X) \triangleq \begin{cases} \sqcap X & \text{if } \sqcap X \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 6. *Up-sets and down-sets of lattice elements are defined as follows:*

$$\uparrow \ell \triangleq \{x \mid x \in \mathcal{L}, \ell \sqsubseteq x\} \quad \text{(Up-set of } \ell\text{)}$$
$$\downarrow \ell \triangleq \{x \mid x \in \mathcal{L}, x \sqsubseteq \ell\} \quad \text{(Down-set of } \ell\text{)}$$

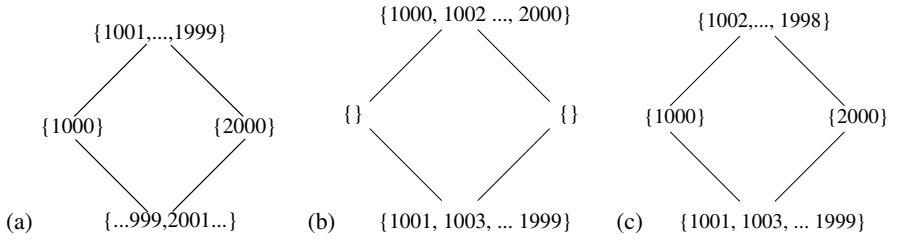


Fig. 4. (a) The 4-valued set of years in the second millennium (same as Figure 3); (b) The even numbers between 1000 and 2000, represented as a 4-valued set; (c) The mv-intersection of the two sets.

3.2 Encoding mv-sets Using j-cuts

Given that we can represent any element of a lattice using the join-irreducibles below it, we proceed to encode mv-sets using the collection of j-cuts for $j \in \mathcal{J}(L)$. We will show that the j-cuts of an mv-set contain sufficient information to rebuild the original mv-set, and prove that all the necessary operations for symbolic model-checking – *intersection*, *union*, *complement*, and *backward image* – can be easily carried out in this encoding, provided the lattice is distributive.

Our first result, which follows trivially from Theorem 1, states that the membership degree of an element in an mv-set is the join of all join-irreducibles whose cuts contain the element.

Theorem 2. *For any mv-set S with underlying lattice L,*

$$S(x) = \bigsqcup \{j \mid j \in \mathcal{J}(L), x \in \Uparrow_j(S)\}$$

We now define the operations over mv-sets in terms of operations over their j-cuts.

Theorem 3. *The ℓ-cut of a multiple-valued intersection is the intersection of the ℓ-cuts of the individual mv-sets. For $j \in \mathcal{J}(L)$, the j-cut of a multiple-valued union is the union of the j-cuts.*

$$\forall \ell \in \mathcal{L} : \Uparrow_\ell (S \cap_L T) = \Uparrow_\ell (S) \cap \Uparrow_\ell (T) \quad \text{(Cut-intersection)}$$
$$\forall j \in \mathcal{J}(L) : \Uparrow_j (S \cup_L T) = \Uparrow_j (S) \cup \Uparrow_j (T) \quad \text{(Cut-union)}$$

As an example, consider the two mv-sets shown in Figure 4, parts (a) and (b). The TF-cut of their mv-intersection should be the same as the (classical) intersection of the TF-cuts. By inspection, the TF-cut of part (a) is $\{1000, \ldots, 1999\}$, and that of part (b) is $\{1000, 1002, \ldots, 2000\}$, with the intersection $\{1000, 1002, \ldots, 1998\}$. The TF-cut of the mv-intersection of (a) and (b), shown in part (c), is $\{1000, 1002, \ldots, 1998\}$, which is indeed the same as the intersection of the individual TF-cuts of these mv-sets. It is easy to check that the same holds for other j-cuts. Theorem 3 is the fundamental result that we will use throughout the remainder of this paper.

Formulation of complementation uses two observations. First is a trivial corollary of Lemma 2, indicating that multiple-valued complement turns each j-cut into a $\neg j$-clip:

Lemma 3. *Each j-cut of an mv-set, S, is the $\neg j$-clip of the complement of S:*

$$\Uparrow_j (S) = \Downarrow_{\neg j} (\overline{S})$$

Second, every up-set of a join-irreducible, j, is the complement (in the lattice) of the down-set of some meet-irreducible m:

Lemma 4. *In all distributive lattices $L = (\mathcal{L}, \sqsubseteq)$ there exists a bijection $f : \mathcal{J}(L) \to \mathcal{M}(L)$ such that, for all join-irreducible j, $\uparrow j = \mathcal{L} \setminus \downarrow f(j)$.*

It follows that for an mv-subset S of some universe U, $\Uparrow_j(S) = U \setminus \Downarrow_{f(j)}(S)$. We use this to construct the j-cut of a multiple-valued complement of an mv-set:

Theorem 4. *Let S be an mv-subset of some U, based on a lattice $L = (\mathcal{L}, \sqsubseteq)$, and let $j \in \mathcal{J}(L)$. The j-cut of the multiple-valued complement of S is the classical complement (in U) of the $f^{-1}(\neg j)$ cut of S:* $\Uparrow_j(\overline{S}) = U \setminus \Uparrow_{f^{-1}(\neg j)}(S)$.

Finally, we demonstrate that the j-cuts of the backward image of an mv-set under a multiple-valued relation are simply the backward images of the j-cuts of the relation:

Theorem 5. *Let $\mathcal{R}$ be a multiple-valued relation on S and T, and Q be an mv-subset of T, with the same underlying lattice. Then*

$$\Uparrow_j(B(\mathcal{R}, Q)) = \{s \mid (\exists t \in \Uparrow_j(Q) : (s,t) \in \Uparrow_j(\mathcal{R}))\} \quad \text{(Cut-reachability)}$$

These results allow us to represent each mv-set compactly as a family of its j-cuts. Theorem 2 guarantees that this representation of mv-sets preserves all information; it also tells us how to recover mv-set membership from the j-cuts.

The advantage of this representation is in the computation of multiple-valued intersections and unions. If mv-sets were represented by families of pieces, i.e., indexed by all lattice values, then intersection and union would both take as many as $O(|\mathcal{L}|^2)$ classical set operations. In contrast, the cut-intersection and the cut-union theorems imply that our representation requires just $O(|\mathcal{J}(L)|)$ classical set operations.

4 Multiple-Valued Model-Checking

In this section we review multiple-valued Kripke structures, which we call χKripke structures, and multiple-valued CTL (χCTL). We then outline a model-checking algorithm based on cuts of join-irreducible lattice values.

4.1 Semantics

A state machine M is a *χKripke structure* if $M = (S, S_0, R, I, A, L)$, where:

- L is a quasi-boolean logic represented by a lattice $(\mathcal{L}, \sqsubseteq)$. L provides the lattice for all mv-sets in the model.
- A is a (finite) set of atomic propositions, otherwise referred to as variables. For simplicity, we assume that all variables are of the same type, with values ranging over the values of the logic L.
- S is a (finite) set[2] of states; each state is identified by a unique (within M) label s.
- $S_0 \subseteq S$ is the non-empty mv-set of initial states.

[2] S is a classical set, but it can also be considered an mv-set where $\forall s : S(s) \in \{\top, \bot\}$.

$$
\begin{aligned}
[\![a]\!] &\triangleq (I(s))(a) \\
[\![\varphi \wedge \psi]\!] &\triangleq [\![\varphi]\!] \cap_L [\![\psi]\!] \\
[\![\neg\varphi]\!] &\triangleq \overline{[\![\varphi]\!]} \\
[\![\varphi \vee \psi]\!] &\triangleq [\![\varphi]\!] \cup_L [\![\psi]\!] \\
[\![EX\varphi]\!] &\triangleq B(R, [\![\varphi]\!]) \\
[\![AX\varphi]\!] &\triangleq \overline{[\![EX\neg\varphi]\!]} \\
[\![E[\varphi U\psi]]\!] &\triangleq [\![\psi]\!] \cup_L ([\![\varphi]\!] \cap_L [\![EXE[\varphi U\psi]]\!]) \\
[\![A[\varphi U\psi]]\!] &\triangleq [\![\psi]\!] \cup_L ([\![\varphi]\!] \cap_L [\![AXA[\varphi U\psi]]\!] \cap_L [\![EXA[\varphi U\psi]]\!]
\end{aligned}
$$

Fig. 5. Semantics of χCTL operators.

- R is the multiple-valued transition relation. For each state s, there is at least one (non-false) transition $(s,t) \in \sigma(R)$, extending the classical notion of Kripke structures.
- I is a labeling function that maps states in S to mv-subsets of A. Intuitively, for any symbol $a \in A$, $(I(s))(a) = \ell$ can be considered equivalent to the variable a having value ℓ in state s.

This description of χKripke structures is that of our previous work [7], influenced by notation used for fuzzy transition systems [26].

Now we review the semantics of χCTL operators on a χKripke structure M over a quasi-boolean logic L. In extending the CTL operators, we want to ensure that the expected CTL properties [11] are still preserved[3]. The semantics of χCTL operators is given in Figure 5. We use the double-brace notation, adopted from denotational semantics, and write $[\![\varphi]\!]$ to denote the mv-set of states representing a degree to which φ holds. The semantics of the EX operator comes from our previous definition of backward image (Definition 3). The definitions of AU, EU and AX are given using the properties of CTL operators [11].

4.2 Multiple-Valued Model-Checking Algorithm

Our multiple-valued model-checking algorithm is shown in Figure 6. In this algorithm, we encode state mv-sets and the transition relation for the model by their j-cuts. For mv-sets, we use the following shorthand:

$$
\begin{aligned}
\llbracket\varphi\rrbracket &\triangleq \{\Uparrow_j ([\![\varphi]\!]) \mid j \in \mathcal{J}(L)\} && \text{(All } j\text{-cuts)} \\
\llbracket\varphi\rrbracket\, j &\triangleq \Uparrow_j ([\![\varphi]\!]) && \text{(Single } j\text{-cut)}
\end{aligned}
$$

The structure of this algorithm is based on our earlier algorithm in [10]. The routine $\mathsf{Check}(p)$ is the entry point, where p is the χCTL formula to be checked. It recurses through p, associating each sub-formula φ with j-cuts ($\llbracket\varphi\rrbracket$) of state mv-sets. The handling of $\wedge$ and $\vee$ follows from Theorem 3. The $\neg$ operator is handled by the negation function, based on Theorem 4. The next-state operators EX and AX are handled by the pred function, based on Theorem 5; pred computes EX, and $AX(\varphi)$ is obtained

[3] Note that the AU fixpoint is somewhat unusual because it includes an additional conjunct, $EXA[\varphi U\psi]$. This term preserves a "strong until" semantics for states that have no outgoing $\top$ transitions in non-Kripke structures [6].

```
Check(p)
    case p ∈ A:          return ⟦p⟧
    case p = ¬φ:         return negation(⟦φ⟧)
    case p = φ ∧ ψ:      ∀j ∈ 𝒥(L): ⟦result⟧j := ⟦φ⟧j ∩ ⟦ψ⟧j
                         return ⟦result⟧
    case p = φ ∨ ψ:      ∀j ∈ 𝒥(L): ⟦result⟧j := ⟦φ⟧j ∪ ⟦ψ⟧j
                         return ⟦result⟧
    case p = AX φ:       return negation(pred(negation(⟦φ⟧)))
    case p = EX φ:       return pred(⟦φ⟧)
    case p = A[φUψ]:     return QUntil(A, ⟦φ⟧, ⟦ψ⟧)
    case p = E[φUψ]:     return QUntil(E, ⟦φ⟧, ⟦ψ⟧)

negation(⟦φ⟧)
    ∀j ∈ 𝒥(L): ⟦n⟧j = S \ (⟦φ⟧ neg[j])
    return ⟦n⟧

QUntil(Q, ⟦φ⟧, ⟦ψ⟧)
    ⟦QU⟧ := ⟦ψ⟧
    repeat
        ⟦EXterm⟧ := pred(⟦QU⟧)
        if Q = A
          ⟦AXterm⟧ := negation(pred(negation(⟦QU⟧)))
        else          // Q = E
          ⟦AXterm⟧ := S
        ∀j ∈ 𝒥(L): ⟦QU⟧j := ⟦ψ⟧j ∪ (⟦φ⟧j ∩ ⟦EXterm⟧j ∩ ⟦AXterm⟧j)
    until ⟦QU⟧ converges
    return ⟦QU⟧

pred(⟦φ⟧)
    ∀j ∈ 𝒥(L): ⟦result⟧j := { s | ∃t: t ∈ ⟦φ⟧j ∧ (s,t) ∈ ⟦R⟧j }
    return ⟦result⟧
```

Fig. 6. The model-checking algorithm.

from $\neg EX(\neg\varphi)$. Finally, EU and AU are fixpoints computed iteratively by the QUntil function using the equations given in Figure 5.

In the negation function, a precomputed neg table is used for quick lookup of $f^{-1}(\neg j)$, for any join-irreducible element j, as required by Theorem 4. It is precomputed from the constructive proof of Lemma 4 where f is specified: first find f using $f(j) = \max(\mathcal{L} \setminus \uparrow j)$, then fill in the neg table using $\mathsf{neg}[\neg f(j)] = j$.

Theorem 6. *Given a χKripke structure and a χCTL formula p to model-check,* Check(p) *in the algorithm in Figure 6 returns $\llbracket p \rrbracket$, i.e., $\{\Uparrow_j (\llbracket p \rrbracket) \mid j \in \mathcal{J}(L)\}$.*

4.3 Running Times

To determine the running time of this algorithm, we start by analyzing individual operations: union, intersection, complement and backward image. We use a Multiple-Valued Binary Terminal Decision Diagram (MBTDD) [13] to represent each j-cut, denoting

Operation	MBTDD-based	MDD-based
$\cup$ and $\cap$	$\lvert\mathcal{J}(L)\rvert \times \text{MBT}(n)^2$	$\text{MDD}(n)^2$
complement	$\lvert\mathcal{J}(L)\rvert \times \text{MBT}(n)$	$\text{MDD}(n)$
back. image	$\lvert\mathcal{J}(L)\rvert \times (\text{MBT}(n) + \text{MBT}(n) \times \text{MBT}(2n))$	$\text{MDD}(n) + \text{MDD}(n) \times \text{MDD}(2n)$

Table 1. Running times for operations on mv-sets using MBTDDs with j-cut encoding vs. MDDs.

the size of a MBTDD for n variables as $\text{MBT}(n)$. The running time of each operation on a Kripke structure is given in the middle column of Table 1, where $n = |A|$. For example, Theorem 3 allows us to do a pairwise union of each of $\mathcal{J}(L)$ j-cuts, taking $|\mathcal{J}(L)| \times \text{MBT}(n)^2$ operations. Computation of backward image for each of $\mathcal{J}(L)$ MBTDDs consists of a disjunction ($\text{MBT}(n)$ operations) of conjunctions (Definition 3) between the formula ($\text{MBT}(n)$) and the transition relation ($\text{MBT}(2n)$ operations).

Next, note that the computation of QUntil in the model-checking algorithm in Figure 6 dominates the running time, with each state changing its position in $[\![QU]\!]$ at most h times, where h is the height of the lattice $(\mathcal{L}, \sqsubseteq)$, so the maximum number of iterations of the repeat-until loop is $|S| \times h$. In each iteration, there are two pred operations, two negation operations, and three unions and intersections. In the worst case, $\text{MBT}(n) = O(|\mathcal{L}|^{|A|-1})$, so the running time of QUntil is $O(|S| \times h \times |\mathcal{J}(L)| \times |\mathcal{L}|^{3|A|-2})$.

Theorem 7. *Given a χKripke structure (S, S_0, R, I, A, L) and a χCTL formula p, the algorithm in Figure 6 terminates in $O(|p| \times |S| \times h \times |\mathcal{J}(L)| \times |\mathcal{L}|^{3|A|-2})$ time, where h is the height of the lattice.*

5 Evaluation

In this section, we compare the running time of the algorithm in this paper with those of our earlier algorithms [7, 10]. Since the structure of the algorithms does not change, the difference in performance is in the cost of the four operations on mv-sets: union, intersection, complement and backward reachability.

In our first multiple-valued model-checker [10], we encoded each lattice value x as a sequence of $|\mathcal{L}|$ bits, where bit x was on if $x \in L$, and off otherwise. In this paper, we encode a lattice value x using a bit for each $j \in \mathcal{J}(L)$, which is on only when $j \sqsubseteq x$. It is always better than the encoding in [10], and thus all operations are faster.

The righthand column of Table 1 lists the running times of the four operations for the MDD-based representation of mv-sets that we used in [7]. We refer to the size of an MDD on n variables as $\text{MDD}(n)$. Running times for MDD operations are similar to those for MBTDD operations; however, for a given mv-set, the MBTDDs representing the j-cuts are usually smaller than an MDD representing the whole mv-set.

We can express the performance improvement of the model-checker in this paper in terms of the ratio between the average size of an MDD and that of an MBTDD for a given set of n variables, which we call β. The performance improvements for the operations in Table 1 are $|\mathcal{J}(L)|/\beta^2$ for unions and intersections, and at most $|\mathcal{J}(L)|/\beta$ for complements and backward images.

We can then assess the overall improvement by estimating β for different types of problem. In the worst case, an MDD and an MBTDD can have the same number of non-terminal nodes, making β roughly 1; then the algorithm of this paper actually results in a slowdown. However, our experience indicates that as problems get larger, β approaches $|\mathcal{L}|$. This improvement is because there are typically more common sub-expressions in boolean-terminal diagrams than in their multi-terminal counterparts, and thus more possibility for sharing. Preliminary experimental results support this analysis. Results in Table 2 were produced by randomly generating a pool of sample functions of a given number of variables (3 to 6), and then creating the single MDD and the collection of $|\mathcal{J}(L)|$ MBTDDs representing the function, and taking the average sizes of the individual decision diagrams over the sample. We give this issue a fuller treatment elsewhere [13]. Thus, we believe that there is a significant speedup when using the model-checking algorithm described in this paper. For unions and intersections this speedup can range from $|\mathcal{L}|$ to as much as $|\mathcal{L}|^2 / \log_2 |\mathcal{L}|$, depending on the shape of the lattice.

Number of variables	Mean MDD size	Mean MBTDD size	Ratio (β)
3	99	66	1.5
4	828	218	3.8
5	7,389	949	7.8
6	66,434	7,510	8.8

Table 2. Empirical results, for randomly-generated functions, with $L = \mathbf{3x3}$.

6 Conclusion

The challenge for efficient model-checking over multiple-valued logics is to find a compact encoding for mv-sets of states that supports fast computation of mv-set union, intersection, complement and backwards reachability. In this paper we described an elegant solution based on the use of join-irreducible elements of a lattice to provide a factorization of the mv-sets. We demonstrated that this representation supports the necessary operations, and showed that the result is a significantly faster model-checking algorithm. The exact speedup depends on the size of the join-irreducible set. For example, when we multiply lattices, as we often do when reasoning about multiple points of view (with or without abstraction) [16], the size of the resulting join-irreducible set grows as the *sum* of sizes of join-irreducible sets of the original lattices.

Furthermore, after casting the multiple-valued model-checking problem in terms of efficient operations on MBTDDs, we can further improve the running times by reusing existing symbolic model-checking technology. In particular, we can further optimize the MBTDD negation operation (to $|\mathcal{J}(L)|$) by using complement arcs, as in Somenzi's CUDD library [28].

The use of join-irreducibility to optimize our algorithms introduces an important restriction on the logics that we can use in the model-checker. We originally chose to restrict ourselves to logics whose values form a quasi-boolean lattice, as these logics

behave similarly to classical logic, but do not enforce the law of excluded middle and the law of non-contradiction [10]. The optimization described in this paper restricts us further to logics whose lattices are distributive. Our motivation for developing a multiple-valued model-checker was to verify properties of models that contain uncertainty and/or disagreement [16]. For this we typically use the logics $\mathbf{3}$ and $\mathbf{2x2}$ (shown in Figure 1), and their products, all of which are distributive. However, when the logic is represented by a non-distributive lattice, e.g., $\mathbf{5}$, our model-checker automatically switches to the MDD-based algorithm, described in [7]. Thus, in the case where the optimization is possible, it is applied, and in the other cases, the more general algorithm is used.

Optimizing model-checking on multi-valued logics through the use of join-irreducible elements is not restricted to branching-time logic or to symbolic model-checking. In particular, in our recent work [8] we showed that when all elements of the lattice are join-irreducible, then multi-valued model-checking reduces to several queries to a classical model-checker, and built a multi-valued LTL model-checker on top of SPIN.

Acknowledgments

We would like to thank Alasdair Urquhart for suggesting that distributive lattices can be represented by their join-irreducible elements. This work was financially supported by NSERC and CITO.

References

1. N.D. Belnap. "A Useful Four-Valued Logic". In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.
2. L. Bolc and P. Borowik. *Many-Valued Logics*. Springer-Verlag, 1992.
3. G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 274–287, 1999.
4. G. Bruns and P. Godefroid. "Generalized Model Checking: Reasoning about Partial State Spaces". In *Proceedings of CONCUR'00*, volume 877 of *LNCS*, pages 168–182, August 2000.
5. R. E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". *Computing Surveys*, 24(3):293–318, September 1992.
6. T. Bultan, R. Gerber, and C. League. "Composite Model Checking: Verification with Type-Specific Symbolic Representations". *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.
7. M. Chechik, B. Devereux, and S. Easterbrook. "Implementing a Multi-Valued Symbolic Model-Checker". In *Proceedings of TACAS'01*, volume 2031 of *LNCS*, pages 404–419. Springer, April 2001.
8. M. Chechik, B. Devereux, and A. Gurfinkel. "Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN". In *Proceedings of the 8th SPIN Workshop on Model Checking Software*, volume 2057 of *LNCS*, pages 16–36, May 2001.
9. M. Chechik and W. Ding. "Lightweight Reasoning about Program Correctness". CSRG Technical Report 396, University of Toronto, March 2000.
10. M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proceedings of FME'01*, volume 2021 of *LNCS*, pages 72–98. Springer, March 2001.

11. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
12. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
13. B. Devereux. Symbolic representation and reasoning over state-based models with multiplicities. Master's thesis, University of Toronto, Department of Computer Science, June 2001.
14. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.
15. J.M. Dunn. "A Comparative Study of Various Model-Theoretic Treatments of Negation: A History of Formal Negation". In Dov Gabbay and Heinrich Wansing, editors, *What is Negation*. Kluwer Academic Publishers, 1999.
16. S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, May 2001.
17. Melvin Fitting. "Many-Valued Modal Logics". *Fundamenta Informaticae*, 15(3-4):335–350, 1991.
18. Melvin Fitting. "Many-Valued Modal Logics II". *Fundamenta Informaticae*, 17:55–73, 1992.
19. Brian R. Gaines. "Logical Foundations for Database Systems". *International Journal of Man-Machine Studies*, 11(4):481–500, 1979.
20. Matthew Ginsberg. "Multi-valued logic". In M. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 251–255. Morgan-Kaufmann Pub., 1987.
21. Reiner Hähnle. *Automated Deduction in Multiple-Valued Logics*, volume 10 of *International Series of Monographs on Computer Science*. Oxford University Press, 1994.
22. S. Hazelhurst. *Compositional Model Checking of Partially Ordered State Spaces*. PhD thesis, Department of Computer Science, University of British Columbia, 1996.
23. J. Łukasiewicz. *Selected Works*. North-Holland, Amsterdam, Holland, 1970.
24. R. S. Michalski. "Variable-Valued Logic and its Applications to Pattern Recognition and Machine Learning". In D. C. Rine, editor, *Computer Science and Multiple-Valued Logic: Theory and Applications*, pages 506–534. North-Holland, Amsterdam, 1977.
25. M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In *Proceedings of 26th Annual ACM Symposium on Principles of Programming Languages*, 1999.
26. E. Santos. "Regular Fuzzy Expressions". In Madan M. Gupta, George N. Saridis, and Brian R. Gaines, editors, *Fuzzy Automata and Decision Processes*, pages 169–176, New York, 1977. North–Holland.
27. Viorica Sofronie-Stokkermans. Automated theorem proving by resolution for finitely-valued logics based on distributive lattices with operators. *Multiple-Valued Logic*, 2000.
28. Fabio Somenzi. "Binary Decision Diagrams". In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.
29. L.A. Zadeh. "Fuzzy Sets". In R. R. Yager, S. Ovchinnikov, R. M. Tong, and H. T. Nguyen, editors, *Fuzzy Sets and Applications: Selected Papers by L.A. Zadeh*, pages 29–44, New York, 1987. John Wiley & Sons, Inc.

Divide and Compose: SCC Refinement for Language Emptiness*

Chao Wang[1], Roderick Bloem[1], Gary D. Hachtel[1], Kavita Ravi[2], and Fabio Somenzi[1]

[1] University of Colorado at Boulder
{wangc,hachtel,Fabio}@Colorado.EDU
[2] Cadence Design Systems
kravi@cadence.com

Abstract. We propose a refinement approach to symbolic SCC analysis, which performs large parts of the computation on abstracted systems, and on small subsets of the state space. For language-emptiness checking, it quickly discards uninteresting parts of the state space; for the remaining states, it adapts the model checking computation to the strength of the SCCs at hand.
We present a general framework for SCC refinement, which uses a compositional approach to generate and refine overapproximations. We show that our algorithm significantly outperforms the one of Emerson and Lei.

1 Introduction

Checking language emptiness of a Büchi automaton is a core procedure in LTL [12,17] and fair-CTL model checking [13], and in language-containment based verification approaches [11]. The classical algorithm by Emerson and Lei [7] used in symbolic model checkers is based on the computation of an *SCC hull* [15], while Xie and Beerel [18] and Bloem, Gabow, and Somenzi [1] use SCC decomposition to decide language emptiness.

Although the Lockstep algorithm of [1] has a better complexity than the one of Emerson and Lei ($n \log n$ versus n^2), the comparison presented in [15] shows that the theoretical advantage seldom translates in shorter CPU times. We present an algorithm that uses abstractions to compute an SCC decomposition of the system by refinement. It combines this with known language emptiness approaches to form a hybrid algorithm that shares the good theoretical characteristics of Lockstep, while outperforming the most popular SCC-hull methods, including the one of Emerson and Lei. This *Divide and Compose* algorithm, called *D'n'C*, has the following features:

1. It is compositional, performing as much work as possible on abstracted systems.

* This work was supported in part by SRC contract 98-DJ-620 and NSF grant CCR-99-71195.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 456–471, 2001.
© Springer-Verlag Berlin Heidelberg 2001

2. It considers only parts of the state space at any time.
3. It uses the *strength* of a given set of SCCs in a given system to decide the proper model checking algorithm.

We assume that the model is the conjunction of a large number of modules, including the Büchi automaton for the property. Our approach exploits property locality [11] by first performing an SCC decomposition on an abstraction obtained by composing a small number of modules. Then it automatically refines the system by composing the current abstraction with one of the previously omitted modules, which enables it to refine the SCC decomposition in turn.

At any stage, if an SCC of an abstracted system does not contain a fair cycle, then we can safely discard that part of the state space, which means we do not have to consider it in a more refined system. Because each SCC of a system is contained in an SCC of a more abstract system, and because we do not have to consider all SCCs, we can often drastically limit the potential space in which a fair cycle can lie. This allows us to make very efficient use of don't care conditions.

The *strength* of a Büchi automaton [10,2] is an important factor in symbolic model checking. Specialized model checking algorithms for weak and especially terminal automata outperform the general language emptiness algorithm of Emerson and Lei: $\mathsf{EF}\,\mathsf{EG}\,\mathsf{fair}$ can be used for weak systems and $\mathsf{EF}\,\mathsf{fair}$ can be used for terminal ones. For strong automata, however, a general fair cycle detection algorithm must be used.

The classification of [2] determines which model checking approach to use, based on the strength of the Büchi *automaton*. This may be inefficient because a Büchi automaton that contains one strong SCC and several weak ones is classified as strong. Our approach considers the strength of each *individual SCC* to decide which model checking procedure to use. Furthermore, it uses *strength reduction*: the fact that the composition of a strong SCC with a Kripke structure may contain weak SCCs. Our approach analyzes SCCs as they are computed to take maximal advantage of their weakness.

The rest of this paper is organized as follows. Section 2 reviews the background material. Section 4 discusses the algorithmic framework for SCC refinement, while Section 5 deals with the compositional approach. Section 6 discusses the implementation details, and presents preliminary experimental results. These results show that the algorithm often achieves substantial savings in memory and CPU time. Section 7 summarizes the contributions of the paper and outlines promising future work.

2 Preliminaries

A *Strongly-Connected Component (SCC) C* of a graph G is a maximal set of nodes such that there is a path between any two nodes in C. An SCC that consists of just one node without a self loop is called *trivial*. An *SCC-closed set* of G is the union of a collection of SCCs. The set of SCCs of G is denoted by

SCCs(G) and is a partition of the vertices of G. A partition π_1 of V is a *refinement* of another partition π_2 of V if, for every $B_1 \in \pi_1$, there exists $B_2 \in \pi_2$ such that $B_1 \subseteq B_2$.

We model the systems we consider as generalized Büchi automata.

Definition 1. *A* (labeled, generalized) Büchi automaton *is a six-tuple*

$$\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle \ ,$$

where Q is the finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $T \subseteq Q \times Q$ is the transition relation, $\mathcal{F} \subseteq 2^Q$ is the set of acceptance conditions, A is the finite set of atomic propositions, and $\Lambda : Q \to 2^A$ is the labeling function.

Note that we have defined automata with labels on the states, not the edges. A *run* of $\mathcal{A}$ is an infinite sequence $\rho = \rho_0, \rho_1, \ldots$ over Q, such that $\rho_0 \in Q_0$, and for all $i \geq 0$, $(\rho_i, \rho_{i+1}) \in T$. A run ρ is *accepting* (or *fair*) if, for each $F_i \in \mathcal{F}$, there exists $q_j \in F_i$ that appears infinitely often in ρ.

The automaton accepts an infinite word $\sigma = \sigma_0, \sigma_1, \ldots$ in A^ω if there exists an accepting run ρ such that, for all $i \geq 0$, $\sigma_i \in \Lambda(\rho_i)$. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the subset of A^ω accepted by $\mathcal{A}$. The language of $\mathcal{A}$ is nonempty iff $\mathcal{A}$ contains an *accepting cycle*: a cycle that is reachable from an initial state and intersects all acceptance conditions. An automaton contains an accepting cycle iff it contains an *accepting SCC*: a reachable SCC that intersects all accepting sets.

A state q is *complete* if for every $a \in A$ there is a successor q' of q such that $a \in \Lambda(q')$. A set of states, or an automaton, is complete if all of its states are.

As is usual in symbolic model checking, given a transition relation T, we define $\mathrm{img}_T(S) = \{q' \mid \exists q \in S : (q, q') \in T\}$, and $\mathrm{pre}_T(S) = \{q \mid \exists q' \in S : (q, q') \in T\}$. A *step* is the computation of either $\mathrm{img}_T(S)$ or $\mathrm{pre}_T(S)$.

We assume that all automata are defined over the same state space and agree on the state labels. Communication then proceeds through the common state space, and composition is characterized by the intersection of the transition relations: The composition $\mathcal{A}_1 \times \mathcal{A}_2 = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$ of two Büchi automata $\mathcal{A}_1 = \langle Q, Q_{01}, T_1, \mathcal{F}_1, A, \Lambda \rangle$ and $\mathcal{A}_2 = \langle Q, Q_{02}, T_2, \mathcal{F}_2, A, \Lambda \rangle$ is defined by $Q_0 = Q_{01} \cap Q_{02}$, $T = T_1 \cap T_2$, and $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$. Hence, composing two automata restricts the transition relation and results in the intersection of the two languages.

If the states of a Büchi automaton are the valuations of r binary state variables, then language emptiness can be checked by a symbolic algorithm in $O(|Q| \log |Q|) = O(r2^r)$ steps [1]. We can improve this bound if we know that the automaton does not control some of the state variables. Specifically, let V a finite set (of binary variables), and let $\mathcal{A} = \langle Q, Q_0, T, \mathcal{F}, A, \Lambda \rangle$ be an automaton such that $Q = 2^V$. Given $q \in Q$ and $v \in V$, let $q^v \in Q$ be the state given by $q \cup \{v\}$ if $v \notin q$ and $q \setminus \{v\}$ otherwise. Then $\mathcal{A}$ *controls* v if there exist $q_1, q_2 \in Q$ such that $(q_1, q_2) \in T$ and $(q_1, q_2^v) \notin T$. Let $V_\mathcal{A}$ the set of variables controlled by $\mathcal{A}$. We define the *effective* number of states of $\mathcal{A}$ as $\eta_\mathcal{A} = 2^{|V_\mathcal{A}|}$. One can easily show that language emptiness for $\mathcal{A}$ can be checked in $O(\eta_\mathcal{A} \log \eta_\mathcal{A})$ steps by the algorithm of [1].

We say that $\mathcal{A} \leq \mathcal{A}'$ if $Q = Q'$, $Q_0 \subseteq Q'_0$, $T \subseteq T'$, $\mathcal{F} = \mathcal{F}'$, and $\Lambda = \Lambda'$. This (rather strong) condition induces a partial order on automata, such that $\mathcal{A} \leq \mathcal{A}'$ implies $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. If $\mathcal{A} \leq \mathcal{A}'$, we say that $\mathcal{A}'$ is an *overapproximation* of $\mathcal{A}$.

Let $C \subseteq Q$ be an SCC of $\mathcal{A}$. We define the *strength* of C as follows (cf. [10,2]).

- C is *weak* if all cycles contained within it are accepting.
- C is *terminal* if it is weak, complete, and there is no edge from a node in C to any non-terminal SCC. Terminality implies acceptance of all runs reaching C.
- C is *strong* if it is not weak.

Note that the definition of weakness is more relaxed than that of [10,2], while still allowing us to use a linear-time symbolic model checking algorithm.

We can order SCCs according to their strength: Strong SCCs are stronger than weak ones, and weak SCCs that are not terminal are stronger than terminal SCCs. In general, the weaker an SCC is, the easier it is to decide language emptiness: An automaton containing a weak (terminal) SCC S has a nonempty language if $\mathsf{EF}\,\mathsf{EG}\,S \cap Q_0 \neq \emptyset$ ($\mathsf{EF}\,S \cap Q_0 \neq \emptyset$) holds.[1] The strength of an SCC-closed set or of an automaton is the maximum strength of its accepting SCCs.

3 Don't Care Conditions

Image and preimage usually account for most of the CPU time in symbolic, BDD-based model checking [4,13]. Therefore, it is important to minimize the sizes of the representations of both the transition relation, and the argument to the (pre-)image computation. The size of a BDD is not directly related to the size of the set it represents. If we need not represent a set exactly, but can instead determine an interval in which it may lie, we can use known techniques [6,16] to find a set within this interval with a small BDD representation.

Often, we are only interested in the results as far as they lie within a *care set* K (or outside a *don't care set* $\overline{K}$). Since the satisfaction of a property is only relevant within the set of reachable states R, we can use R as a care set to add or delete edges that emanate from unreachable states. By doing this, the image of a set that is contained within R remains the same. Likewise, the part of the preimage of a set S that intersects R remains the same, even if we add unreachable states to S. This use of R as care set depends on the fact that no edges from reachable to unreachable states are added.

The algorithm of Section 4 manipulates small portions of the state space, defined by SCC-closed sets. This allows us to use care sets that are often much smaller than the set of reachable states, and thus to increase the chance of finding small BDDs. We cannot use the approach outlined for the reachable states, since there may be edges from an SCC-closed set to other states. We show here that in order to use arbitrary sets as care sets in image computation, a 'safety zone'

[1] $\mathsf{EF}\,S$ is the subset of Q from which S is reachable, while $\mathsf{EG}\,S$ is the set of states in Q that are the origins of infinite paths entirely contained in S.

consisting of the preimage of the care set needs to be kept; similarly for preimage computation.

Theorem 1. *Let Q be a set of states and let $T \subseteq Q \times Q$ be a transition relation. Let $K \subseteq Q$ be a care set, $B \subseteq K$ a set of states,*

$$T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K}), \text{ and}$$
$$B \subseteq B' \subseteq B \cup \overline{\mathrm{pre}_{T'}(K)} \ .$$

Then, $\mathrm{img}_{T'}(B') \cap K = \mathrm{img}_T(B) \cap K$.

Proof. First, suppose that $q' \in \mathrm{img}_{T'}(B') \cap K$, and let $q \in B'$ be such that $q' \in \mathrm{img}_{T'}(\{q\}) \cap K$. Since $q' \in K$, we have $q \in \mathrm{pre}_{T'}(K)$. Hence, $q \in B'$ implies $q \in B$, and $q, q' \in K$, which means that $q' \in \mathrm{img}_T(\{q\}) \cap K$. Finally, $q \in B$ implies $q' \in \mathrm{img}_T(B) \cap K$. Conversely, suppose that $q' \in \mathrm{img}_T(B) \cap K$, and let $q \in B$ be such that $q' \in \mathrm{img}_T(\{q\}) \cap K$. Now $q, q' \in K$, and hence $q' \in \mathrm{img}_{T'}(\{q\}) \cap K$, and since $q \in B'$, $q' \in \mathrm{img}_{T'}(B') \cap K$. □

Hence, we can choose T' and B' to be sets within the given interval that have a small representation,[2] and use them instead of T and B. Through symmetry, we can prove the following theorem.

Theorem 2. *Let Q be a set of states and let $T \subseteq Q \times Q$. Let $K \subseteq Q$, $B \subseteq K$, $T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K})$, and $B \subseteq B' \subseteq B \cup \mathrm{img}_{T'}(K)$. Then,* $\mathrm{pre}_{T'}(B') \cap K = \mathrm{pre}_T(B) \cap K$.

Edges are added to and from the set $\overline{K}$ (outside K) and the safety zone for (pre-)image computation excludes the immediate (successors) predecessors of K. Note that the validity of the aforementioned use of the reachable set as care set follows as a corollary of these two theorems.

4 SCC Refinement

This paper describes a hybrid algorithm for fair cycle detection that combines SCC refinement with more classical algorithms—like the one of Emerson and Lei [7]—that compute an SCC hull [15]. We shall here describe the general framework, and in later sections we shall discuss the implementation choices that we have made. The refinement processes uses a set of overapproximations of the system. We separate the generation of the overapproximations from their use; the method presented here works for any set of overapproximations.

[2] The actual use of don't care information for the transition relation is asymmetric: We add edges out of don't care states, but not into them. This is due to the useful minimizations that can be performed on a partitioned transition relation.

4.1 Refinement

The core of our refinement approach is expressed by the following theorem.

Theorem 3. *Let $\mathcal{A}$ be a Büchi automaton, and let $\mathcal{A}_1, \ldots, \mathcal{A}_n \geq \mathcal{A}$ be overapproximations. Then,* $\mathrm{SCCs}(\mathcal{A})$ *is a refinement of*

$$\{C_1 \cap \cdots \cap C_n \mid C_1 \in \mathrm{SCCs}(\mathcal{A}_1), \ldots, C_n \in \mathrm{SCCs}(\mathcal{A}_n)\} \setminus \emptyset \ .$$

In particular, the set of SCCs of $\mathcal{A}$ is a refinement of the set of SCCs of any overapproximation $\mathcal{A}'$ of $\mathcal{A}$. Hence, an SCC of $\mathcal{A}'$ is an SCC-closed set (but not necessarily an SCC) of $\mathcal{A}$. This theorem allows us to gradually refine the set of SCCs until we arrive at the SCCs of $\mathcal{A}$.

One of the benefits of this approach is that we can often decide early that an SCC-closed set does not contain an accepting cycle. This allows us to trim the state space before considering the exact system, keeping around only 'suspect' SCCs.

Observation 4 *Let C be an SCC-closed set of $\mathcal{A}$. If $C \cap F_i = \emptyset$ for any $F_i \in \mathcal{F}$, then C has no states in common with any accepting cycle.*

We also have the following strength-reduction theorem, which allows us to use special algorithms for weak and terminal automata without analyzing the complete system.

Theorem 5. *Let $\mathcal{A}$ and $\mathcal{A}'$ be Büchi automata with $\mathcal{A} \leq \mathcal{A}'$, and let $\mathcal{A}$ be complete. If C is a weak (terminal) set of $\mathcal{A}'$, then C is a weak (terminal) set of $\mathcal{A}$.*

The strength of a strong SCC-closed set may actually reduce in going from $\mathcal{A}'$ to $\mathcal{A}$. For example, a strong SCC may split into two weak ones.

4.2 Algorithm

The results of Section 4.1 motivate the algorithm GENERIC-REFINEMENT of Fig. 1. The algorithm takes as arguments a Büchi automaton $\mathcal{A}$ and a set L of overapproximations to $\mathcal{A}$, which includes $\mathcal{A}$ itself. The relation $\leq$ on L is not required to be a total order. The algorithm returns true if an accepting cycle exists in $\mathcal{A}$, and false otherwise.

The algorithm keeps a set Work of obligations, each consisting of a set of states, the series of approximations that have been applied to it, and an upper bound on its strength. Initially, the entire state space is in Work, and the algorithm keeps looping until Work is empty or a fair SCC has been found. The loop starts by selecting an element (S, L', s) from Work and a new approximation $\mathcal{A}'$ from L. If $\mathcal{A}' = \mathcal{A}$, the algorithm may decide to run a standard model checking procedure on the SCC at hand. Otherwise, it decomposes S into accepting SCCs, and after analyzing their strengths, adds them as new Work. The algorithm uses several subroutines.

Subroutine SCC-DECOMPOSE, takes an automaton $\mathcal{A}'$ and a set S, intersects the state space of $\mathcal{A}'$ with S to yield a new automaton $\mathcal{A}''$, and returns the set of accepting SCCs of $\mathcal{A}''$. Note that an SCC of $\mathcal{A}''$ is not necessarily an SCC of $\mathcal{A}'$. The subroutine discards any unfair SCCs, as justified by Observation 4. Subroutine ANALYZE-STRENGTH returns the strength of the set of states. (See Section 2.) Subroutine MODEL-CHECK (shown) returns true iff a fair cycle is found using the appropriate model-checking technique for the strength of the given automaton.

The way entries and approximations are picked is not specified, and neither is it stated when ENDGAME returns true. These functions can depend on factors such as the strength of the entry, the approximations that have been applied to it, and its order of insertion. In later sections we shall make these functions concrete.

It follows from Theorem 3 that at any point of the algorithm, for any entry (S, L', s) of Work, S is an SCC-closed set of $\mathcal{A}$. At any point, the sets of states in Work are disjoint. Termination is guaranteed by the finiteness of L and of the set of SCCs of $\mathcal{A}$.

When decomposing an SCC-closed set S, we can use the complement of S as a don't care condition as discussed in Section 3. Because S may be small in comparison to Q, this may lead to a significant improvement in efficiency.

The SCC-refinement algorithm computes much of the needed information about the SCCs of a system on overapproximations of it. Because these overapproximations are often much simpler than the concrete system, this approach may be very efficient. Because the SCCs of a system are a refinement of the SCCs of any overapproximation, any computation on an overapproximate system divides the state space into several components, some of which are thrown away without considering them in the exact system, and some of which are analyzed further in isolation.

At any point in time, we keep around an overapproximation of the reachable states to discard unreachable SCCs. Whenever we refine the system, we compute the set of reachable states anew, limiting the search to the overapproximation that was obtained before. For this computation, we can use the overapproximation as care set. This scheme computes reachability multiple times, but [14] has shown that the use of approximate reachability information as a care set may more than compensate for that.

4.3 Underapproximations

The refinement approach that we have presented can be extended to the use of underapproximations. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be underapproximations of $\mathcal{A}$. First, as overapproximations can be used to discard the possibility of an accepting cycle, underapproximations can be used to assert their existence: if $\mathcal{A}_1$ contains an accepting cycle, then so does $\mathcal{A}$.

Furthermore, if an an SCC C_1 of $\mathcal{A}_1$ and an SCC C_2 of $\mathcal{A}_2$ overlap, then $\mathcal{A}$ contains an SCC $C \supseteq C_1 \cup C_2$. SCC-decomposition algorithms [18,1] compute each SCC starting from a seed. The seed is usually a single state, but, in order

```
type Entry = record
    S;   // An SCC-closed set of A
    L';  // Set of approximations that have been considered
    s    // Upper bound on the strength of the SCC
end

MODEL-CHECK(A, S, s){   // Automaton A, SCC-closed set S, and its strength s
    case s of
        strong: return Q_0 ∩ EG_F(S) ≠ ∅;   //call Emerson-Lei
        weak: return Q_0 ∩ EF EG(S) ≠ ∅;
        terminal: return Q_0 ∩ EF(S) ≠ ∅
    esac
}

GENERIC-REFINEMENT(A, L){   // Büchi automaton A = ⟨Q, Q_0, T, F, A, Λ⟩, and
                            //   set of overapproximations L with A ∈ L
    var
        Work: set of Entry;

    Work = {(Q, ∅, strong)}
    while Work ≠ ∅ do
        Pick an entry E = (S, L', s) from Work;
        Choose A' ∈ L such that there is no A'' ∈ L' with A'' ≤ A';

        if A' = A and ENDGAME(S, s) then
            if MODEL-CHECK(A, S, s) then
                return true
            fi
        else
            C := SCC-DECOMPOSE(S, A');
            if C ≠ ∅ and A' = A then
                return true
            else
                for all C ∈ C do
                    s := ANALYZE-STRENGTH(C, A');
                    insert (C, L' ∪ {A'}, s) in Work
                od
            fi
        fi
    od
    return false
}
```

Fig. 1. The generic SCC-refinement algorithm

to find C, we can use $C_1 \cup C_2$. Hence, we can avoid the recomputation of C_1 or C_2, and use $C_1 \cup C_2$ as a don't care set.

5 Composition

The SCC refinement algorithm described in Section 4 is generic because it does not specify: (1) What set of overapproximations L of the Büchi automaton $\mathcal{A}$ is available; (2) The rule to select the next approximation $\mathcal{A}'$ to be applied to a set S; (3) The priority function used to choose what element to retrieve from the Work set; and (4) The criterion used to decide when to switch to the endgame. These four aspects make up a *policy*; the first three are the subject of this section, while the fourth is discussed in Section 6.

5.1 Choice of the Approximations

We assume that $\mathcal{A}$ is the composition of a set of modules $M = \{M_1, \ldots, M_n\}$, and that the set L of overapproximations consists of the compositions of subsets of M:

$$L \subseteq \{M_{i_1} \times \cdots \times M_{i_p} \mid \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, n\}\} \ .$$

More flexible strategies (e.g., [9]) may generate larger sets of approximations and be compatible with our approach, but we shall not discuss them further.

We also assume that the states of $\mathcal{A}$ are the valuations of a set of r binary variables V; and that the sets of variables controlled by each module M_i are nonempty and form a partition of V. Hence, $n \leq r$ and for each $\mathcal{A}' \in L$ distinct from $\mathcal{A}$, $2\eta_{\mathcal{A}'} \leq \eta_{\mathcal{A}}$.

The set of all overapproximations generated from subsets of M forms a lattice, shown in Fig. 2 for $n = 4$. In the case illustrated by this figure, the coarsest approximation, which is the set of no modules, is the 1 of the lattice. (This approximation is never used in practice.) The exact system is the composition

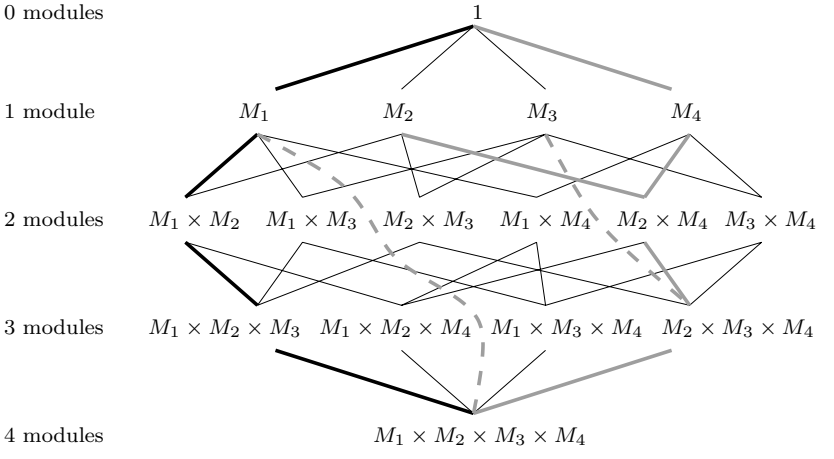


Fig. 2. Lattice of approximations

of all four modules. For sufficiently large n, it is impractical to make use of all 2^n overapproximations; consequently, we shall only consider efficient policies, in which a given state is contained in the set passed to SCC-DECOMPOSE $O(r)$ times.

Specifically, we shall stipulate that there is a constant λ, such that L can be partitioned into subsets $L_1, \ldots, L_r$ satisfying the following conditions: (1) $|L_i| \leq \lambda$; (2) for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$; (3) $\mathcal{A} \in L_r$.

Two cases are illustrated in the figure. In both cases, $(j_1, \ldots, j_n)$ is a permutation of $(1, \ldots, n)$ that identifies a linear order of the modules. At the left of the figure (solid thick lines), the algorithm of Fig. 1 uses a *popcorn-line* policy with $(j_1, \ldots, j_4) = (1, 2, 3, 4)$ and $\lambda = 1$. The approximations are:

$$L = \{\mathcal{A}_i = M_{j_1} \times \cdots \times M_{j_i} \mid 1 \leq i \leq n\} \ .$$

When an entry $E = (S, L', s)$ is retrieved from Work, the $\mathcal{A}_i$ of lowest index that is not present in L' is chosen as the next approximation $\mathcal{A}'$.

At the right (thick grey lines), $2n-1$ approximations are used in a *lightning-bolt* policy, for which $\lambda = 2$:

$$L = \{\mathcal{A}_{2i-1} = M_{j_1} \times \cdots \times M_{j_i} \mid 1 \leq i \leq n\} \cup \{\mathcal{A}_{2i} = M_{j_{1+1}} \mid 1 \leq i < n\} \ .$$

The selection of $\mathcal{A}'$ is done as in the previous case.

In Fig. 2, the order of the modules is $(4, 2, 3, 1)$. The approximations are:

$$\begin{array}{ll} \mathcal{A}_1 = M_4, & \mathcal{A}_2 = M_2, \\ \mathcal{A}_3 = M_4 \times M_2, & \mathcal{A}_4 = M_3, \\ \mathcal{A}_5 = M_4 \times M_2 \times M_3, & \mathcal{A}_6 = M_1, \\ \mathcal{A}_7 = M_4 \times M_2 \times M_3 \times M_1. & \end{array}$$

In both cases, the number of times a state is in the set passed to SCC-DECOMPOSE is bounded by the number of approximations in L. Therefore a popcorn-line policy tends to call SCC-DECOMPOSE fewer times, but a lightning-bolt policy may break up the SCC-closed sets with easy approximations ($\{\mathcal{A}_{2i}\}$) before applying to them harder approximations ($\{\mathcal{A}_{2i-1}\}$). Therefore, it tends to use less memory.

5.2 Complexity

The refinement algorithm described thus far cannot improve the worst-case complexity of the language emptiness check. Indeed, if all approximations distinct from $\mathcal{A}$ consist of one fair SCC, no benefit comes from the incremental SCC analysis. Under the stipulated conditions, however, it is easy to show that the incremental approach is within a constant factor from the non-incremental one.

Theorem 6. *If the set of approximations L can be partitioned into subsets $L_1, \ldots, L_r$ such that, for some constant λ, (1) $|L_i| \leq \lambda$; (2) for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$; and (3) $\mathcal{A} \in L_r$, then the generic refinement algorithm runs in $O(\eta_{\mathcal{A}} \log \eta_{\mathcal{A}})$ steps.*

Proof. The cost of SCC analysis for $\mathcal{A}'$ is bounded by $k\eta_{\mathcal{A}'} \log \eta_{\mathcal{A}'}$, for some constant k. Hence, the cost of analyzing all approximations and $\mathcal{A}$ itself is bounded by

$$k\eta_{\mathcal{A}} \log \eta_{\mathcal{A}}(\lambda + \lambda/2 + \lambda/4 + \cdots + \lambda/2^r) \ ,$$

which is bounded by $2\lambda k\eta_{\mathcal{A}} \log \eta_{\mathcal{A}}$. □

While we cannot hope for an improved run time in the worst case, we can expect that the refinement-based approach will be beneficial when the state space breaks up into many small SCC-closed sets. In particular, we can prove the following result.

Theorem 7. *Under the assumptions for L of Theorem 6, if for some constant γ, the pairs $(S, \mathcal{A}')$ passed to* SCC-DECOMPOSE *satisfy $|S| \leq \gamma\eta_{\mathcal{A}}/\eta_{\mathcal{A}'}$, then the refinement algorithm runs in $O(\eta_{\mathcal{A}})$ time.*

Proof. The analysis of $\mathcal{A}$ consists of the decomposition of SCC-closed sets of size bounded by γ. Their number is linear in $\eta_{\mathcal{A}}$, and each decomposition takes constant time. Hence, the total time for the analysis of $\mathcal{A}$ is $O(\eta_{\mathcal{A}})$. If $|C|$ is the number of states in SCC C of $\mathcal{A}'$, then $|C|\eta_{\mathcal{A}'}/\eta_{\mathcal{A}}$ is the *effective size* of C. The cost of analyzing $\mathcal{A}'$ is therefore $O(\eta_{\mathcal{A}'})$. With reasoning analogous to the one of Theorem 6, one finally shows that the total time is also $O(\eta_{\mathcal{A}})$. □

Another reason why the refinement-based approach may significantly outperform other algorithms is because it can discard large parts of the state space as soon as it establishes that they intersect no fair cycles by applying Observation 4. The advantage due to this ability to prune the search can be arbitrarily large.

5.3 Decomposition Trees

The popcorn-line approach, defines an SCC decomposition tree like the one of Fig. 3 that highlights the potential advantages of SCC refinement. The figure corresponds to a model of eight dining philosophers, with a property that states that under given fairness constraints, if a philosopher is hungry, she eventually eats. The system has nine modules. (The property automaton besides the philosophers.) The property passes, i.e., no fair cycles exist in the system. The tree of fair SCCs is shown. The nodes at Level i are the SCCs of $\mathcal{A}_i$. ($\mathcal{A}_1$ is the property automaton.) The size of each SCC is given; there are about 47k reachable states. Note that only very small sets of states remain after the first four modules[3] are composed, and that very little work is done on the exact system. The effective size of each SCC is also shown in parentheses. Since the effective sizes correlate to the actual computational effort, the numbers of Fig. 3 show that the cost is quite modest at all levels of refinement.

[3] These four modules are the property automaton, the philosopher named in the property, and her two neighbors.

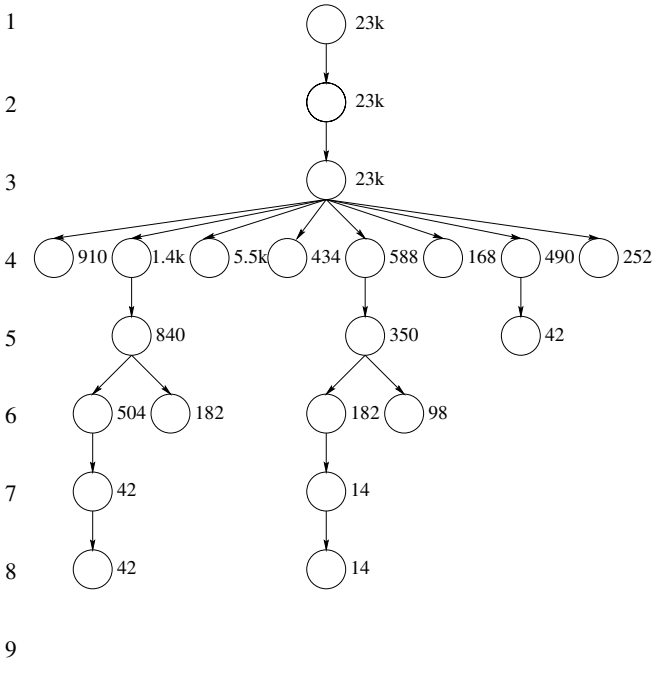


Fig. 3. SCC refinement tree

To define a policy we need to specify the order in which elements are retrieved from the Work set. Two obvious choices are FIFO and LIFO order. As one would expect, the SCC refinement tree is traversed in breadth-first manner for a FIFO order, and in depth-first manner for a LIFO order. When, as in Fig. 3, there are no fair cycles in $\mathcal{A}$, the order in which the tree is visited is immaterial. However, in the presence of fair cycles, one strategy may lead to earlier termination than the other. If one assumes that fair cycles are numerous, then depth-first search is particularly attractive. Breadth-first search, on the other hand, can be implemented with low overhead.

6 Implementation and Experiments

We describe here details of two implemented policies for the SCC refinement algorithm D'n'C, and of the experiments we ran. Both versions implement the popcorn-line approach, with breadth-first search of the SCC refinement tree. Both heuristically partition the system to be verified according to the strategy of [9]. They then sort the modules according to their distances from the state variables of the property automaton.

The two policies differ in when they switch to the endgame: The first policy de-emphasizes compositionality in comparison to strength reduction by perform-

ing only two levels of composition. At the first level it computes the SCCs of the property automaton, and at the second level it composes all the other modules of the system.

The second policy tries to exploit the full compositionality implied by Figs. 2 and 3. For ease of reference, we refer to the first policy as the *Two-level* method, and to the second as the *Multi-Level* method.

In both policies, if any fair SCCs are present, the algorithm checks their strength. If any of them are weak, they are grouped together, and the exact system is checked for cycles within these SCCs. The underlying assumption is that model checking weak SCCs is much cheaper than model checking strong SCCs. If D'n'C finds a cycle in the exact system, it terminates, otherwise it discards these SCCs. If no SCCs are present, the algorithm also terminates: there are no cycles. Otherwise, the approximate system is refined.

The Multi-Level method heuristically stops the refinement at some point, and then immediately composes all the remaining modules, thus proceeding directly to the exact system. Right now we are using a simple heuristic—we stop linear composition after 30% of the latches have been composed, and then "jump" to the the exact system to limit overhead, and to avoid having too many fair SCCs in the full SCC refinement tree. Once the exact system is reached, the Vis implementation of the Emerson-Lei algorithm is applied to each of its SCC-closed sets.

Our algorithm is implemented in Vis-1.4 [3], and the results of Table 1 were obtained by appropriately calling the standard Language Emptiness command of Vis. SCC analysis is performed with the Lockstep algorithm of [1] implemented as described in [15]. In Table 1, all examples were run with the same fixed order (obtained with dynamic variable reordering). For the same set of models and property automata, we also obtained a second table, with dynamic variable ordering turned on for each example. Similarly, we obtained a third table, using the EL2 variant of the Emerson-Lei algorithm [8]. Since the character of the results was not significantly different, the second and third tables were omitted for brevity.[4] All experiments were run on an IBM Intellistation running Linux with a 400MHz Pentium II processor with 1GB of SDRAM.

The table has four columns. The three fields of the first column give the name of the example, a symbol indicating whether the formula passes (P: no fair cycles exist) or fails (F: a fair cycle exists), and the number of binary latches in the system. The three fields of the second column, obtained by directly applying the Vis Emerson-Lei algorithm, give: (1) the time it took to run the experiment (Time/Out (T/O) indicates a run time greater than 14400s); (2) the peak number of live BDD nodes (in millions—the datasize limit was set to 750MB); and (3) the total number of preimage (EX) / image (EY) computations needed.

These same field descriptors also apply to the third and fourth columns (for the Two-Level and Multi-Level versions of the D'n'C algorithm), except that the latter has an additional field that indicates how the verification process

[4] The only exception to the statement was the fact that the example nmodem1 took only 209 seconds with EL2, versus 4384 for the original Emerson-Lei algorithm.

Table 1. Experimental results

			Emerson-Lei (Vis LE)			D'n'C Two-Level				D'n'C Multi-Level				
Circuit and LTL	P/F	latch num	Time (s)	Bdd (M)	EX/EY	Time (s)	Bdd (M)	EX/EY	time ratio	Time (s)	Bdd (M)	EX/EY	time ratio	
bakery1	F	56	212	5.1	5337/0	31	1.3	354/4	14%	27	1.3	484/328	12%	
bakery2	P	49	69	3.4	526/0	20	1.3	10/4	28%	20	1.3	62/73	28%	n
bakery3	P	50	421	14	1593/0	46	2.5	90/4	10%	43	1.8	537/428	10%	
bakery4	F	58	T/O	-	-/-	1950	3.4	1088/5	<13%	1337	4.7	947/96	<9%	
bakery5	F	59	T/O	-	-/-	1009	6.1	127/5	<7%	623	6.1	216/243	<4%	
eisenb1	F	35	23	1.0	416/0	16	0.9	21/4	69%	16	0.9	21/4	69%	
eisenb2	F	35	T/O	-	-/-	4800	8.2	162/5	<33%	1683	7.7	105/93	<11%	w
elevator1	F	37	210	14	163/0	49	2.8	132/9	23%	41	2.2	155/31	19%	
nmodem1	P	56	4384	11	5427/0	192	1.1	992/4	4%	233	0.6	5007/71	5%	
peterson1	F	70	17	1.1	24/0	20	1.3	19/4	117%	21	1.2	157/173	123%	
philo1	F	133	371	12	258/0	7	0.2	8/12	1%	7	0.2	8/12	1%	w
philo2	F	133	73	2.8	557/0	30	1.3	258/5	41%	12	0.5	25/44	16%	w
philo3	P	133	T/O	-	-/-	T/O	-	-/-	-	115	1.2	993/224	<1%	
shamp1	F	143	44	2.1	8/0	103	5.6	9/6	234%	87	2.2	266/280	197%	
shamp2	F	144	T/O	-	-/-	1892	16.	74/6	<13%	101	2.9	345/349	<1%	
shamp3	F	145	T/O	-	-/-	337	4.4	19/17	<2%	335	4.4	19/17	<2%	w
twoq1	P	69	12	0.4	25/0	4	0.1	7/9	33%	4	0.1	7/9	33%	n
twoq2	P	69	241	8.9	175/0	27	0.8	91/5	11%	30	0.9	181/95	12%	

terminates: 'n' means that the algorithm arrives at some intermediate level of the refinement process in which there no longer exists any fair SCC; 'w' means that there is a weak fair SCC found and it contains a fair cycle.

The property automata being used in the experiment are translated from LTL formulae. In order to avoid bias in favor of our approach, each model is checked against a *strong* LTL property automaton. Note that the presence of the n or w in the last field demonstrates that both pruning of the SCC refinement tree and strength reduction are active in these experiments.

We first compare the D'n'C algorithm to the one by Emerson and Lei. With only three exceptions out of 18 examples, the rows of the table indicate a significant (more than a factor of 2) performance advantage for the D'n'C algorithm.

Comparing the Two-Level and Multi-Level versions, one sees that on these examples, with four exceptions (eisenb2, philo2, philo3, and shamp2), the two policies give comparable performance. We think that this is because most of our examples are simple mutual-exclusion and arbitration protocols, in which the properties have little locality. We expect the compositional algorithm to do even better on models with more locality, and we are still enlarging the diversity of our sample set. On the other hand, one can see that the greater compositionality of the Multi-Level version proves its worth, especially on the larger examples.

7 Conclusions

In this paper we have presented a hybrid algorithm for fair cycle detection that uses abstractions to gradually refine the SCC-closed sets of a system to its SCCs. We have shown a general framework for SCC refinement and we have discussed different policies, based on the traversal of a lattice of overapproximate systems. Our algorithm has the advantages of being compositional, considering only parts of the complete state space, and taking into account the strength of an SCC to decide the proper model checking algorithm.

We have implemented two policies. In comparison to the original Emerson-Lei algorithm, our experimental results demonstrate significant and almost consistent performance improvement. This indicates the importance of all three improvement factors built into D'n'C: (1) SCC refinement, (2) compositionality, and (3) strength reduction. Though the compositional approach does not improve the worst-case complexity over the algorithm of [1], we have identified conditions under which the proposed algorithm runs in linear time.

The D'n'C algorithm can be highly parallelized by assigning different entries from the Work list to different processors. Processors that deal with disjoint sets of states have minimal communication and synchronization requirements. Although, the algorithm is geared towards symbolic model checking, SCC refinement can also be combined with explicit state enumeration approaches.

The experimental results show that even the simpler Two-Level policy performs very well in comparison to the Emerson-Lei algorithm. On all examples, the Multi-Level version of D'n'C is either superior to, or comparable to the Two-Level version. We have noted that superiority occurs for the larger examples, and we speculate that D'n'C will ultimately be able to handle some significantly larger examples. The simplicity of the implemented policies in comparison to the generality of Sections 4.1 and 5 suggests that there are many promising extensions and variations that so far remain experimentally unexplored. Among these, the joint application of over- and underapproximations is of special interest. Several iterative approaches to model checking have been proposed [11, 9,5]. These approaches do not carry the SCC decomposition from one level of refinement to the next. On the other hand, they use counterexamples to guide refinement—something that is currently missing from our implementation, and that could improve its effectiveness.

References

[1] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, November 2000. LNCS 1954.

[2] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 154–169. Springer-Verlag, Berlin, July 2000.

[6] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, November 1990.

[7] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[8] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.

[9] J.-Y. Jang. *Iterative Abstraction-based CTL Model Checking*. PhD thesis, University of Colorado, Department of Electrical and Computer Engineering, 1999.

[10] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[11] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[12] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[14] I.-H. Moon, J.-Y. Jang, G. D. Hachtel, F. Somenzi, C. Pixley, and J. Yuan. Approximate reachability don't cares for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 351–358, San Jose, CA, November 1998.

[15] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.

[16] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proceedings of the Design Automation Conference*, pages 225–231, San Diego, CA, June 1994.

[17] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[18] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design*, 19(10):1225–1230, October 2000.

Unavoidable Configurations of Parameterized Rings of Processes

Marie Duflot[1], Laurent Fribourg[1], and Ulf Nilsson[2]

[1] LSV, CNRS & ENS de Cachan, 61 av. du Prés. Wilson,
94235 Cachan cedex, France
[2] IDA, Linköping University, 581 83 Linköping, Sweden
{duflot,fribourg}@lsv.ens-cachan.fr, ulfni@ida.liu.se

Abstract. Rewrite systems over words are often used for modeling distributed algorithms over linear networks (or rings) of N processes, where N is a parameter. Here we are interested in constructing a regular set of configurations $\mathcal{G}$ which is *unavoidable*, i.e., such that any infinite derivation intersects $\mathcal{G}$. We give some sufficient conditions of the rewrite system that allow us to construct an unavoidable set $\mathcal{G}$ using Caucal's algorithm of *prefix rewriting*. This construction is used to show the *convergence* property of distributed algorithms to closed subsets of configurations. The method is useful for proving the correctness of self-stabilizing algorithms and the liveness property of termination detection algorithms. It has been implemented, and successfully applied to several significant examples, treated in a uniform mechanical way for the first time.

1 Introduction

We consider distributed algorithms over linear or circular arrays with a parametric number N of machines. All machines (except perhaps the bottom and the top ones) are identical finite-state automata, which communicate by reading the state of their neighbors. A global system configuration is the concatenation of all the local states of the machines. For brevity, we consider only moves that simultaneously modify the local states a and b of *two* contiguous machines. (Moves modifying just one or more than two machines can be treated similarly.) The moves are described uniformly by rewrite rules of the form $ab \rightarrow a'b'$. We want to prove *convergence* of the system to a restricted set $\mathcal{L}$ of configurations, i.e., proving that every infinite sequence of reductions reaches $\mathcal{L}$.

As shown in [3], using a theoretical result of Dershowitz, the problem can be solved by exploring the branches of an infinite tree, representing repeated steps of reduction and "narrowing" (i.e. unification coupled with reduction). In order to make the exploration feasible, the infinite tree has to be folded into a finite, oriented graph. But this presupposes a manual process of generalization to infer a regular language from a finite set of words (see [3]). In this paper we address the problem of generating such a graph in an automatic and exact way (without over-approximation, as in [3]). More precisely, we observe that if only narrowing steps are taken into account, each step can be viewed either

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 472–486, 2001.
© Springer-Verlag Berlin Heidelberg 2001

as a form of prefix or suffix rewriting, in which case Caucal's algorithm for generating regular languages using prefix rewriting can be applied to generate the graph automatically. We also discuss sufficient conditions under which we may disregard reduction steps altogether without loss of completeness.

The method has been implemented, and all examples of convergence of self-stabilizing algorithms treated manually in [3], have been processed mechanically. As a further application of the new method, we also demonstrate the verification of liveness properties of *termination detection* algorithms.

Comparison with related work. As mentioned above, we focus on the problem of proving convergence properties of distributed algorithms. In the classical framework, such proofs are typically done by exhibiting a well-founded measure on the set of configurations that strictly decreases after each step of reduction (or a bounded number of them) as long as the set $\mathcal{L}$ has not been reached. These measures are usually very subtle and require a deep knowledge of the analyzed algorithm in order to be discovered by hand (e.g., [17,25,14]). Our method, in contrast, does not require such specific knowledge, and can be mechanized.

We use several basic ideas that already appear in the literature. The idea of viewing distributed algorithms over linear networks of machines as rewrite systems and sets of configurations as regular languages, has been used before (see e.g., [1,16,19,20,22,26]). However, interest is generally restricted to showing that the *reachability set* of a distributed algorithm, i.e. the set of all configurations reached by a *finite* derivation, is regular. In contrast we focus on unavoidable sets, i.e., sets such that any *infinite* derivation traverses them. That is, we want to prove liveness properties rather than safety properties. To our knowledge, only two recent works investigate the verification of liveness properties for parameterized rings [7,24]. In [7] Bouajjani et al. prove liveness properties (or, more generally, ω-regular properties) of distributed algorithms by using strategies for guessing automatically extended forms of reachability sets in the framework of Büchi automata. Pnueli and Shahar [24] use related techniques of "acceleration" but in the framework of second-order monadic logic. In [7], the method basically relies on the fact that every rewrite system $\mathcal{S}$ modeling a studied algorithm is Noetherian (i.e., it does not contain infinite derivations). This is never the case in the examples treated here. We only assume that a strict subset of $\mathcal{S}$ (obtained by removing the rules affecting the top machine) is Noetherian, which is a crucial difference. In [24], the method must take into account an unbounded number of fairness assumptions, several for each process, and these requirements are also parameterized, often making the computation diverge (see [24], p. 340). In contrast, our method assumes only the weak fairness assumption that the top machine is affected at least once in every infinite derivation. The techniques that we use here are also very different from the ones used in [7,24], and involve first-order rewriting operations such as narrowing.

Plan of the paper. Section 2 recalls some definitions about string rewrite systems. Section 3 explains the basic operation of narrowing as a form of prefix rewriting. Section 4 explains how to construct regular unavoidable languages

using a closure assumption on the set of narrowings. Section 5 gives a syntactic criterion ensuring the closure of this set. Section 6 explains how to use unavoidable languages for proving the correctness property of self-stabilizing algorithms and the liveness property of termination detection algorithms. The implementation and experimental results are given in Section 7. Section 8 concludes.

2 String Rewriting for Linear Networks of Machines

We use string rewrite systems to model the moves that affect the local states of the component machines. Local states of machines belong to a finite alphabet Σ. Each word $a_1 \cdots a_N$ (where N is a parameter) constitutes a global configuration of the system where machine i is in state a_i ($1 \leq i \leq N$). In the following, each move involves exactly two contiguous machines. The moves are described by rewrite rules. Special rules have to be considered for the bottom and top machines. In this framework, we model the actions of a distributed algorithm as a set $\mathcal{S}$ of rewrite rules. We will use first-order variables W, X, Y to represent in a symbolic way a generic set of states for a portion of contiguous machines. We first adapt some basic definitions from (string) rewrite systems [5,11] to our framework.

A *ground word* is an element of Σ^*, with ε denoting the empty word. An *open word* is an expression of the form uWv where u, v are elements of $\Sigma^+ (= \Sigma^* - \{\varepsilon\})$. Henceforth, *word* refers to an open or ground word, which is written t, u or v (possibly adorned). Letters are denoted by a, b, c or d (possibly adorned). A rewrite system $\mathcal{S}$ is a set of length-preserving rules, divided into three subsets $Top_{\mathcal{S}}$, $Bottom_{\mathcal{S}}$ and $Middle_{\mathcal{S}}$: *bottom rules* in $Bottom_{\mathcal{S}}$ are applied to the two leftmost letters of words; *top rules* in $Top_{\mathcal{S}}$ are applied to the leftmost and rightmost letters of words; the other rules in $Middle_{\mathcal{S}}$ are called *middle rules*. More precisely, let a, b, a' and b' be letters of Σ, and X, Y variables. Then:

- $Bottom_{\mathcal{S}}$ consists of rules of the form $abX \to a'b'X$
- $Middle_{\mathcal{S}}$ consists of rules of the form $XabY \to Xa'b'Y$ (with $X \neq \varepsilon$)
- $Top_{\mathcal{S}}$ consists of rules of the form $bXa \to b'Xa'$

A top rule makes the leftmost and rightmost components of the array communicate: the system should be considered a ring of machines.[1]

Example. Consider the alphabet $\Sigma = \{0, 1, 2\}$ and the rewrite rules $\mathcal{BD} = \{B_1, M_1, M_4, T_4\}$, which is a simplified version of Beauquier-Debas' original system [4] (see the example of Sect. 6.1 for the full system):

$$
\begin{aligned}
&B_1 : \ 12X \to 21X \\
&M_1 : X10Y \to X01Y \text{ with } X \neq \varepsilon \\
&M_4 : X02Y \to X20Y \text{ with } X \neq \varepsilon \\
&T_4 : \ 2X1 \to 1X2
\end{aligned}
$$

Here B_1 is a bottom rule, T_4 a top rule, M_1 and M_4 middle rules.

[1] One can consider also top rules of the form $Xab \to Xa'b'$ where such communication does not exist. In this case $Y \neq \varepsilon$ should be added to middle rules.

Reduction consists in replacing an instance of a rule's left-hand side by the corresponding instance of the rule's right-hand side. Formally, given two ground terms t and t', we say that *reduction* applies from t to t' via middle rule $M: XabY \to Xa'b'Y$ if $t = uabv$ and $t' = ua'b'v$ for some strings $u \in \Sigma^+, v \in \Sigma^*$. This is written $t \to_M t'$. The notion of reduction via M also applies when t is an open word of the form $t_1 W t_2$. Reduction of t then consists in replacing ab with $a'b'$, either in the left part t_1 or in the right part t_2 of t. The notion of reduction of a ground/open word t is defined similarly for a bottom rule B (resp. top rule T) instead of a middle rule M. In the case of a bottom reduction (resp. top reduction), the replaced letters a and b of t occur in the first and second position (resp. last and first ones). We say that reduction applies from t to t' via $\mathcal{S}$, and write $t \to_{\mathcal{S}} t'$, if $t \to_R t'$ for some rule $R \in \mathcal{S}$. The reduction of a set of words J via $\mathcal{S}$ is defined by $Reduce_{\mathcal{S}}(J) = \{t' \mid \exists t \in J \ \wedge \ t \to_{\mathcal{S}} t'\}$. The set J is said to be *closed* under a rewrite system $\mathcal{S}$ if $Reduce_{\mathcal{S}}(J) \subseteq J$.

Example. The set $\mathcal{L} = 20^*10^* \cup 10^*20^*$ is closed under $\mathcal{BD}$.

The mapping $Reduce_{\mathcal{S}}$ can be seen as a *transducer* (see, e.g., [20]) and $Reduce_{\mathcal{S}}(J)$ is regular if J is regular. As usual, $\to_{\mathcal{S}}^*$ denotes the reflexive-transitive closure of $\to_{\mathcal{S}}$. It is well-known that the set generated from J by iterated application of $\to_{\mathcal{S}}$ (i.e, by application of $\to_{\mathcal{S}}^*$), is generally not regular, even if J is regular. Nevertheless, Caucal has shown that iterated *prefix* reduction (at the beginning of a word) and iterated *suffix* reduction (at the end of a word), behave as transducers. Furthermore, Caucal has designed an efficient algorithm, polynomial in time for the size of $\mathcal{S}$, for these computations [8]. In this paper, we use prefix and suffix strategies of reduction in order to apply Caucal's algorithm.

A *ground derivation* is a (possibly infinite) sequence of reductions over ground words. A set of rules $\mathcal{S}$ is said to be *Noetherian* if any ground derivation using rules of $\mathcal{S}$ is finite.

Example. For the system $\mathcal{BD}$ defined above, we have $t_1 = 2\underline{10}0 \to_{M_1} t_2 = 20\underline{10} \to_{M_1} t_3 = \underline{20}0\underline{1} \to_{T_4} t_4 = 10\underline{02} \to_{M_4} t_5 = 1\underline{02}0 \to_{M_4} t_6 = \underline{12}00 \to_{B_1} t_1 = 2100$. (The letters replaced at each step of reduction are underlined for the sake of clarity.) We have thus an infinite (cyclic) ground derivation. Hence $\mathcal{BD}$ is *non*-Noetherian. On the other hand, $\mathcal{BD} - T_4$ is Noetherian. Consider the well-founded measure ψ that associates $\sum_{i \in S_1}(N-i) + \sum_{j \in S_2} j$ with every word $a_1 \cdots a_N$, where S_1 (resp. S_2) is the set of positions i (resp. j) such that $a_i = 1$ (resp. $a_j = 2$). It is easy to see that application of rule M_1 (or M_4) makes ψ decrease by 1, while B_1 makes ψ decrease by 2.

Note that testing the closure of a regular set J under $\mathcal{S}$ is decidable: it consists in checking the inclusion of two regular sets.

For each middle rule M (resp. bottom rule B) it is convenient to define two variants of M (resp. one variant of B) as follows:

Definition 1. *Given a middle rule $M: XabY \to Xa'b'Y$ and a bottom rule $B: abY \to a'b'Y$,*

- *the* prefix extension *of* M*, written* M_{pre}*, is:* $X\odot bY \to X\odot a'b'Y$,
- *the* suffix extension *of* M*, written* M_{suf}*, is:* $Xa\odot Y \to Xa'b'\odot Y$,
- *the* suffix extension *of* B*, written* B_{suf}*, is:* $a\odot Y \to a'b'\odot Y$,

where $\odot$ *is a new constant added to* Σ.

Let t be an open word t_1Wt_2, and denote by $t_\odot$ the word $t_1\odot t_2$ obtained from t by instantiating W with $\odot$. When t_2 starts with b, reduction via M_{pre} applies to $t_\odot$. Such a reduction affects a prefix of $\odot t_2$, and is hence a *prefix reduction* of the right part of $t_\odot$. Likewise, reduction via M_{suf} (resp. B_{suf}) affects a suffix of $t_1\odot$, and is hence a *suffix reduction* of the left part of $t_\odot$. The rule extensions will allow us to consider the operation of *narrowing*, defined hereafter, as a form of prefix or suffix reduction, and thus to apply Caucal's algorithm.

3 Narrowing

Next we define *narrowing* (e.g., [11]) over open words t_1Wt_2, with $t_1, t_2 \in \Sigma^+$. In this context, narrowing can be seen as a form of replacement of W with Wb, aW or ε, followed by reduction of the resulting word, as explained below.

3.1 Right, Left, and Bottom Narrowing

Definition 2. *Consider a middle rule* $M: XabY \to Xa'b'Y$*, a bottom rule* $B: abX \to a'b'X$*, and two open words* t, t'.

- *We say that* right narrowing *applies from* t *to* t' *via* M *if* $t = uWbv$ *and* $t' = uWa'b'v$*, for some* $u \in \Sigma^+, v \in \Sigma^*$*. This is written* $t \leadsto_M^{right} t'$.
- *We say that* left narrowing *applies from* t *to* t' *via* M *if* $t = uaWv$ *and* $t' = ua'b'Wv$*, for some* $u, v \in \Sigma^+$*. This is written* $t \leadsto_M^{left} t'$.
- *We say that* bottom narrowing *applies from* t *to* t' *via* B *if* $t = aWv$ *and* $t' = a'b'Wv$*, for some* $v \in \Sigma^+$*. This is written* $t \leadsto_B^{bot} t'$.

Right narrowing from t to t' via M corresponds to prefix reduction from $t_\odot$ to $t'_\odot$ via M_{pre} (since $uWbv \leadsto_M^{right} uWa'b'v$ iff $u\odot bv \to_{M_{pre}} u\odot a'b'v$). Likewise, left narrowing via M (resp. bottom narrowing via B) corresponds to suffix reduction via M_{suf} (resp. B_{suf}). We have indeed: $uaWv \leadsto_M^{left} ua'b'Wv$ iff $ua\odot v \to_{M_{suf}} ua'b'\odot v$ (resp. $aWv \leadsto_B^{bot} a'b'Wv$ iff $a\odot v \to_{B_{suf}} a'b'\odot v$).

Example. Using $M_4 : X02Y \to X20Y$, we have $01W200 \leadsto_{M_4}^{right} 01W2000$. This narrowing is mimicked by the reduction $01\odot 200 \to_{M_{4pre}} 01\odot 2000$, using the prefix extension $M_{4pre}: X\odot 2Y \to X\odot 20Y$.

We write $t \leadsto_{\mathcal{S}}^{right} t'$ if $t \leadsto_M^{right} t'$ for some rule $M \in \mathcal{S}$, and similarly for left and bottom narrowing. We will abbreviate $\leadsto_{\mathcal{S}}^{bot} \cup \leadsto_{\mathcal{S}}^{left} \cup \leadsto_{\mathcal{S}}^{right}$ by $\leadsto_{\mathcal{S}}$. Given a set J of open words, we write:

$$Narrow_{\mathcal{S}}^*(J) = \{t' \mid \exists t \in J \wedge t \leadsto_{\mathcal{S}}^* t'\}$$
$$(Narrow_{\mathcal{S}} + Reduce_{\mathcal{S}})^*(J) = \{t' \mid \exists t \in J \wedge t\ (\leadsto_{\mathcal{S}} \cup \to_{\mathcal{S}})^*\ t'\}.$$

The first set is obtained by iterated application of narrowing only; the second one by interleaving narrowing and reductions via $\mathcal{S}$. Let $I_{\mathcal{S}}$ be the initial set of $\mathcal{S}$; i.e., the set of open words $b'Wa'$ corresponding to all right-hand sides of top rules $bXa \to b'Xa'$ of $\mathcal{S}$ (with W replacing X). We write:

$$\mathcal{N}^*_{\mathcal{S}} = Narrow^*_{\mathcal{S}}(I_{\mathcal{S}})$$
$$(\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^* = (Narrow_{\mathcal{S}} + Reduce_{\mathcal{S}})^*(I_{\mathcal{S}}).$$

In the computation of $\mathcal{N}^*_{\mathcal{S}}$, right narrowing on the one hand, bottom and left narrowing on the other hand, are independent (since they involve sub-words separated by W) and can be done in parallel. Since right narrowing (resp. bottom and left narrowing) can be simulated by prefix reduction (resp. suffix reduction), $\mathcal{N}^*_{\mathcal{S}}$ can be computed by iterated prefix and suffix reductions done in parallel, starting from the right-hand sides of $Top_{\mathcal{S}}$. By Caucal's result [8] (cf. [5,6]), it follows directly that:

Proposition 3. *$\mathcal{N}^*_{\mathcal{S}}$ is regular and can be constructed in polynomial time in the size of $\mathcal{S}$.*

We decompose $\mathcal{N}^*_{\mathcal{S}}$ into $\mathcal{M}_0 \cup \mathcal{M}_1$, where $\mathcal{M}_0$ (resp. $\mathcal{M}_1$) is the subset generated with 0 (resp. 1) application of bottom narrowing. (It is easy to see that bottom narrowing can be applied at most once). More precisely, $\mathcal{N}^*_{\mathcal{S}} = \mathcal{M}_0 \cup \mathcal{M}_1$ where

- $\mathcal{M}_0$ is the result of applying $(\rightsquigarrow^{right})^*$ to $I_{\mathcal{S}}$, and
- $\mathcal{M}_1$ the result of applying $\rightsquigarrow^{bot} \circ(\rightsquigarrow^{left})^*$ in parallel with $(\rightsquigarrow^{right})^*$ to $I_{\mathcal{S}}$.

Example. For the Beauquier-Debas system $\mathcal{BD}$, the initial set $I_{\mathcal{BD}}$ is $1W2$. Application of $(\rightsquigarrow^{right})^*$ to $I_{\mathcal{BD}}$ via $M_4 : X02Y \to X20Y$ yields $\{1W2, 1W20, 1W200, 1W2000, \dots\}$. Thus the set $\mathcal{M}_0$ is $1W20^*$. On the other hand, the application of $\rightsquigarrow^{bot}$ to $I_{\mathcal{BD}}$ via B_1 yields $21W2$. Then subsequent application of $(\rightsquigarrow^{left})^*$ via $M_1 : X10Y \to X01Y$ yields $\{21W2, 201W2, 2001W2, \dots\}$, that is 20^*1W2. By (parallel) application of $(\rightsquigarrow^{right})^*$, we finally generate $\mathcal{M}_1 = 20^*1W20^*$. We have $\mathcal{N}^*_{BD} = \mathcal{M}_0 \cup \mathcal{M}_1 = 1W20^* \cup 20^*1W20^*$.

3.2 Grounding Narrowing

We now consider an operation which removes W from an open word by replacing W with ε followed by a reduction step, thus yielding a ground word.

Definition 4. *Consider a middle rule $M : XabY \to Xa'b'Y$, a bottom rule $B : abY \to a'b'Y$, an open word t and a ground word t'.*

- *We say that grounding narrowing applies from t to t' via M if $t = uaWbv$ and $t' = ua'b'v$, for some $u \in \Sigma^+, v \in \Sigma^*$. This is written $t \rightsquigarrow^{gr}_M t'$.*
- *We say that grounding narrowing applies from t to t' via B if $t = aWbv$ and $t' = a'b'v$, for some $v \in \Sigma^*$. This is written $t \rightsquigarrow^{gr}_B t'$.*

We write $t \leadsto_{\mathcal{S}}^{gr} t'$ *iff* $t \leadsto_{M}^{gr} t'$ *or* $t \leadsto_{B}^{gr} t'$ *for some rules* M, B *of* $\mathcal{S}$. *Given a set* J *of open words, the grounding narrowing of* J *via* $\mathcal{S}$, *written* $Gr_{\mathcal{S}}(J)$ *or more simply* $Gr(J)$, *is* $Gr(J) = \{t' \mid \exists t \in J \wedge t \leadsto_{\mathcal{S}}^{gr} t'\}$ *and we set* $\mathcal{G}_{\mathcal{S}} = Gr_{\mathcal{S}}(\mathcal{N}_{\mathcal{S}}^*)$.

The mapping Gr can also be seen as a transducer, hence $Gr(J)$ is regular if J is regular. In particular, $\mathcal{G}_{\mathcal{S}}$ is regular. It can be obtained by applying grounding narrowing to $\mathcal{M}_0$ via a bottom rule B and grounding narrowing to $\mathcal{M}_1$ via a middle rule M.

Example. By application of grounding narrowing to $\mathcal{M}_0 = 1W20^*$ via the rule $B_1 :\ 12X \to 21X$, we get 210^*. There is no grounding narrowing applicable to $\mathcal{M}_1 = 20^*1W20^*$ via M_1 or M_4. Hence $\mathcal{G}_{\mathcal{BD}} = Gr(\mathcal{M}_0) = 210^*$.

4 Unavoidable Regular Languages and Convergence

We now focus on the problem of constructing a regular set unavoidable by $\mathcal{S}$. Let us first define the notion of "unavoidable set".

Definition 5. *Given a rewrite system* $\mathcal{S}$, *a set* $\mathcal{K}$ *of ground words is said to be unavoidable by* $\mathcal{S}$ *if, for all infinite ground derivation of the form* $t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} \cdots \to_{\mathcal{S}} t_n \to_{\mathcal{S}} \cdots$, *there exists a word* $u \in \mathcal{K}$ *such that* $u = t_i$ *for some* $i > 0$.

Using the notion of unavoidable set, we can now rephrase the main result (Theorem 1, Section 3.2) of [3] in a more elegant way, as follows:

Theorem 6. [3] *If the rewrite system* $\mathcal{S} - Top_{\mathcal{S}}$ *is Noetherian, then* $Gr((\mathcal{N}_{\mathcal{S}} + \mathcal{R}_{\mathcal{S}})^*)$ *is unavoidable by* $\mathcal{S}$.

The idea of the proof is as follows: Since $\mathcal{S} - Top_{\mathcal{S}}$ is Noetherian, all infinite ground derivations contain at least one application of $Top_{\mathcal{S}}$. After application of such a rule, say $bXa \to b'Xa'$, the ground word generated is of the form $b'ua'$ for some $u \in \Sigma^*$. Therefore the set of words derived from $b'ua'$ is unavoidable. However, many words are generated in a redundant manner. Exploiting some reordering properties of infinite ground derivations (first discovered by Dershowitz [10]), it is possible to prove that no unavoidable word is lost when focusing on the derivations that start from $b'Wa'$, and replace iteratively W via successive narrowings interleaved with reductions. See [3] for the full proof.

We claim that the Noetherian condition on $\mathcal{S} - Top_{\mathcal{S}}$ is generally met in practice. Such a condition implies that all infinite derivations affect the rightmost machine at least once (hence an infinite number of times). It can be considered as a weak form of fairness concerning the top machine. For instance, in "token passing" algorithms (as those considered in our examples), the tokens visit the top machine at least once during infinite derivations, so the Noetherian condition on $\mathcal{S} - Top_{\mathcal{S}}$ is always satisfied. Note however that, in order to prove formally that the condition actually holds, we must generally discover, by hand, a specific well-founded measure that decreases after each step of reduction, top excepted (as, e.g., ψ given in an example of Section 2, or norm F of [25], p. 464).

The construction of an unavoidable set thus reduces to the generation of $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$. In [3], over-approximations of $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ were generated in the form of regular languages, but the process was done in an *ad hoc* manner for each treated example, using human insight. It also required the construction of a graph expressing reachability relations among the inferred sublanguages, in order to detect possible divergent loops of execution. In contrast, we now state a simple and sufficient condition that guarantees that $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ is regular and can be constructed in both an exact and automated way. Let us suppose that $\mathcal{N}_\mathcal{S}^*$ is closed under $\mathcal{S}$ (i.e., $Reduce_\mathcal{S}(\mathcal{N}_\mathcal{S}^*) \subseteq \mathcal{N}_\mathcal{S}^*$). Then $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^*$ is simply $\mathcal{N}_\mathcal{S}^*$. Using proposition 1, Theorem 1 thus becomes:

Proposition 7. *If the rewrite system $\mathcal{S} - Top_\mathcal{S}$ is Noetherian and $\mathcal{N}_\mathcal{S}^*$ is closed under $\mathcal{S}$, then $\mathcal{G}_\mathcal{S}$ is a regular unavoidable set.*

Example. In the Beauquier-Debas example, $\mathcal{N}_{\mathcal{BD}}^* = 1W20^* \cup\ 20^*1W20^*$. It is closed under $\mathcal{BD}$, and $\mathcal{BD} - Top_{\mathcal{BD}}$ is Noetherian. Therefore $\mathcal{G}_{\mathcal{BD}} = 210^*$ is unavoidable.

Checking the closure of $\mathcal{N}_\mathcal{S}^*$ under $\mathcal{S}$ is decidable (since $\mathcal{N}_\mathcal{S}^*$ is regular). Note that, even when $\mathcal{N}_\mathcal{S}^*$ is closed, $\mathcal{G}_\mathcal{S}$ is generally not closed itself. Therefore, infinite derivations traverse $\mathcal{G}_\mathcal{S}$, without staying within it. But if, additionally, $\mathcal{G}_\mathcal{S}$ is contained in a *closed* set $\mathcal{L}$, all infinite ground derivations are eventually "captured" by $\mathcal{L}$; we have *convergence* to $\mathcal{L}$. Formally:

Definition 8. *Let $\mathcal{S}$ be a rewrite system and $\mathcal{L}$ a set of configurations. We say that $\mathcal{S}$* converges to *$\mathcal{L}$ and write $Conv(\mathcal{S}, \mathcal{L})$, if, for all infinite ground derivation of the form $t_1 \rightarrow_\mathcal{S} t_2 \rightarrow_\mathcal{S} \cdots \rightarrow_\mathcal{S} t_n \rightarrow_\mathcal{S} \cdots$, there exists $k > 0$ such that $t_i \in \mathcal{L}$ for all $i \geq k$.*

Convergence properties are useful for proving correctness of self-stabilizing algorithms and liveness properties of termination detection algorithms, as illustrated in Section 6.

Proposition 9. *Let $\mathcal{S}$ be a rewrite system and $\mathcal{L}$ a set of configurations closed under $\mathcal{S}$. If there exists an unavoidable set $\mathcal{K}$ for $\mathcal{S}$ such that $\mathcal{K} \subseteq \mathcal{L}$, then we have $Conv(\mathcal{S}, \mathcal{L})$.*

From proposition 2 and proposition 3, it follows:

Theorem 10. *Consider a system $\mathcal{S}$ such that $\mathcal{S} - Top_\mathcal{S}$ is Noetherian, and a set $\mathcal{L}$ closed under $\mathcal{S}$. If $\mathcal{N}_\mathcal{S}^*$ is closed under $\mathcal{S}$ and if $\mathcal{G}_\mathcal{S}$ is included in $\mathcal{L}$, then we have $Conv(\mathcal{S}, \mathcal{L})$.*

Note that, if $\mathcal{L}$ is regular, the inclusion of $\mathcal{G}_\mathcal{S}$ in $\mathcal{L}$ is decidable (since $\mathcal{G}_\mathcal{S}$ is regular). Theorem 2 is a major enhancement of the results in [3], as it allows to mechanically prove convergence properties, using the construction of $\mathcal{N}_\mathcal{S}^*$ via prefix/suffix rewriting.

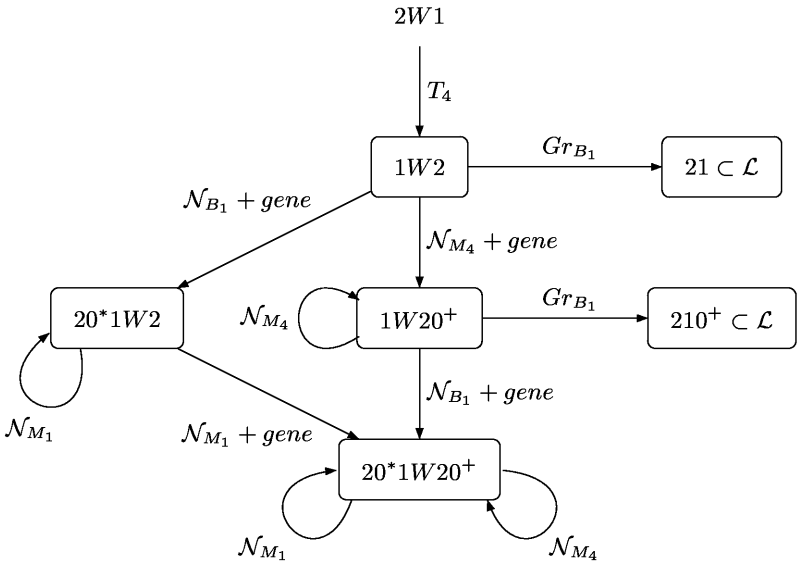


Fig. 1. Construction by hand of the graph in [3] for proving convergence of system $\mathcal{BD}$

Example. In the Beauquier-Debas example, $\mathcal{N}^*_{\mathcal{BD}} = 1W20^* \cup 20^*1W20^*$ is closed under $\mathcal{BD}$ and $\mathcal{G}_{\mathcal{BD}} = 210^*$ is included in $\mathcal{L} = 20^*10^* \cup 10^*20^*$. Since $\mathcal{BD} - Top_{\mathcal{BD}}$ is Noetherian and $\mathcal{L}$ is closed, we have $Conv(\mathcal{BD}, \mathcal{L})$. Such a proof has been automatically produced by an implemented program (see Section 7). This is a breakthrough with respect to [3], where the proof of convergence of $\mathcal{BD}$ to $\mathcal{L}$ required manual construction of a graph via an ad hoc process of generalization, as illustrated in Figure 1. (In the figure, an edge labeled with $\mathcal{N} + gene$ models the inference of a regular language through one step of narrowing followed by generalization. For example the edge, labeled with $\mathcal{N}_{M_4} + gene$, from $1W2$ to $1W20^+$, means that narrowing via M_4 leads from $1W2$ to $1W20$, which is generalized to $1W20^+$.)

5 A Sufficient Syntactic Condition for Closure of $\mathcal{N}^*_{\mathcal{S}}$

Although the closure condition of $\mathcal{N}^*_{\mathcal{S}}$ in Theorem 2 is decidable, it is difficult to verify, from simple inspection of the rules of $\mathcal{S}$, whether it holds or not. We next give a syntactic criterion on $\mathcal{S}$, which has a simple operational meaning, and guarantees the closure condition. This criterion is especially useful for self-stabilizing algorithms (see Section 6) involving mutual exclusion. In such a case, the reducible positions in configurations correspond to "tokens" or "privileges". The number φ of such tokens never increases (no creation of tokens). The problem is to show that φ always eventually decreases until just one token remains; in which case the set $\mathcal{L}$ of "legitimate" configurations is reached. We explain now that, if we focus on derivations that preserve the number φ of tokens, then

the system is likely to satisfy an operational condition of *unidirectionality* that guarantees the closure property. We need some preliminary definitions.

Definition 11. *A middle U-turn from left to left (resp. from right to right) in a ground derivation is a sequence of reductions of the form* $u\underline{abc}v \to ua'\underline{b'c}v \to u\underline{a'b}''c'v \to ua''b'''c'v$ *(resp.* $ua\underline{bc}v \to u\underline{ab}'c'v \to ua'\underline{b''c}'v \to ua'b'''c''v$*), where* u *and* v *are strings.*

Definition 12. *A bottom U-turn in a ground derivation is a sequence of reductions of the form* $\underline{abc}v \to a'\underline{b'c}v \to \underline{a'b}''c'v \to a''b'''c'v$*. A top right (resp. top left) U-turn is a sequence of reductions* $\underline{c}ua\underline{b} \to c'u\underline{ab}' \to c'ua'b''$ *such that* $\underline{dvb}'' \to d'vb'''$ *for some* $d \in \Sigma, v \in \Sigma^+$ *(resp.* $\underline{c}au\underline{b} \to \underline{c'a}ub' \to c''a'ub'$ *such that* $\underline{c''v}\underline{d} \to c'''vd'$ *for some* $d \in \Sigma, v \in \Sigma^+$*).*

Definition 13. *A repetition in a derivation is a sequence of two consecutive reductions that rewrite the same positions in the term (e.g.,* $u\underline{ab}v \to u\underline{a'b'}v \to ua''b''v$*).*

A rewriting system $\mathcal{S}$ is said to be *unidirectional* if no U-turn (middle from left to left, middle from right to right, bottom, top left, top right) and no repetition are possible, along any derivation via $\mathcal{S}$. From an operational point of view, it roughly means that each token always travels in the same direction. It is easy to determine by inspection of the left- and right-hand sides of rules, whether a system $\mathcal{S}$ is unidirectional or not.

Proposition 14. *In a unidirectional system, we have* $(\mathcal{N}_\mathcal{S} + \mathcal{R}_\mathcal{S})^* = \mathcal{N}_\mathcal{S}^*$

More precisely, starting from a word of $I_\mathcal{S}$, once we have narrowed a word an arbitrary number of times at prefix or suffix position, there is no possibility of reducing the resulting word anywhere else. Unidirectionality is thus a syntactic sufficient condition that guarantees that $\mathcal{N}_\mathcal{S}^*$ is closed under $\mathcal{S}$.

A priori, a distributed algorithm $\mathcal{S}$ does not satisfy the unidirectionality property. However we claim that $\mathcal{S}$ often can be transformed into a simplified system $\mathcal{T}$, which is unidirectional, and such that $Conv(\mathcal{T}, \mathcal{L}) \Rightarrow Conv(\mathcal{S}, \mathcal{L})$. This is done by restricting the possible rules of $\mathcal{S}$ with respect to the mapping φ. Since φ never increases, all the infinite derivations via $\mathcal{S}$ preserve φ after a finite number of initial steps. We construct $\mathcal{T}$ so that $\mathcal{T}$ preserves φ (i.e., $x \to_\mathcal{T} x'$ iff $x \to_\mathcal{S} x' \wedge \varphi(x) = \varphi(x')$). Then clearly $Conv(\mathcal{T}, \mathcal{L})$ holds iff $Conv(\mathcal{S}, \mathcal{L})$ does. The behavior of tokens in $\mathcal{T}$ is more restrained than in $\mathcal{S}$. (In particular, there is no collision of tokens since collision usually eliminates one or two tokens.) So $\mathcal{T}$ is more likely to satisfy the unidirectionality property. As an example, in Sect. 6.1, we derive a simplified system $\mathcal{BD}$ from the original Beauquier-Debas system $\mathcal{S}$ such that $Conv(\mathcal{BD}, \mathcal{L}) \Leftrightarrow Conv(\mathcal{S}, \mathcal{L})$. The original system is not unidirectional (for example, there is a U-turn: $u1\underline{10}v \to_{M_1} u\underline{10}1v \to_{M_1} u0\underline{11}v \to_{M_2} u002v$). In contrast, the simplified system $\mathcal{BD}$ is unidirectional and $\mathcal{N}_\mathcal{BD}^*$ is therefore closed.

Note that the unidirectionality criterion is a sufficient, but not necessary, condition for the closure of $\mathcal{N}_\mathcal{S}$. (There are examples, e.g., Hoepman's algorithm

Fig. 2. Execution-graph for self-stabilizing systems

[18], which are not unidirectional, even in the simplified form, but are still closed.) However, unidirectionality allows us to understand better why our assumption of closure of $\mathcal{N}_{\mathcal{S}}^*$ is met in practice by several mutual exclusion self-stabilizing algorithms, like those of Beauquier-Debas and Ghosh [4,17].

6 Applications

We now show some applications of Theorem 2. Throughout this section, we make three assumptions about $\mathcal{S}$ and $\mathcal{L}$:

α. $\mathcal{L}$ is regular,
β. $\mathcal{L}$ is closed under $\mathcal{S}$,
γ. $\mathcal{S} - Top_{\mathcal{S}}$ is Noetherian.

We first show how to prove self-stabilization, and then a liveness property of termination detection algorithms.

6.1 Proof of Self-Stabilization

Self-stabilization is an important concept in distributed computing, and refers to a system's ability to recover automatically from unexpected faults [14]. Formally, a rewrite system $\mathcal{S}$ is *self-stabilizing* wrt a closed set $\mathcal{L}$ if all infinite derivations via $\mathcal{S}$ reach a configuration of $\mathcal{L}$ for all initial configurations [12]. The configurations of $\mathcal{L}$ are called *legitimate*. The self-stabilization of a system $\mathcal{S}$ wrt $\mathcal{L}$ is generally proven by showing two properties:(1) every ground configuration is reducible via $\mathcal{S}$, and (2) $\mathcal{S}$ converges to $\mathcal{L}$. The former is easy to check: it reduces to an equality test between two regular languages. The main problem is to prove property (2), depicted in Fig. 2.

In self-stabilizing algorithms for rings, the set $\mathcal{L}$ of legitimate configurations is always closed and can often be expressed naturally as a regular set. So (α) and (β) are met in practice. Condition (γ) is generally met by $\mathcal{S}$ as explained in Sect. 4. However the closure of $\mathcal{N}_{\mathcal{S}}^*$ under $\mathcal{S}$ is not always satisfied. In this case, as explained in Sect. 5, we try to find a simplified set $\mathcal{T}$ of rules such that $Conv(\mathcal{T}, \mathcal{L}) \Rightarrow Conv(\mathcal{S}, \mathcal{L})$.

Fig. 3. Execution-graph for $\mathcal{S}$ satisfying $Liv(\mathcal{S}, Term)$ (left) and $\mathcal{S}^{-1}$ satisfying $Conv(\mathcal{S}^{-1}, \neg Term)$ (right)

Example. (*Beauquier-Debas*). The system $\mathcal{S}$ modeling the original Beauquier-Debas algorithm looks as follows:

$$
\begin{array}{ll}
B_1 : 12X \to 21X & T_1 : 0X0 \to 1X2 \\
M_1 : X10Y \to X01Y \ (X \neq \varepsilon) & T_2 : 0X1 \to 1X0 \\
M_2 : X11Y \to X02Y \ (X \neq \varepsilon) & T_3 : 0X2 \to 1X1 \\
M_3 : X12Y \to X00Y \ (X \neq \varepsilon) & T_4 : 2X1 \to 1X2 \\
M_4 : X02Y \to X20Y \ (X \neq \varepsilon) & T_5 : 2X2 \to 1X0 \\
M_5 : X22Y \to X10Y \ (X \neq \varepsilon) &
\end{array}
$$

$\mathcal{L}$ is defined as $20^*10^* \cup 10^*20^*$. One can show that $\mathcal{N}_{\mathcal{S}}^*$ is not closed under $\mathcal{S}$, but as remarked in [4], T_1, T_2, T_3 are applied at most once. As a consequence $\mathcal{S}$ converges to $\mathcal{L}$ iff $\mathcal{S}_0 \equiv \mathcal{S} - \{T_1, T_2, T_3\}$ converges to $\mathcal{L}$. Let φ be a measure mapping every ground word to its number of non-zero elements. Obviously, any application of rule M_2, M_3, M_5 or T_5 of $\mathcal{S}_0$ strictly decreases φ, while rules B_1, M_1, M_4, T_4 preserve it. Therefore, M_2, M_3, M_5 and T_5 can be applied only a finite number of times. It follows that $\mathcal{S}_0$ converges to $\mathcal{L}$ iff $\mathcal{BD} \equiv \{B_1, M_1, M_4, T_4\}$ converges to $\mathcal{L}$. Hence $Conv(\mathcal{BD}, \mathcal{L}) \Rightarrow Conv(\mathcal{S}, \mathcal{L})$. As explained above, we know that $Conv(\mathcal{BD}, \mathcal{L})$. Thus we have convergence of the original algorithm $\mathcal{S}$. Note that classical proof methods require human intervention involving quite subtle well-founded measures (see [4]).

6.2 Proving a Liveness Property of Termination Detection

A distributed algorithm terminates when it reaches a global configuration where no further step is applicable, but some individual machines may be unaware that the computation has terminated. A *termination detection* algorithm is an algorithm that observes the system computation and detects that the computation has reached a terminal configuration of the underlying algorithm (see [25], p.268). In our framework, given a set *Term* of terminal configurations (closed under $\mathcal{S}$), the *liveness* property $Liv(\mathcal{S}, Term)$ of a termination detection algorithm can be stated as follows: once a configuration of *Term* is reached, the algorithm terminates after a finite number of steps. This is depicted to the left in Fig. 3. Formally:

Definition 15. *Let $\mathcal{S}$ be a rewrite system and Term a set of configurations closed under $\mathcal{S}$. We say that $\mathcal{S}$ satisfies the property $Liv(\mathcal{S}, Term)$, if there is no infinite ground derivation starting from an element of Term.*

The method described before does not apply directly for proving $Liv(\mathcal{S}, Term)$, but can be easily adapted as follows. We consider the reverse system $\mathcal{S}^{-1}$, i.e., the system where the right- and left-hand sides of the rules have been exchanged. If we consider derivations via $\mathcal{S}^{-1}$ instead of $\mathcal{S}$, the property is now depicted to the right in Fig. 3. Proving that any execution in *Term* via $\mathcal{S}$ is finite is equivalent to proving that any execution in *Term* via $\mathcal{S}^{-1}$ is finite, i.e., $Conv(\mathcal{S}^{-1}, \neg Term)$. Assuming $(\alpha), (\beta), (\gamma)$ for $\mathcal{S}$ and *Term*, we can use our method to prove $Conv(\mathcal{S}^{-1}, \neg Term)$, hence $Liv(\mathcal{S}, Term)$, since the following conditions hold:

α'. $\neg Term$ is regular (since *Term* is regular),
β'. $\neg Term$ is closed under $\mathcal{S}^{-1}$ (since *Term* is closed under $\mathcal{S}$),
γ'. $\mathcal{S}^{-1} - Top_{\mathcal{S}^{-1}}$ is Noetherian (since $\mathcal{S} - Top_{\mathcal{S}}$ is Noetherian, and $\mathcal{S}$ is length-preserving).

Formally we have:

Theorem 16. *Consider a system $\mathcal{S}$ such that $\mathcal{S} - Top_{\mathcal{S}}$ is Noetherian. Suppose also that Term is regular and closed under $\mathcal{S}$. If $\mathcal{N}^*_{\mathcal{S}^{-1}}$ is closed under $\mathcal{S}^{-1}$ and if $\mathcal{G}_{\mathcal{S}^{-1}} \subseteq \neg Term$, then we have $Liv(\mathcal{S}, Term)$.*

Checking the closure of $\mathcal{N}^*_{\mathcal{S}^{-1}}$ and inclusion of $\mathcal{G}_{\mathcal{S}^{-1}}$ are both decidable properties. In contrast to self-stabilizing algorithms where any configuration is reachable, termination detection algorithms often concern only a restricted set of "admissible" configurations, e.g., configurations with at most one token. Taking this into account, Theorem 3 can be refined by replacing $N^*_{\mathcal{S}^{-1}}$ with $N^*_{\mathcal{S}^{-1}} \cap Adm$ where *Adm* is a regular language of (open) words containing all admissible configurations. This is illustrated in [15] on the termination detection algorithm of Dijkstra-Feijen-van Gasteren (cf., [13,25] and [19]).

7 Implementation and Experimental Results

Our approach has been implemented in SICStus Prolog using the Finite State Automata Utilities of Gertjan van Noord [23]. The resulting program consists of approximately 400 lines of Prolog code (100 clauses) excluding the FSA utilities.

The core of the implementation uses Caucal's algorithm for computing the reachable configurations by prefix and suffix rewriting via systems $\mathcal{S}_{pre}$ (the prefix extensions of $\mathcal{S}$) and $\mathcal{S}_{suf}$ (the suffix extensions of $\mathcal{S}$) given an initial word t_0 of the form $u_0 \odot v_0$. The algorithm boils down to construction of finite automata $A_1(\mathcal{S}_{pre}, t_0)$ and $A_2(\mathcal{S}_{suf}, t_0)$ characterizing the reflexive-transitive closure of $\rightarrow_{\mathcal{S}_{pre}}$ and $\rightarrow_{\mathcal{S}_{suf}}$ applied to $\odot v_0$ and $u_0 \odot$ respectively.

We have run our program on several algorithms drawn from Beauquier-Debas [4], Ghosh [17], Hoepman [18] and Dijkstra-Feijen-van Gasteren [13]. (Ghosh's and Hoepman's systems involve 3-letters rewrite systems instead of 2, but our method extends to them in a natural way.) Running each example takes at most a few seconds on a 200MHz Pentium Pro with 128MB of memory (including various clean-up operations to facilitate human reading of the output). The main code of the implementation is given in [15] as well as typical automata output by the program on the examples.

8 Conclusions

We have given a new procedure, based on prefix and suffix rewriting, for constructing a regular set $\mathcal{G}_\mathcal{S}$ of "unavoidable" configurations of a rewrite system $\mathcal{S}$. This is useful e.g., for proving the convergence $Conv(\mathcal{S}, \mathcal{L})$ of $\mathcal{S}$ to a set $\mathcal{L}$ of configurations. The procedure works under a novel and decidable sufficient condition: the closure of the narrowing language $\mathcal{N}^*_\mathcal{S}$. If $\mathcal{N}^*_\mathcal{S}$ is not closed, we suggest a way to find a simplified form $\mathcal{T}$ of $\mathcal{S}$ which is often closed in practice, and such that $Conv(\mathcal{T}, \mathcal{L}) \Rightarrow Conv(\mathcal{S}, \mathcal{L})$. An implementation exists, and all the examples treated manually in [3], have been processed here for the first time in a uniform automated manner. The method applies only to linear or circular arrays of machines. As an extension it would be interesting to investigate more complicated topologies such as trees, using concepts of tree automata and term rewriting [9] instead of string rewriting. Nevertheless

- We have simplified the framework of [3], giving a statement of the main theorem in a more elegant way.
- We have given a new and significant application of prefix rewriting (analysis of convergence of distributed algorithms) in addition to those given in [6,21].
- Using the idea to reverse the system $\mathcal{S}$, we have shown how to apply the method for proving the liveness of termination detection algorithms over parameterized rings (in addition to the correctness of self-stabilizing algorithms).

It is well-known that proving a property such as convergence of parametric linear array of finite machines is an undecidable problem [2]. Our work can be seen as a step towards isolating sufficient conditions of parametric systems, which are both met in practice and allow to verify some liveness properties mechanically.

Acknowledgement. We are grateful to Moshe Vardi for suggesting the use of prefix rewriting for automatic generation of regular languages.

References

1. P.A. Abdulla, A. Bouajjani, B. Jonsson and M. Nilsson. "Handling global conditions in parameterized system verification". *Proc. CAV'99*, LNCS 1633, Springer-Verlag, 1999, pp. 134-145.
2. K.R. Apt and D.C. Kozen. "Limits for automatic verification of finite-state concurrent systems". *Information Processing Letters 22*, 1986, pp. 307–309.
3. J. Beauquier, B. Bérard, L. Fribourg and F. Magniette. "Proving convergence of self-stabilizing systems using first-order rewriting and regular languages". *Distributed Computing 14:2*, 2001, pp. 83-95.
4. J. Beauquier and O. Debas. "An optimal self-stabilizing algorithm for mutual exclusion on uniform bidirectional rings". *Proc. 2nd Workshop on Self-Stabilizing Systems*, Las Vegas, 1995, pp. 226-239.
5. R.V. Book and F. Otto. *String-Rewriting Systems*. Springer-Verlag, 1993.

6. A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems and P. Wolper. "An Efficient Automata Approach to Some Problems on Context-Free Grammars". *Information Processing Letters* 74 (5-6), 2000, pp. 221-227.
7. A. Bouajjani, B. Jonsson, M. Nilsson and T. Touili. "Regular Model-Checking". *Proc. CAV'00*, LNCS 1855, Springer-Verlag, 2000, pp. 403-418.
8. D. Caucal. "On the regular structures of prefix rewriting". *Proc CAAP'90*, Copenhagen, 1990, LNCS 431, Springer, pp. 87-102.
9. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison and M. Tommasi. *Tree Automata Techniques and Applications.* Available on `http://www.grappa.univ-lille3.fr/tata/`.
10. N. Dershowitz. "Termination of Linear Rewriting Systems". *Proc. ICALP*, LNCS 115, Springer-Verlag, 1981, pp. 448-458.
11. N. Dershowitz and J.-P. Jouannaud. "Rewrite Systems". *Handbook of Theoretical Computer Science*, vol. B, Elsevier - MIT Press, 1990, pp. 243-320.
12. E.W. Dijkstra. "Self-stabilizing systems in spite of distributed control". *Comm. ACM* 17:11, 1974, pp. 643-644.
13. E.W. Dijkstra, W.H.J Feijen and A.J.M. van Gasteren. "Derivation of a Termination Detection Algorithm for Distributed Computations". *Information Processing Letters*, vol. 16, 1983, pp. 217-219.
14. S. Dolev. *Self-Stabilization.* MIT-Press, 2000.
15. M. Duflot, L. Fribourg and U. Nilsson. *Unavoidable Configurations of Parameterized Rings of Processes.* Research Report LSV-00-10, ENS de Cachan, Nov. 2000. (Available on `http://www.lsv.ens-cachan.fr/~fribourg/publis.html`)
16. L. Fribourg and H. Olsén. "Reachability sets of parametrized rings as regular languages". *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of Electronical Notes in Theoretical Computer Science. Elsevier Science, 1997.
17. S. Ghosh. "An Alternative Solution to a Problem on Self-Stabilization". *ACM TOPLAS* 15:4, 1993, pp. 735-742.
18. J.-H. Hoepman. "Uniform Deterministic Self-Stabilizing Ring-Orientation on Odd-Length Rings". *Proc. 8th Workshop on Distributed Algorithms*, LNCS 857, Springer-Verlag. 1994, pp.265-279.
19. B. Jonsson and M. Nilsson. "Transitive closures of regular relations for verifying infinite-state systems". *Proc. TACAS'00*, LNCS 1785, Springer-Verlag, 2000.
20. Y. Kesten, O. Maler, M. Marcuse, A. Pnueli and E. Shahar. "Symbolic Model-Checking with Rich Assertional Languages". *Proc. CAV'97*, LNCS 1254, Springer-Verlag, 1997, pp. 424-435.
21. O. Kupferman and M.Y. Vardi. "An Automata-Theoretic Approach to Reasoning about Infinite-State Systems". *Proc. CAV'00*, LNCS 1855, Springer-Verlag, 2000.
22. D. Lesens, N. Halbwachs and P. Raymond. "Automatic Verification of Parametrized Linear Network of Processes". Proc. *POPL'97*, Paris, 1997.
23. G. van Noord. *Fsa Utilities User Manual Version 5*, 1998.
24. A. Pnueli and E. Shahar. "Liveness and Acceleration in Parameterized Verification". *Proc. CAV'00*, LNCS 1855, Springer-Verlag, 2000, pp. 328-343.
25. G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.
26. P. Wolper and B. Boigelot. "Verifying systems with infinite but regular state spaces". *Proc. CAV'98*, LNCS 1427, Springer-Verlag, 1998, pp. 88-97.

Logic of Global Synchrony

Yifeng Chen[1] and J.W. Sanders[2]

[1] Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester LE1 7RH, UK
[2] Programming Research Group, Oxford University Computing Laboratory, Parks Road, Oxford OX1 3QD, UK

Abstract. An intermediate-level specification notation, Logs, is presented for PRAM/BSP-style programming. It extends pre-post style semantics to reveal state at points of global synchronization *before* termination (if that occurs). The result is an integration of the pre-post, finite and reactive-process styles of specification and in particular an extension of standard BSP. The language is provided with a complete set of laws, formulated to benefit from a simple predicative semantics and to be quite close to programming intuition. The language is compositional, and parallel composition is simply logical conjunction. Use of Logs, and of the laws for reasoning about it, is demonstrated on the problem of the dining philosophers.

1 Introduction

Parallel programs are not only hard to develop in practice but also notoriously difficult to model in theory. That may be part of the reason that no language has gained the acceptance for parallel programming which Dijkstra's guarded-command language has gained for sequential programming. Perhaps the most desirable property of a parallel model is compositionality (in particular parallel modularity) without which it is hard to imagine a language supporting a useful programming methodology. Unfortunately only models based on point-to-point message passing such as CSP [13] and CCS [18] seem to be compositional. Most variable-sharing models such as TLA [15] and UNITY [2] are not; nor are the implementation languages PRAM [11] and (in particular) BSP [16,21,12] which we target, for sake of definiteness, in this paper.

Distributed systems are conveniently specified using global information, although they are efficiently and robustly implemented by relying on only local state and point-to-point communications. The derivation of distributed or parallel algorithms is therefore the task of moving from the former to the latter: the global to the local. We present here an intermediate-level language for describing and reasoning about BSP algorithms abstracted from the global '**put**' and '**get**' commands which BSP uses to implement global synchrony. Since it has been designed to emphasize description (rather than execution, which is the purpose of BSP itself) we call the intermediate language a logic, the *logic of global synchrony*, Logs. Its parallel composition is simply logical conjunction and hence is compositional. That results in good properties including strong refinement

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 487–501, 2001.
© Springer-Verlag Berlin Heidelberg 2001

laws. In short, the language Logs is fully compositional; reasoning largely follows programming intuition; and it integrates the pre/post and reactive styles of programming. Laws for the further transformation from Logs to real BSP programs are covered elsewhere (see [6]).

The language BSP provides a controlled form of synchrony, intermediate in level of abstraction. Communications achieve a global effect at each 'superstep'. As a result BSP supports reactive programming and also a weak form of global synchrony. Nonetheless it is targeted very much towards implementation at the expense of specification and reasoning. Thus it fails to support abstraction in the way the guarded-command language achieves it for sequential programming. In particular BSP offers little modularity, only finite nondeterminism, relies on synchronisation commands using explicit communications, expresses composition as parallel-via-medium, and requires a finite tree of executable commands. The language Logs removes all those restrictions whilst providing a programming theory for *global synchrony*.

The guarded-command language, like any sequential language, is supported by semantics (either relational or predicate-transformer, for example) in which a program is described by the way it transforms initial state to final state. In Logs, in order to encompass reactive programs, we reveal the sequence of intermediate states of a computation at the points of global synchrony. That provides the abstraction necessary to describe the effect of BSP code but frees the programmer from preoccupation with how such synchrony is achieved. That view provides the basic Logs command, a computation with n points of global synchrony. If the computation terminates then the sequence of intermediate states is finite; if it is a reactive nonterminating computation then the sequence is infinite with no final state.

The assertional style, developed originally for sequential programming, has been used with only limited success in the parallel case [20]. For it offers limited support to the programmer for how state should be expected to depend on later, more refined, state through extra ('auxiliary') variables. However we follow the use of a relational model and represent assertions as particular binary relations (which we shall identify as 0Logs commands).

A complicated specification language makes formal reasoning unnecessarily complicated. Here the concern must be that the interposition of intermediate states in a computation has that effect at least on the semantics of Logs. An important feature of our work has been to provide a relatively simple predicative semantics and manner of demonstrating soundness with respect to it of the laws of the notation. That is achieved by the technique of *inheritance* stemming from [14] (see [6]).

In section 2 the basic commands of Logs are defined, their laws given and completeness verified. Important derived operators and their laws are set out in section 3. In section 4 we show Logs, and in particular its derived operators for liveness and safety, at work on the benchmark example of the dining philosophers. A predicative semantics of Logs is enclosed in the appendix.

2 Specification Language Logs

No matter how useful it is, global synchronization is expensive to implement on parallel/distributed systems. There is a case to be made for providing the programmer with a language intermediate in abstraction between the standard low-level language (like BSP, for example, with explicit global synchronizations via the `put` and `get` events with respect to which they are implemented) and a more reified one which abstracts them completely. For a language of the latter type, though important for requirements capture, does not enable a programmer to make tradeoffs based on efficiency; and that is the concern at intermediate stages of design.

In this paper we introduce language Logs. It makes explicit the intermediate global states at synchronization points between start and (possible) termination, in such a way that the programmer is aware of the presence of global synchronizations without having to implement them (at this level). It also enables the programmer to reason, in a simple way, about safety and liveness properties.

On the other hand communications are abstracted in Logs. That is to encourage the programmer to focus on the proposed effect of communications on the global state rather than to decide prematurely which communication commands to use. Logs is thus an intermediate specification language, chosen for convenience, with no pretence at being as abstract as possible.

The semantic space (see appendix A) of Logs is a complete lattice of predicates in which the ordering $\sqsubseteq$ is reverse implication and so corresponds to removal of nondeterminism.

$\top$	magic (top)	$\bot$	chaos (bottom)
$\sqcap$	nondeterministic choice (glb)	$\sqcup$	parallel composition (lub)
$\sqsubseteq$	(refinement) ordering	$\sim$	negation (complement)

For a vector $\boldsymbol{\omega}$ of program variables, the primitives of Logs are commands on $\boldsymbol{\omega}$ taking n steps, for $n \in \mathbb{N} \cup \{\infty\}$. Each command starts in its initial state $\overleftarrow{\boldsymbol{\omega}}$ (pronounced 'pre-$\boldsymbol{\omega}$') and, after n intermediate steps, if $n < \infty$ terminates in a final state $\overrightarrow{\boldsymbol{\omega}}$ (pronounced 'post-$\boldsymbol{\omega}$') but otherwise does not terminate, accumulating forever its infinite sequence of intermediate states (in which case there is no final state).

A typical n-step command is written $\langle\, p \,\rangle_n$ where predicate

$$p = p(\overleftarrow{\boldsymbol{\omega}}, \boldsymbol{\omega}_0, \ldots, \boldsymbol{\omega}_{n-1}, \overrightarrow{\boldsymbol{\omega}})$$

is called the *internal predicate* of the command; in it each $\boldsymbol{\omega}_k$ with $k < n$ denotes the state at the k-th intermediate synchronization point; thus $\boldsymbol{\omega}_0$ records the first synchronisation after the initial state. The set of all n-step commands is written nLogs.

Example: $\langle\, \overleftarrow{x} + 1 = x_0 = \overrightarrow{x} - 1 \,\rangle_1$ is a 1Logs command in which the program variable x is increased by 1 by the time of its intermediate synchronization

point and increased by 1 again by termination. Alternatively we can think of this command as a predicate

$$(\overleftarrow{x}+1 = x_0 = \overrightarrow{x}-1) \ \wedge \ (\# = 1)$$

where $\#$ denotes the number of intermediate synchronizations. Such a simple interpretation aids comprehension but is too weak to provide a fully abstract semantics, which must incorporate more information about the cumulation of traces (see appendix A). □

The sequential composition of P and Q is written $P \fatsemi Q$. Sequential composition is associative and the composition of an nLogs command with an mLogs command forms an $(n{+}m)$Logs command. Using standard notation for variable substitution, the definition becomes the first law.

Law 1 $\quad \langle\, p \,\rangle_n \fatsemi \langle\, q \,\rangle_m = \langle\, \exists \boldsymbol{\omega} \cdot p[\boldsymbol{\omega}/\overrightarrow{\boldsymbol{\omega}}] \wedge q[\boldsymbol{\omega}, \boldsymbol{\omega}_n, \cdots, \boldsymbol{\omega}_{n+m-1}/\overleftarrow{\boldsymbol{\omega}}, \boldsymbol{\omega}_0, \cdots, \boldsymbol{\omega}_{m-1}] \,\rangle_{n+m}$

Example: The following command is a composition of a 0Logs command without any synchronization and a 1Logs command with exactly one synchronization point:

$$\langle\, \overleftarrow{x} = \overrightarrow{x} + 1 \,\rangle_0 \fatsemi \langle\, \overleftarrow{x} + 1 = x_0 = \overrightarrow{x} - 1 \,\rangle_1 \ = \ \langle\, \overleftarrow{x} = x_0 = \overrightarrow{x} - 1 \,\rangle_1$$

which is indeed the relational composition of its internal predicates. The final state of the first Logs command is linked to the initial state of the second and the interface is then hidden. □

The nondeterministic choice between two nLogs commands is the disjunction of their internal predicates. That implies monotonicity of the embedding which takes a predicate to an nLogs command.

Law 2 $\quad (1)\ \langle\, p \,\rangle_n \sqcap \langle\, q \,\rangle_n = \langle\, p \vee q \,\rangle_n \qquad (2)\ \langle\, p \,\rangle_n \sqsubseteq \langle\, q \,\rangle_n \quad \text{iff} \quad q \Rightarrow p$

The parallel composition of two nLogs commands is the conjunction of their internal predicates, while that between two nLogs commands of different lengths is the magic command, which contains no behaviour.

Law 3 $\quad (1)\ \langle\, p \,\rangle_n \sqcup \langle\, q \,\rangle_n = \langle\, p \wedge q \,\rangle_n \qquad (2)\ \langle\, p \,\rangle_n \sqcup \langle\, q \,\rangle_m = \top \qquad (m \neq n)$

Example: Laws 3 and 2 can be easily understood in terms of our informal notation: $\langle\, p \,\rangle_n = (p \wedge \# = n)$ and $\langle\, q \,\rangle_m = (q \wedge \# = m)$. Non-deterministic choice is disjunction. If $n = m$ then the commands are merged into a single nLogs command $(p \vee q) \wedge \# = n$; otherwise, they become a disjunction $(p \wedge \# = n) \vee (q \wedge \# = m)$, which cannot be further simplified. Parallel composition is conjunction. If $n = m$ then the commands are merged into a single nLogs command $p \wedge q \wedge \# = n$; otherwise, their conjunction becomes 'magic' or false. This informal interpretation is not our formal semantics which models sequential composition and recursion properly, and thus needs further healthiness conditions (refer to appendix A). □

Negation forms a complement. The negation of an nLogs command can be calculated inductively by construction of the command. De Morgan's laws hold for negation of (arbitrary) disjunctions and conjunctions so that either might have been introduced as a derived construct. In the following law $i, n : \mathbb{N} \cup \{\infty\}$.

Law 4 (1) $P \sqcup \sim P = \top$ (2) $P \sqcap \sim P = \bot$
(3) $\sim\sim P = P$ (4) $\sim \langle\, p \,\rangle_n = \langle\, \neg p \,\rangle_n \sqcap \bigsqcap_{i \neq n} \langle\, \mathtt{true} \,\rangle_i$
(5) $\sim \bigsqcap \mathcal{P} = \bigsqcup \{\sim P \mid P \in \mathcal{P}\}$ (6) $\sim \bigsqcup \mathcal{P} = \bigsqcap \{\sim P \mid P \in \mathcal{P}\}$

Finite commands in Logs have a normal form: $\bigsqcap_{n \leqslant \infty} \langle\, q_n \,\rangle_n$. A proof of the following theorem appears in [5].

Theorem 1 (Completeness) *The laws of* Logs *are complete for finite* Logs *commands: semantic equality between two finite* Logs *commands is provable using (just) the laws of* Logs *and those of first-order logic.*

3 Derived Logs Commands and Their Algebraic Laws

The derived commands studied in this section are widely applicable and support modularised specification and derivation in Logs.

The extreme elements of the semantic lattice are themselves useful specifications. Chaos $\bot$ specifies a command with all possible behaviours. Magic $\top$ specifies a command without any behaviour!

Two further important commands are complements:

$\rhd$	command with all terminating behaviours
$\lhd$	command with all nonterminating behaviours

that is, $\rhd \sqcap \lhd = \bot$, $\rhd \sqcup \lhd = \top$ and $\rhd = \sim\lhd$. In terms of nLogs commands, we have the following laws (in which $n : \mathbb{N} \cup \{\infty\}$).

Law 5 (1) $\bot = \bigsqcap_{k \leqslant \infty} \langle\, \mathtt{true} \,\rangle_k$ (2) $\top = \langle\, \mathtt{false} \,\rangle_n$
(3) $\rhd = \bigsqcap_{k < \infty} \langle\, \mathtt{true} \,\rangle_k$ (3) $\lhd = \langle\, \mathtt{true} \,\rangle_\infty$

In particular $\top = \bigsqcap_{n \leqslant \infty} \langle\, \mathtt{false} \,\rangle_n$. The (sequential) interactions between the extreme commands are shown in Table 1.

Table 1. Interactions between extreme commands

$P \fatsemi Q$	$\top$	$\bot$	$\rhd$	$\lhd$
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\lhd$	$\bot$	$\bot$	$\lhd$
$\rhd$	$\top$	$\bot$	$\rhd$	$\lhd$
$\lhd$	$\lhd$	$\lhd$	$\lhd$	$\lhd$

The next few derived commands correspond to code.

`skip`	skip, no operation	$(b)_\top$	conditional magic
$\lhd\, b \,\rhd$	binary conditional		

Command `skip` and conditional magic $(b)_\top$ are special 0Logs commands:

$$\texttt{skip} \mathrel{\widehat{=}} \langle\, \overleftarrow{\boldsymbol{\omega}} = \overrightarrow{\boldsymbol{\omega}} \,\rangle_0 \quad \text{and} \quad (b(\boldsymbol{\omega}))_\top \mathrel{\widehat{=}} \langle\, b(\overleftarrow{\boldsymbol{\omega}}) \wedge \overleftarrow{\boldsymbol{\omega}} = \overrightarrow{\boldsymbol{\omega}} \,\rangle_0 \,.$$

Binary conditional can be derived from skip and conditional magic as usual:

$$P \lhd b \rhd Q \;\mathrel{\widehat{=}}\; ((b)_\top \fatsemi P) \sqcap ((\neg b)_\top \fatsemi Q) \,.$$

If b is true in the initial state then P is executed; otherwise Q is executed. Conditional magic satisfies the following straightforward laws in which $m{<}\infty$.

Law 6 (1) $(b)_\top \fatsemi \langle\, p \,\rangle_n = \langle\, b(\overleftarrow{\boldsymbol{\omega}}) \wedge p \,\rangle_n$ (2) $(a)_\top \fatsemi (b)_\top = (a \wedge b)_\top$
(3) $\langle\, p \,\rangle_m \fatsemi (b)_\top = \langle\, p \wedge b(\overrightarrow{\boldsymbol{\omega}}) \,\rangle_m$

Several repetition commands are of use:

Table 2. Recursion, iteration and repetition

μf	recursion	`do` $b \to P$ `od`	iteration
P^n	repetition n times	P^∞	infinite repetition
P^*	arbitrary repetition	$P^\circledast$	finite repetition
P^+	non-zero repetition	$P^\oplus$	non-zero finite repetition

The modelling of recursion is subtle enough for most existing computational models to simplify it by disallowing unguarded recursions [9], disallowing non-tail recursions, disallowing reactive or real-time behaviors [19], disallowing command `skip` (or `skip`, the unit of sequential composition) [1], or disallowing the body of a loop to take zero time [8].

To integrate pre-post, finite and unbounded specifications, we cannot afford those compromises. We believe that the calculation of the fixpoint of a recursive program should start from nontermination not chaos:

$$\lhd,\; f(\lhd),\; f^2(\lhd),\; \cdots,\; f^\kappa(\lhd),\; \cdots$$

If $\lhd \sqsubseteq f(\lhd)$ with regard to some ordering, the above sequence eventually reaches a fixpoint; otherwise a more general technique called the *strongest negative fixpoint* [7] is required.

Iteration `do` $b \to P$ `od` is a kind of recursion $\mu X \cdot (P \fatsemi X) \lhd b \rhd$ `skip` on which repetitions can be defined:

$$P^0 \;\mathrel{\widehat{=}}\; \texttt{skip}, \quad P^{n+1} \;\mathrel{\widehat{=}}\; P \fatsemi P^n \quad \text{and} \quad P^\infty \;\mathrel{\widehat{=}}\; \texttt{do true} \to P \texttt{ od}\,.$$

Arbitrary repetition is defined by $\bigsqcap_n P^n$, finite repetition by $\bigsqcap_{n<\infty} P^n$, non-zero repetition by $\bigsqcap_{0<n\leqslant\infty} P^n$, and non-zero finite repetition by $\bigsqcap_{0<n<\infty} P^n$.

Example: $\langle\, \overleftarrow{x} + 1 = x_0 = \overrightarrow{x} \,\rangle_1^\infty$ specifies a nonterminating reactive command that increases variable x by 1 at every synchronization point. □

The repetition operators satisfy some laws (in which $\lambda, \mu : \{n, \infty, *, \circledast, +, \oplus\}$).

Law 7
(1) $P^* = P^\circledast \sqcap P^\infty$ (2) $P^+ = P^\oplus \sqcap P^\infty$
(3) $P^\infty \fatsemi Q = P^\infty$ (4) $P^\lambda \fatsemi P^\mu = P^\mu \fatsemi P^\lambda$
(5) $P^\circledast \fatsemi P^\circledast = P^\circledast$ (6) $P^\oplus \fatsemi P^\circledast = P^\oplus$
(7) $P^* \fatsemi P^* = P^\circledast \fatsemi P^* = P^*$
(8) $P^+ \fatsemi P^* = P^+ \fatsemi P^\circledast = P^\oplus \fatsemi P^* = P^+$
(9) $(P \sqcap Q)^\infty = P^* \fatsemi Q \fatsemi (P \sqcap Q)^\infty$

An important kind of safety property states that $p(\overleftarrow{\boldsymbol{\omega}}, \boldsymbol{\omega}_0, \overrightarrow{\boldsymbol{\omega}})$ holds for every synchronization point. Such a safe computation simply becomes the arbitrary repetition, $\bigsqcap_{k\leqslant\infty} \langle\, p \,\rangle_1^k$, of a 1Logs specification. In most applications, it is more convenient to assume a stable state *before* each synchronization point. That leads us to the definition of a special 1Logs command called a *transition*: $[\, p(\overleftarrow{\boldsymbol{\omega}}, \overrightarrow{\boldsymbol{\omega}}) \,] \;\widehat{=}\; \langle\, p(\overleftarrow{\boldsymbol{\omega}}, \overrightarrow{\boldsymbol{\omega}}) \wedge \overleftarrow{\boldsymbol{\omega}} = \boldsymbol{\omega}_0 \,\rangle_1$.

An always-true safety property is defined: $|\, p \,| \;\widehat{=}\; \bigsqcap_{k\leqslant\infty} [\, p \,]^k$.

Example: $|\, \overleftarrow{x} = \overrightarrow{x} \,| \sqcup \langle\, \overleftarrow{y} + 1 = \overrightarrow{y} \,\rangle_2$ is a 2Logs command that increases variable y by 1 while keeping variable x unchanged. □

A typical kind of liveness property is a *terminating pre-post specification* whose final state is related to its initial state by an internal predicate after a finite number of synchronizations: $[\![\, q(\overleftarrow{\boldsymbol{\omega}}, \overrightarrow{\boldsymbol{\omega}}) \,]\!] \;\widehat{=}\; \bigsqcap_{n<\infty} \langle\, q(\overleftarrow{\boldsymbol{\omega}}, \overrightarrow{\boldsymbol{\omega}}) \,\rangle_n$.

Example: Computation $[\![\, \overleftarrow{x} + 1 = \overrightarrow{x} \,]\!]$ terminates after finitely many steps and eventually increases the value of x by 1. The intermediate states of this computation are 'chaotic', and no useful information can be extracted by observing them. □

Safety property $|\, p \,|$ distributes various structures consisting of transitions. In the following laws $\lambda \in \{n, \infty, *, \circledast, +, \oplus\}$, $P = P \sqcup |\, \mathtt{true} \,|$ and $Q = Q \sqcup |\, \mathtt{true} \,|$.

Law 8
(1) $|\, p \,| \sqcup \rhd = [\, p \,]^\circledast$ (2) $|\, p \,| \sqcup \lhd = [\, p \,]^\infty$
(3) $|\, p \,| \sqcup [\, q \,] = [\, p \wedge q \,]$ (4) $|\, p \,| \sqcup \langle\, q \,\rangle_0 = \mathtt{skip} \sqcup \langle\, q \,\rangle_0$
(5) $|\, p \,| \sqcup Q^\lambda = (\, |\, p \,| \sqcup Q \,)^\lambda$ (6) $|\, p \,| \sqcup |\, q \,| = |\, p \wedge q \,|$
(7) $|\, p \,| \sqcup (\, P \fatsemi Q \,) = |\, p \,| \sqcup P \fatsemi |\, p \,| \sqcup Q$
(8) $(\, \rhd \fatsemi |\, p \,| \,) \sqcup (\, \rhd \fatsemi Q^\infty \,) = (\, \rhd \fatsemi |\, p \,| \sqcup Q^\infty \,)$

The parallel composition of pre-post commands becomes a conjunction of their internal predicates, while a nondeterministic choice becomes a disjunction.

Law 9 (1) $[\![\, p \,]\!] \sqcup [\![\, q \,]\!] = [\![\, p \wedge q \,]\!]$ (2) $[\![\, p \,]\!] \sqcap [\![\, q \,]\!] = [\![\, p \vee q \,]\!]$

Reactive commands satisfy the following laws.

Law 10 (1) $\rhd \fatsemi [p(\overleftarrow{\boldsymbol{\omega}})]^\infty = \rhd \fatsemi [p(\overrightarrow{\boldsymbol{\omega}})]^\infty$
(2) $\rhd \fatsemi [p]^\infty \sqcup \rhd \fatsemi [q]^\infty = \rhd \fatsemi [p \wedge q]^\infty$

A computation satisfies a rely-guarantee $P \rightrightarrows Q$ iff whenever P is satisfied Q is guaranteed $\quad P \rightrightarrows Q \mathrel{\widehat{=}} {\sim}P \sqcap Q$.

Thus $P \rightrightarrows Q$ allows any computation that does not satisfy P, in which case it may not guarantee Q. This corresponds to the reply-guarantee specifications in TLA [15] and UNITY [2]. A rely-guarantee specification satisfies the laws:

Law 11 (1) $P \rightrightarrows P = \bot$ (2) $P \sqcup (P \rightrightarrows Q) = P \sqcup Q$

4 Case Study: The Dining Philosophers

4.1 Dining Philosophers

Since it was first described in [10], the example of the dining philosophers has become a benchmark for the calibration of theories of concurrency and the way they facilitate reasoning about resource contention. Five philosophers are seated at a circular dining table. Each philosopher cycles through the phases of *thinking*, **t**, being *hungry*, **h**, and *eating*, **e**. Neighbouring philosophers may not eat at the same time. We require that a hungry philosopher eventually eats provided that thinking and eating are achieved in finitely-many steps.

Let each philosopher have state $x_k \in \{\mathbf{t}, \mathbf{h}, \mathbf{e}\}$ and let the state of the vector of philosophers be $x \mathrel{\widehat{=}} x_0, x_1, \ldots, x_4$. The initial state of the system of philosophers is specified as follows, with $k^- \mathrel{\widehat{=}} (k{-}1)\ mod\ 5$ and $k^+ \mathrel{\widehat{=}} (k{+}1)\ mod\ 5$.

$$\textbf{SPEC 1} \quad \widehat{=} \quad \bigsqcup_k \left(\begin{array}{l} [\,\overleftarrow{x}_k = \mathbf{t}\,]^\oplus \fatsemi \\ [\,\overleftarrow{x}_k = \mathbf{h}\,]^\oplus \fatsemi \\ [\,\overleftarrow{x}_{k^-} \neq \overleftarrow{x}_k = \mathbf{e} \neq \overleftarrow{x}_{k^+}\,]^\oplus \end{array} \right)^\infty$$

The term $[\,\overleftarrow{x}_k = \mathbf{h}\,]^\oplus$ specifies termination of philosopher k's hungry phase.

4.2 Forks

The specification is to be refined by a distributed design in which contention is mediated by forks, one between adjacent philosophers and numbered like the philosophers: philosopher k requires forks k and k^+ in order to eat. Let the state of fork k be denoted y_k with the vector of forks being denoted $y \mathrel{\widehat{=}} y_0, y_1, \ldots, y_4$. Each fork has two states: either **l** (being used by the philosopher to its left) or **r** (to its right). Thus to eat, philosopher k requires $y_k = \mathbf{r}$ and $y_{k^+} = \mathbf{l}$.

$$\textbf{SPEC 2} \quad \widehat{=} \quad \bigsqcup_k \left(\begin{array}{l} [\,\overleftarrow{x}_k = \mathbf{t}\,]^\oplus \fatsemi \\ [\,\overleftarrow{x}_k = \mathbf{h}\,]^\oplus \fatsemi \\ [\,\overleftarrow{x}_k = \mathbf{e}\,]^\oplus \end{array} \right)^\infty \sqcup \mid \mathit{forks} \mid$$

$$\mathit{where}\ \ \mathit{forks} \quad \widehat{=} \quad \forall k \cdot (\overleftarrow{x}_k = \mathbf{e} \ \Rightarrow\ \overrightarrow{y}_k = \mathbf{r} \wedge \overrightarrow{y}_{k^+} = \mathbf{l})$$

The validity of that design is ensured as follows.

Theorem 2 SPEC *1* $\sqsubseteq$ SPEC *2*

Proof.
SPEC 2
$=$ Law 8(3)(5)(6)(7)

$$\bigsqcup_k \left(\begin{array}{l} [\,\overleftarrow{x}_k = \mathbf{t} \wedge \mathit{forks}\,]^{\oplus} \fatsemi \\ [\,\overleftarrow{x}_k = \mathbf{h} \wedge \mathit{forks}\,]^{\oplus} \fatsemi \\ [\,\overleftarrow{x}_k = \mathbf{e} \wedge \overrightarrow{y}_k = \mathbf{r} \wedge \overrightarrow{y}_{k^+} = \mathbf{l}\,]^{\oplus} \end{array} \right)^{\infty}$$

$\sqsupseteq$ monotonicity of various compositions
SPEC 1 □

4.3 A Strategy

In this design the forks are resources and the philosophers are (resource) consumers. The thinking phase represents a period during which a consumer needs no shared resource; the hungry phase represents a period of waiting for required resources; the eating phase represents a resource-consuming period. Thus termination of both thinking and eating must be guaranteed by each resource consumer, whilst termination of the hungry phase has to be guaranteed by a distributed implementation.

Initially we suppose that termination of the hungry phase is unknown. We propose the following strategy for each philosopher k and later prove it to terminate:

1. a thinking philosopher may either continue thinking or become hungry;
   $(\overleftarrow{x}_k = \mathbf{t}) \Rightarrow (\overrightarrow{x}_k = \overleftarrow{x}_k \vee \overrightarrow{x}_k = \mathbf{h})$
2. a hungry philosopher may either remain hungry or immediately eat, provided two adjacent forks are available;
   $(\overleftarrow{x}_k = \mathbf{h}) \Rightarrow (\texttt{if}\ \ \overleftarrow{y}_k = \mathbf{r} \wedge \overleftarrow{y}_{k^+} = \mathbf{l}\ \ \texttt{then}\ \ \overrightarrow{x}_k = \mathbf{e}\ \ \texttt{else}\ \ \overrightarrow{x}_k = \overleftarrow{x}_k)$
3. an eating philosopher may either continue eating or stop to think;
   $(\overleftarrow{x}_k = \mathbf{e}) \Rightarrow (\overrightarrow{x}_k = \overleftarrow{x}_k \vee \overrightarrow{x}_k = \mathbf{t})$
4. if two adjacent philosophers are thinking, the fork between them will not change direction;
   $(\overleftarrow{x}_{k^-} = \mathbf{t} = \overleftarrow{x}_k) \Rightarrow (\overleftarrow{y}_k = \overrightarrow{y}_k)$
5. if a philosopher is thinking while his left-hand neighbour is not, then the philosopher will 'lose' the fork between them;
   $(\overleftarrow{x}_{k^-} \neq \mathbf{t} = \overleftarrow{x}_k) \Rightarrow (\overrightarrow{y}_k = \mathbf{l})$
6. if a philosopher is not thinking but his left-hand neighbour is, the neighbour will lose the fork between them;
   $(\overleftarrow{x}_{k^-} = \mathbf{t} \neq \overleftarrow{x}_k) \Rightarrow (\overrightarrow{y}_k = \mathbf{r})$
7. if neither of two adjacent philosophers is thinking, the fork between them will not change direction.
   $(\overleftarrow{x}_{k^-} \neq \mathbf{t} \neq \overleftarrow{x}_k) \Rightarrow (\overleftarrow{y}_k = \overrightarrow{y}_k)$

That strategy is formalised as follows.

SPEC 3 $\widehat{=}$ $(acyc)_\top \fatsemi \bigsqcup_k P_k$ *where*

$$\begin{array}{rcl} P_k & \widehat{=} & \left(T_k^\oplus \fatsemi H_k^+ \fatsemi E_k^\oplus\right)^\infty \\ T_k & \widehat{=} & [\,\overleftarrow{x}_k = \mathbf{t} \wedge strategy_k\,] \\ H_k & \widehat{=} & [\,\overleftarrow{x}_k = \mathbf{h} \wedge strategy_k\,] \\ E_k & \widehat{=} & [\,\overleftarrow{x}_k = \mathbf{e} \wedge strategy_k\,] \end{array}$$

and $strategy_k$ *is the conjunction of the seven strategies just introduced.*

P_k represents philosopher k, while T_k, H_k and E_k represent thinking, hungry and eating respectively. SPEC 3 is initialised by the requirement *acyc* that the forks form an *acyclic* priority graph (whose formalisation is standard). The following lemma states that the fork graph is always acyclic. (In the case of improper initialization, SPEC 3 becomes miraculous and so still refines SPEC 2.)

Lemma 3 (Acyclic safety) SPEC $3 \sqsupseteq [\,\overleftarrow{acyc}\,]^\infty$

The proof (see [5]) shows that $[\,\overleftarrow{acyc}\,]^\infty$, established by initialisation, is maintained by $\bigwedge_k strategy_k$.

4.4 Liveness

To reason about liveness we first decompose each philosopher k in two:

$$\begin{array}{rcl} L_k & \widehat{=} & \left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus\right)^\infty \\ D_k & \widehat{=} & \left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus\right)^\circledast \fatsemi T_k^\oplus \fatsemi H_k^\infty\,. \end{array}$$

We think of L_k as denoting a 'living' philosopher and D_k as denoting a 'dying' one who eventually remains hungry forever. For each philosopher life or death is a nondeterministic choice! Indeed:

$$\begin{array}{ll} & P_k \\ = & \qquad \text{definition of } P_k \\ & \left(T_k^\oplus \fatsemi H_k^+ \fatsemi E_k^\oplus\right)^\infty \\ = & \qquad \text{Law 7(2)} \\ & \left(T_k^\oplus \fatsemi (H_k^\oplus \sqcap H_k^\infty) \fatsemi E_k^\oplus\right)^\infty \\ = & \qquad \text{distributivity of } \sqcap \text{ and Law 7(3)} \\ & \left((T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus) \sqcap (T_k^\oplus \fatsemi H_k^\infty)\right)^\infty \\ = & \qquad \text{Law 7(9)(3)} \\ & \left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus\right)^* \fatsemi T_k^\oplus \fatsemi H_k^\infty \\ = & \qquad \text{Law 7(1)(3)} \\ & \left(\left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus\right)^\circledast \fatsemi T_k^\oplus \fatsemi H_k^\infty\right) \sqcap \left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus\right)^\infty \\ = & \qquad \text{definitions of } D_k \text{ and } L_k \\ & D_k \sqcap L_k\,. \end{array}$$

Now we observe that SPEC 3 is refined by a *group* of living philosophers:

SPEC 4 $\quad \widehat{=} \quad (acyc)_\top \fatsemi \bigsqcup_k L_k$.

In fact SPEC 4 equals SPEC 3 because the strategy guarantees that every philosopher lives. To prove that, we need to eliminate the possibility of deadlock or starvation. Lemma 4 excludes the possibility that any philosopher dies by waiting for a fork from a neighbour (i.e. deadlock), whilst Lemma 6 excludes that of starvation.

Lemma 4 (Deadlock freedom) $\quad (acyc)_\top \fatsemi \bigsqcup_k D_k = \top$

The philosophers are not all dead; otherwise all philosophers would become hungry forever after some point, which conflicts with our strategies and the safety property that the directions of the forks are loop free.

Proof.
$(acyc)_\top \fatsemi \bigsqcup_k D_k$
$=$ definition of D_k
$(acyc)_\top \fatsemi \bigsqcup_k \left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus \right)^\circledast \fatsemi T_k^\oplus \fatsemi H_k^\infty$
$\sqsupseteq$ monotonicity of $[\cdot]^\infty$ and termination of $\left(T_k^\oplus \fatsemi H_k^\oplus \fatsemi E_k^\oplus \right)^\circledast \fatsemi T_k^\oplus$
$(acyc)_\top \fatsemi \bigsqcup_k [\, strategy_k \,]^\infty \;\sqcup\; (acyc)_\top \fatsemi \bigsqcup_k (\rhd \fatsemi [\, \overleftarrow{x}_k = \mathbf{h} \,]^\infty)$
$\sqsupseteq$ Lemma 3 and Law 6(1)
$[\, strategy \wedge \overleftarrow{acyc} \,]^\infty \sqcup (\rhd \fatsemi [\, \overleftarrow{x} = \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \,]^\infty)$
$\sqsupseteq$ monotonicity of $[\cdot]^\infty$
$[\, \overleftarrow{x} = \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \Rightarrow \overrightarrow{x} \neq \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \,]^\infty \sqcup (\rhd \fatsemi [\, \overleftarrow{x} = \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \,]^\infty)$
$\sqsupseteq$ weakening
$(\rhd \fatsemi [\, \overrightarrow{x} \neq \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \,]^\infty) \sqcup (\rhd \fatsemi [\, \overleftarrow{x} = \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h}, \mathbf{h} \,]^\infty)$
$=$ Law 10(1)(2), Law 5(2) and $(\rhd \fatsemi \top = \top)$ in Table 1
$\top$.
□

The following lemma (proved in [5]) states that a living philosopher is unable to dine with a dying neighbour.

Lemma 5 $\quad L_{k^-} \sqcup D_k = \top$

We therefore infer that at least one philosopher is living and no living philosopher can have a dying neighbour. Consequently all philosophers are living.

Lemma 6 (Starvation freedom) $\quad$ SPEC *3* $=$ SPEC *4*

Proof.
SPEC 3
$=$ definition of P_k
$(acyc)_\top \fatsemi \bigsqcup_k (L_k \sqcap D_k)$
$=$ $\sqcap$ distributes through $\sqcup$
$(acyc)_\top \fatsemi (\bigsqcup_k L_k) \sqcap (D_0 \sqcup L_1 \sqcup \cdots \sqcup L_4)$
$\qquad\qquad\qquad\quad \sqcap \;\cdots\cdots$
$\qquad\qquad\qquad\quad \sqcap \;(L_0 \sqcup D_1 \sqcup \cdots \sqcup D_4) \sqcap (\bigsqcup_k D_k)$

$=$ lemma 4 and lemma 5
$(acyc)_\top \fatsemi \bigsqcup_k L_k \sqcap \top \sqcap \top$
$=$ $\top$ is the top of the complete-lattice space.
SPEC 4

□

In fact specification SPEC 4 is very close to a BSP program. For the derivation of such a program from SPEC 4 we refer to [6]. Here it remains to check the validity of SPEC 3 with respect to SPEC 1. But by lemma 6 that follows from the observation that SPEC 3 refines the safety property $|\,forks\,|$:

Theorem 7 SPEC *2* $\sqsubseteq$ SPEC *3*

5 Conclusions and Acknowledgements

This paper has introduced a specification language, Logs, which supports the design of parallel programs based on global synchronizations. It integrates, in one simple language, specifications of pre-post, finite and infinite reactive processes in a compositional program-like style. Examples of targeted parallel implementation languages include PRAM and BSP.

The most significant property of Logs is its compositionality (or parallel modularity), which appears to be absent from other variable-sharing based models. Lack of parallel modularity has probably been the main difficulty of BSP programming. In Logs parallel composition is simply logical conjunction. That may be simplistic in terms of implementation; however from the viewpoint of specification, it proves to be rather powerful and, most importantly, to guarantee compositionality. Although in this paper we take program derivation only as far as Logs, in [6] we also introduce refinement laws, concerning variable protection, to transform compositional Logs specifications into BSP programs.

The benchmark problem of the dining philosophers has been studied. The solution is genuinely distributed and its freedom from livelock is guaranteed. Furthermore the cycle of five philosophers is easily extended to more general topologies; and it may be modified to allow a philosopher to remain thinking forever (see [6]). Although that study is not the ideal vehicle for demonstrating the conversion of global synchronisation to communications, it does show that BSP is now suitable for MIMD programming. Our refinement and solution appear encouragingly simple compared with that in say UNITY [2]. The reasoning seems to be closer to programming intuition. This is mainly because of Logs's compositionality, global synchronization mechanism and its program-like style.

That also distinguishes it from standard temporal logic, although the temporal operators are represented readily in Logs. Synchronisation in Logs results in a restricted form of fairness, one reason for its strong laws. In this paper we have found no need to exploit the temporal operators, though both 'must' $\Box$ and 'may' $\Diamond$ are readily defined in Logs:

$$\Diamond P \mathrel{\widehat{=}} \rhd \fatsemi P \fatsemi \bot \qquad \Box P \mathrel{\widehat{=}} \sim \Diamond \sim P\,.$$

In the more general language Temporal Logic of Actions [15], an action describes a transition similarly to 0Logs, while here P may contain more than one step.

It can no longer be sustained that BSP is suitable only for SPMD and SIMD data-parallelism but not MIMD programming. Such unsuitability was previously seen as being due to the lack of a rigorous top-down programming methodology. Our derivation technique differs substantially from previous work on BSP, allows synchronizations in the body of a loop, and therefore fully supports MIMD parallel programming.

This paper has benefited from wide-ranging comments from its referees to whom we are grateful.

References

1. J.A. Bergstra and J.W. Klop, Algebra of communicating processes with abstraction, *Theoretical Computer Science*, **37**(1): 77-121, 1985.
2. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
3. Y. Chen, How to write a healthiness condition, *2nd International Conference on Integrated Formal Methods*, LNCS, **1945**:299-317, Springer-Verlag, 2000.
4. Y. Chen and J.W. Sanders, Weakest specifunctions for BSP, *Parallel Processing Letters* (to appear) 2001.
5. Y. Chen and J.W. Sanders, Logic of global synchrony, *Technical Report*, RR-01-01, (`http://web.comlab.ox.ac.uk/oucl/publications/tr/index.html`), Oxford University Computing Laboratory, 2001.
6. Y. Chen, *Formal Methods for Global Synchrony*, D.Phil. Thesis, Oxford University Computing Laboratory, 2001.
7. Y. Chen, Fixpoints of non-monotonic functions, *Technical Report* 29, (`http://www.mcs.le.ac.uk/techreports/2001/tr-2001-29.ps.gz`), Department of Mathematics and Computer Science, University of Leicester, 2001.
8. J. Davies and S. Schneider, A brief history of Timed CSP, *Theoretical Computer Science*, **138**: 243-271, 1995.
9. E.W. Dijkstra, Guarded commands, non-determinacy and the formal derivation of programs, *Communications of the ACM*, **18**: 453-457, 1975.
10. E.W. Dijkstra, Two starvation free solutions to general exclusion problem, EWD 625, plataanstraat 5, 5671 Al Nuenen, The Netherlands, 1978.
11. S. Fortune and J. Wyllie, Parallelism in random access machines. *Proc. 10th Annual ACM Symposium on Theory of Computing*: 114-118, 1978.
12. M.W. Goudreau et al. A proposal for the BSP world-wide standard library (preliminary version), 1996.
13. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
14. C.A.R. Hoare and J. He, *Unifying Theories of Programming*, Prentice Hall, 1998.
15. L. Lamport, A temporal logic of actions, *ACM Transctions on Programming Languages and Systems*, **16**(3): 872-923, 1994.
16. D.S. Lecomber, *Methods of* BSP *Programming*, Oxford University Computing Laboratory DPhil. thesis, 1998.
17. W.F. McColl, Scalability, portability and predictability: The BSP approach to parallel programming, *Future Generation Computer Systems* **12**: 265-272, 1996.
18. R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

19. G. Nelson, A generalization of Dijkstra's calculus, *ACM Transactions on Programming Languages and Systems*, **11**(4):517-561, 1989.
20. S.S. Owicki and D. Gries, An axiomatic proof technique for parallel programs I, *Acta Informatica*, **6**(4): 319-340, 1976.
21. D.B. Skillicorn, Building BSP programs using the refinement calculus, *Formal Methods for Parallel Programming and Applications*, IPPS/SPDP'98, 1998.

A Predicative Semantics of Logs

In this appendix we present a predicative semantics for Logs with respect to which the laws provided in this paper are sound, and exhibit healthiness conditions which characterise Logs commands in that semantic space. Proofs and further details appear in [6].

Let ω denote the vector of all program (or system) variables. For each logical variable x in ω its initial state is (also) denoted x, its final state is denoted x', its initial trace record is denoted $x.tr$ and its final trace record is denoted $x.tr'$ $(= (x.tr)')$. For brevity we adopt these conventions:

ω' denotes the vector of dashed variables x' for x in ω.

τ denotes the vector of trace variables $x.tr$ for x in ω.

τ' denotes the vector of trace variables $x.tr'$ for x in ω.

A Logs command can be expressed as a predicate with four free variables $\omega, \omega', \tau, \tau'$. However not every such predicate represents a Logs command. The following four *healthiness conditions*, presented here in the style of [14], must hold. (For a detailed description of healthiness conditions in this context we refer to [3]. The conditions here arise from a more general unifying theory called *cumulative computing* [6].)

The first healthiness condition states that if a command starts after a failure or a nonterminating command, it becomes unobservable: if the initial trace record is infinite then the predicate is arbitrary. Writing $|\tau|$ for the length of sequence τ, that is expressed $A = (|\tau| \neq \infty \Rightarrow A)$, or equivalently:

Healthiness condition: $H_0(A) \mathrel{\widehat{=}} A = (A \vee |\tau| = \infty)$.

The trace record of Logs is monotonic: a command retains the trace before its start but may append to it. Writing $\tau \preccurlyeq \tau'$ to mean that sequence τ is a prefix of sequence τ', that is expressed:

Healthiness condition: $H_\tau(A) \mathrel{\widehat{=}} A = (A \wedge \tau \preccurlyeq \tau')$.

The final state of a nonterminating Logs command is not observable. $|\tau'|$ denotes the length of the trace record at the end of the computation.

Healthiness condition: $H_\infty(A) \mathrel{\widehat{=}} A = (A \vee A \fatsemi (|\tau'| = \infty \wedge \tau \preccurlyeq \tau'))$.

A Logs command does not depend on the trace record before its start and satisfies an additional healthiness condition called *shift invariance*: a Logs command can be 'moved' backwards or forwards without change. Note that a

nonterminating command with an unobservable final state cannot, however, be moved backwards to become a terminating command!

Healthiness condition: $H_{\leftrightarrow}(A) \;\widehat{=}\; A = (\exists \tau_0, \tau_0' \;\cdot\; A[\tau_0, \tau_0'/\tau, \tau'] \wedge (\tau_0' - \tau_0 = \tau' - \tau) \wedge (|\tau_0'| = \infty \Rightarrow |\tau'| = \infty))\,.$

In fact $H_{\leftrightarrow}$ can be written as a generic composition and so *each* of the healthiness functions is monotonic and idempotent on predicates (see [6] for details). Thus the composed function

$$H \;\widehat{=}\; H_{\leftrightarrow} \circ H_{\infty} \circ H_{\tau} \circ H_0$$

is again monotonic and idempotent which, by the inheritance theorem in [6], ensures:

(a) the range of H forms a complete lattice (consisting of the 'healthy predicates') on which the combinators $\sqcap$, $\sqcup$, ($\fatsemi$) and $\mathtt{var}\ v$ are closed, and
(b) the laws of that complete lattice involving only $\sqcap$, $\sqcup$ and ($\fatsemi$) are inherited from the more abstract language CML (see [6]) so that the soundness of the foregoing laws of Logs is assured.

We note that conversely every healthy predicate forms the predicative semantics of a Logs command; see [6].

We refer to section 2 for the intuition behind a basic Logs command, and define its predicative semantics as follows.

The semantics of a nLogs command is simply the predicate $P(\omega, \omega', \tau, \tau')$ which describes its effect in terms of its free variables $\omega, \omega', \tau, \tau'$. Examples have appeared in sections 2 and 4. From now on we assume each predicate has those four vectors of free variables.

The semantics of $P \sqcap Q$ is the disjunction of the semantics of P and Q. This corresponds to the usual representation of the demonic nondeterminism arising from the abstraction resulting from concealment of local variables.

The semantics of $P \fatsemi Q$ for P is given, as usual, by relational composition

$$P \fatsemi Q \;\;\widehat{=}\;\; \exists \omega_0, \tau_0 \cdot P[\omega_0, \tau_0/\omega', \tau'] \wedge Q[\omega_0, \tau_0/\omega, \tau]\,.$$

In particular if P is nonterminating then $P \fatsemi Q$ equals P.

The semantics of $P \sqcup Q$ is the conjunction of the semantics of P and Q. That provides the simplest kind of parallel composition, and no real parallel computer system directly implements it. Parallel compositions in real programming languages are normally much more complicated; their implementation details are mainly due to efficiency concerns and not suitable for specification purposes, which is of primary concern here.

The semantics of $\sim P$ is the closure under H of the negation of the predicative semantics of P.

The semantics of derived Logs commands follows from their definitions, provided in section 3, in terms of the basic commands.

Compositional Modeling of Reactive Systems Using Open Nets*

P. Baldan[1], A. Corradini[1], H. Ehrig[2], and R. Heckel[3]

[1] Dipartimento di Informatica, Università di Pisa, Italy
{baldan,andrea}@di.unipi.it
[2] Computer Science Department, Technical University of Berlin, Germany
ehrig@cs.tu-berlin.de
[3] Dept. of Math. and Comp. Science, University of Paderborn, Germany
reiko@upb.de

Abstract. In order to model the behaviour of open concurrent systems by means of Petri nets, we introduce *open Petri nets*, a generalization of the ordinary model where some places, designated as *open*, represent an interface of the system towards the environment. Besides generalizing the token game to reflect this extension, we define a truly concurrent semantics for open nets by extending the Goltz-Reisig process semantics of Petri nets. We introduce a composition operation over open nets, characterized as a pushout in the corresponding category, suitable to model both interaction through open places and synchronization of transitions. The process semantics is shown to be compositional with respect to such composition operation. Technically, our result is similar to the amalgamation theorem for data-types in the framework of algebraic specifications. A possible application field of the proposed constructions and results is the modeling of interorganizational workflows, recently studied in the literature. This is illustrated by a running example.

1 Introduction

Among the various models of concurrent and distributed systems, Petri nets [16] are certainly not the most expressive or the best-behaved. However, due to their intuitive graphical representation, Petri nets are widely used both in theoretical and applied research to specify and visualize the behaviour of systems. Especially when explaining the concurrent behaviour of a net to non-experts, one important feature of Petri nets is the possibility to describe their execution within the same visual notation, i.e., in terms of processes [5].

However, when modeling *reactive systems*, i.e., concurrent systems with interacting subsystems, Petri nets force us to take a global perspective. In fact, ordinary Petri nets are not adequate to model *open* systems which can interact

* Research partially supported by the EC TMR Network GETGRATS, by the ESPRIT Working Group APPLIGRAPH, by the MURST project TOSCA and by the DFG researcher group Petri Net Technology.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 502–518, 2001.
© Springer-Verlag Berlin Heidelberg 2001

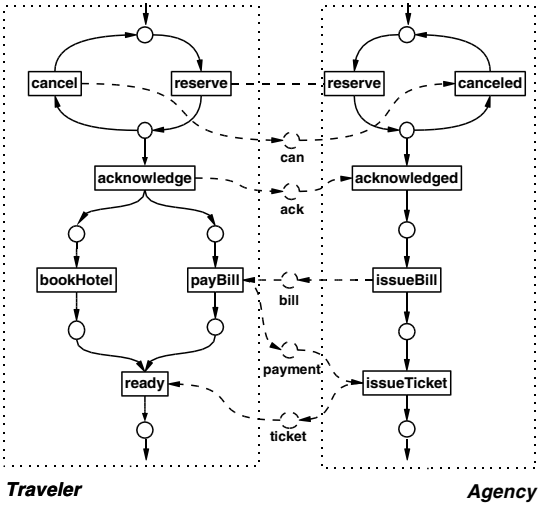


Fig. 1. Sample net modeling an interorganizational workflow.

with their environment or, in a different view, which are only partially specified. This contradicts the common practice, e.g., in software engineering, where a large system is usually built out of smaller components.

Let us explain this problem in more detail by means of a typical application of Petri nets, the specification of workflows. A *workflow* describes a business process in terms of tasks and shared resources, as needed, for example, when the integration of different organizations is an issue. A *workflow net* [17] is a Petri net satisfying some structural constraints, like the existence of one initial and one final place, and a corresponding *soundness condition*: from each marking reachable from the initial one (one token on the initial place) we can reach the final marking (one token on the final place). An *interorganizational workflow* [18] is modeled as a set of such workflow nets connected through additional places for asynchronous communication and synchronization requirements on transitions.

For instance, Fig. 1 shows an interorganizational workflow consisting of two local workflow nets Traveler and Agency related through communication places can, ack, bill, payment and ticket and a synchronization requirement between the two reserve transitions, modeled by a dashed line. The example describes the booking of a flight by a traveler in cooperation with a travel agency. After some initial negotiations (which is not modeled), both sides synchronize in the reservation of a flight. Then, the traveler may either acknowledge or cancel and re-enter the initial state. In both cases an asynchronous notification (e.g., a fax), modeled by the places ack and can, respectively, is sent to the travel agency. Next the local workflow of the traveler forks into two concurrent threads, the booking of a hotel and the payment of the bill. The trip can start when both tasks are completed and the ticket has been provided by the travel agency.

The overall net in Fig. 1 describes the system from a global perspective. Hence, the classical notion of behaviour (described, e.g., in terms of processes) is completely adequate. However, for a local subnet in isolation (like Traveler) which will only exhibit a meaningful behaviour when interacting with other subnets, this semantics is not appropriate because it does not take into account the possible interactions.

To overcome these limitations of ordinary Petri nets, we extend the basic model introducing *open nets*. An open net is a P/T Petri net with a distinguished set of places which are intended to represent the interface of the net towards the external world. Some similarities exist with other approaches to net composition, like the *Petri box calculus* [2,9,8], the *Petri nets with interface* [12, 15] and the *Petri net components* [7], which will be discussed in the conclusions. As a consequence of the (hidden, implicit) interaction between the net and the environment, some tokens can "freely" appear in or disappear from the open places. Besides generalizing the token game to reflect this changes, we provide a truly concurrent semantics by extending the ordinary *process semantics* [5] to open nets.

The embedding of an open net in a context is formally described by a morphism in a suitable category of open nets. Intuitively, in the target net new transitions can be attached to open places and, moreover, the interface towards the environment can be reduced by "closing" open places. Therefore, open net morphisms do not preserve but reflect the behaviour, i.e., any computation of the target (larger) net can be projected back to a computation in the source (smaller) net.

A *composition operation* is introduced over open nets. Two open nets Z_1 and Z_2 can be composed by specifying a common subnet Z_0 which embeds both in Z_1 and Z_2, and gluing the two nets along the common part. This is permitted only if the prescribed composition is consistent with the interfaces, i.e., only if the places of Z_1 and Z_2 which are used when connecting the two nets are actually open. The composition operation is characterized as a pushout in the category of open nets, where the conditions for the existence of the pushout nicely fit with the mentioned condition over interfaces.

Based on these concepts, the representation of the system of Fig. 1 in terms of two interacting open nets is given by the top part of Fig. 2, which comprises the two component nets Traveler and Agency, and the net Common which embeds into both components by means of open net morphisms. Places with incoming/outgoing dangling arcs are open. Observe that the common subnet Common of the components Traveler and Agency closely corresponds to the dashed items of Fig. 1, which represent the "glue" between the two components. The net resulting from the composition of Traveler and Agency over the shared subnet Common is shown in the bottom part of Fig. 2.

Obviously, one would like to have a clear relationship between the behaviours of the component nets (nets Traveler and Agency in the example) and the behaviour of the composition (net Global in the example). We show that indeed, the behaviour of the latter can be constructed "compositionally" out of the

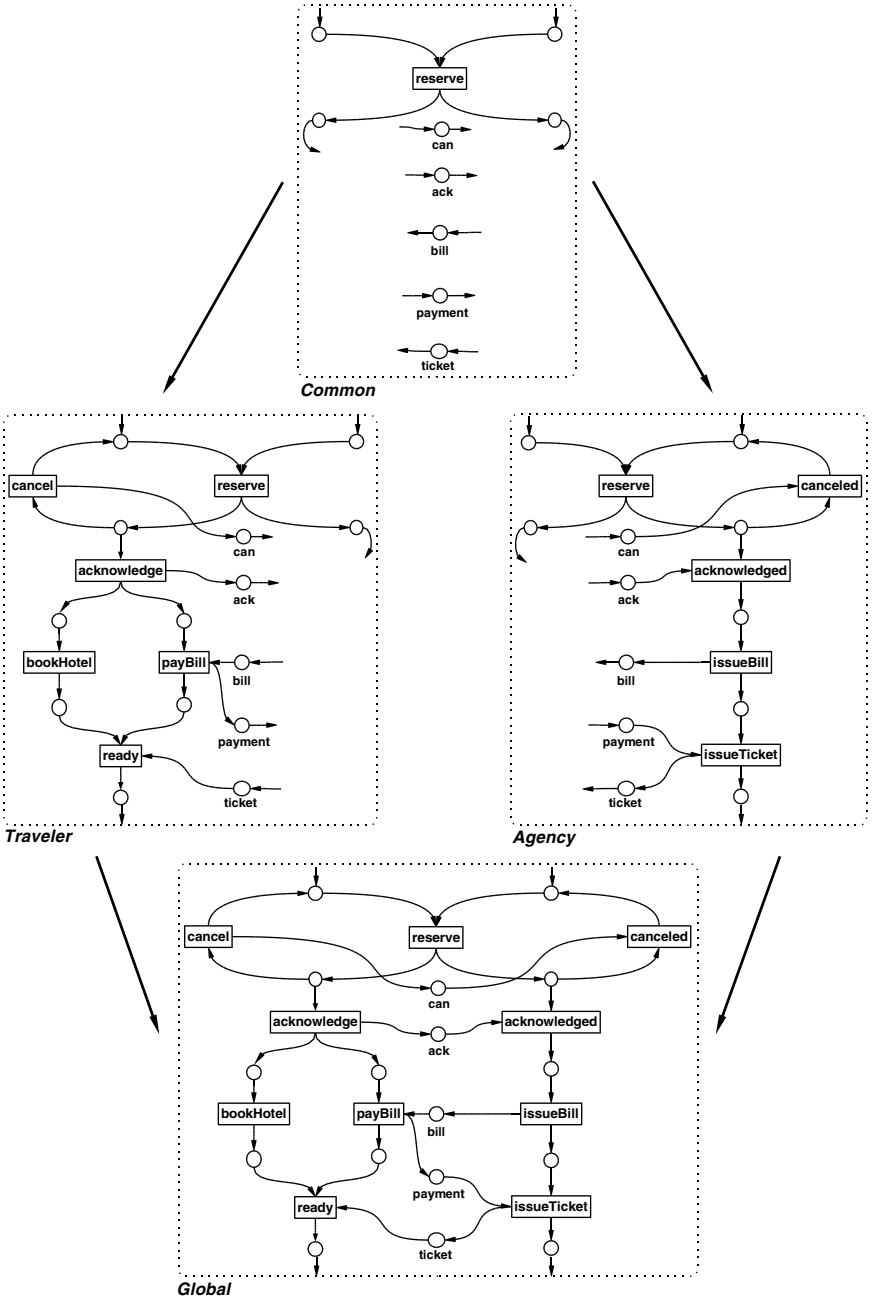


Fig. 2. Interorganizational workflow as composition of open nets Traveler and Agency.

behaviours of the former, in the sense that two deterministic processes which "agree" on the shared part, can be synchronized to produce a deterministic process over the composed net. Vice versa, *any* deterministic process of the global net can be decomposed into processes of the component nets, which, in turn, can be synchronized to give the original process again. Fig. 3 shows two processes of the nets Traveler and Agency, the corresponding common projections over net Common and the process of Global arising from their synchronization.

The synchronization of processes resembles the *amalgamation* of data-types in the framework of algebraic specifications, and therefore we will speak of *amalgamation of processes*. In analogy with the amalgamation theorem for algebraic specifications [4], the main result of this paper shows that the amalgamation and decomposition constructions mentioned above are inverse to each other, establishing a bijection between pairs of processes of two nets which agree on the common subnet and processes of the net resulting from their composition.

The rest of the paper is organized as follows. Section 2 introduces the open Petri net model and the corresponding category. Section 3 extends the notion of process from ordinary to open nets and defines the operation of behaviour projection. Section 4 introduces the composition operation for open nets. Section 5 presents the compositionality result for the process semantics of open nets. Finally, Section 6 discusses some related work in the literature and outlines possible directions of future investigation. The proofs of the results presented in this paper can be found in [1].

2 Open Nets

An *open net* is an ordinary P/T Petri net with a distinguished set of places which are intended to represent the interface of the net towards the external world (environment). As a consequence of the (hidden, implicit) interaction between the net and the environment, some tokens can freely appear in and disappear from the open places. Concretely, an open place can be either an *input* or an *output* place (or both), meaning that the environment can put or remove tokens from that place.

Given a set X we denote by $X^{\oplus}$ the free commutative monoid generated by X and by $\mathbf{2}^X$ its powerset. Moreover for a function $h : X \to Y$ we denote by $h^{\oplus} : X^{\oplus} \to Y^{\oplus}$ its monoidal extension and by the same symbol $h : \mathbf{2}^X \to \mathbf{2}^Y$ the extension of h to sets.

Definition 1 (P/T Petri net). *A* P/T Petri net *is a tuple* $N = (S, T, \sigma, \tau)$ *where* S *is the set of places,* T *is the set of transitions* ($S \cap T = \emptyset$) *and* $\sigma, \tau : T \to S^{\oplus}$ *are the functions assigning to each transition its pre- and post-set.*

In the following we will denote by ${}^{\bullet}(\cdot)$ and $(\cdot)^{\bullet}$ the monoidal extensions of the functions σ and τ to functions from $T^{\oplus}$ to $S^{\oplus}$. Furthermore, given a place $s \in S$, the pre- and post-set of s are defined by ${}^{\bullet}s = \{t \in T \mid s \in t^{\bullet}\}$ and $s^{\bullet} = \{t \in T \mid s \in {}^{\bullet}t\}$.

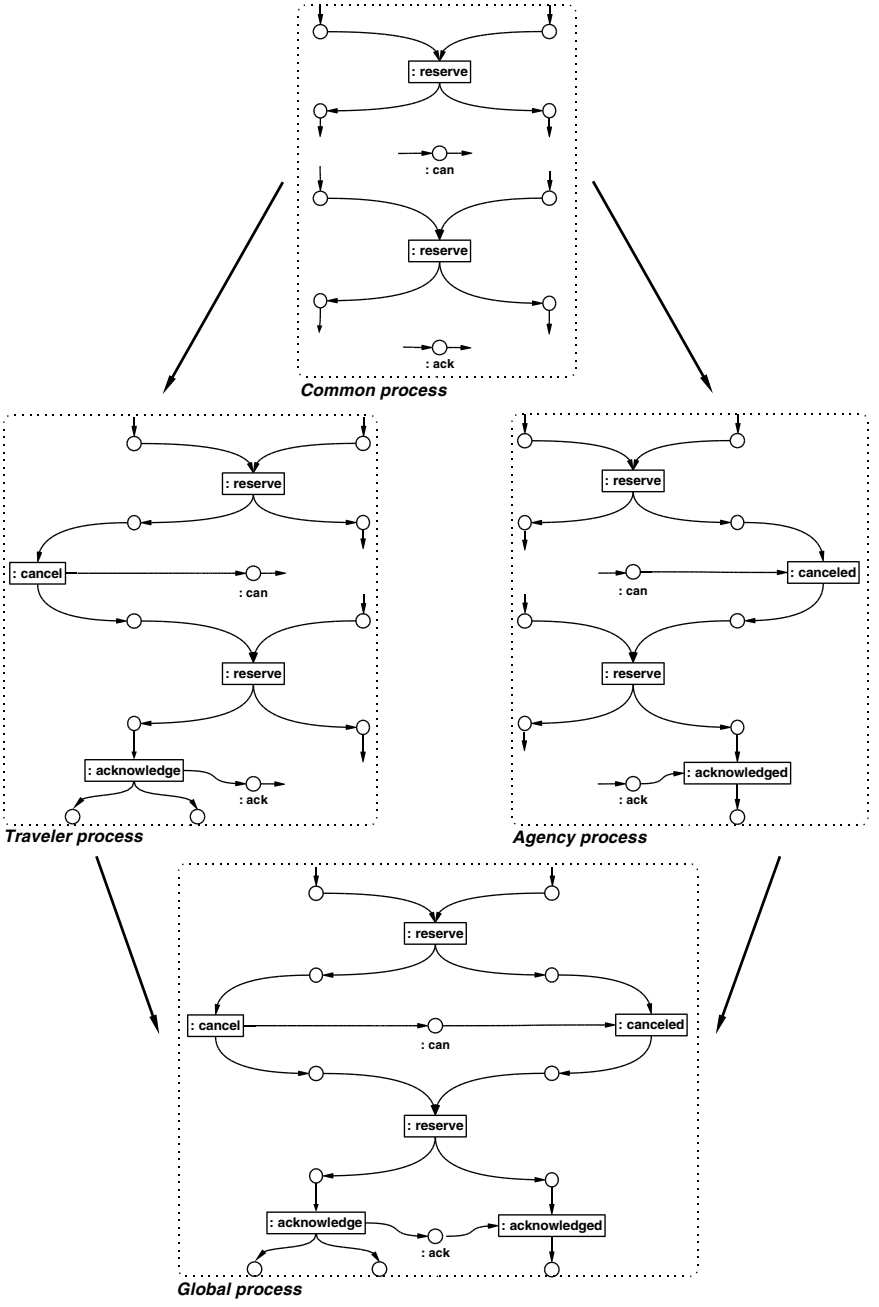


Fig. 3. Amalgamation of processes for the nets Traveler an Agency.

Definition 2 (Petri net category). *Let N_0 and N_1 be Petri nets. A* Petri net morphism *$f : N_0 \to N_1$ is a pair of total functions $f = \langle f_T, f_S \rangle$ with $f_T : T_0 \to T_1$ and $f_S : S_0 \to S_1$, such that for all $t_0 \in T_0$, ${}^\bullet f_T(t_0) = {f_S}^\oplus({}^\bullet t_0)$ and $f_T(t_0)^\bullet = {f_S}^\oplus(t_0{}^\bullet)$. The category of P/T Petri nets and Petri net morphisms is denoted by* **Net**.

Category **Net** is a subcategory of the category **Petri** of [10]. The latter has the same objects, but more general morphisms which can map a place into a multiset of places.

Definition 3 (open net). *An* open net *is a pair $Z = (N_Z, O_Z)$, where $N_Z = (S_Z, T_Z, \sigma_Z, \tau_Z)$ is an ordinary P/T Petri net and $O_Z = (O_Z^+, O_Z^-) \in \mathbf{2}^{S_Z} \times \mathbf{2}^{S_Z}$ are the input and output open places of the net.*

The notion of enabledness for transitions is the usual one, but, besides the changes produced by the firing of the transitions of the net, one considers also the interaction with the environment, modelled by a kind of invisible actions producing/consuming tokens in the input/output places of the net. The actions of the environment which produce and consume tokens in an open place s are denoted by $+_s$ and $-_s$, respectively.

Definition 4 (firing). *Let Z be an open net. A* sequential move *can be (i) the firing of a transition $m \oplus {}^\bullet t \ [t\rangle \ m \oplus t^\bullet$, with $m \in {S_Z}^\oplus$, $t \in T_Z$; (ii) the creation of a token by the environment $m \ [+_s\rangle \ m \oplus s$, with $s \in O_Z^+$, $m \in {S_Z}^\oplus$; (iii) the deletion of a token by the environment $m \oplus s \ [-_s\rangle \ m$, with $m \in {S_Z}^\oplus$, $s \in O_Z^-$. A* parallel move *is of the form*

$$m \oplus {}^\bullet A \oplus m^- \ [A\rangle \ m \oplus A^\bullet \oplus m^+,$$

with $m \in {S_Z}^\oplus$, $A \in {T_Z}^\oplus$, $m^+ \in {(O_Z^+)}^\oplus$, $m^- \in {(O_Z^-)}^\oplus$.

Example. The open nets for the local workflows Traveler and Agency of Fig. 1 are shown in the middle of Fig. 2. Ingoing and outgoing arcs without source or target designate the input and output places, respectively. The synchronization transition reserve is common to both nets and the communication places, like can, become open places.

Definition 5 (open net category). *An* open net morphism *$f : Z_1 \to Z_2$ is a Petri net morphism $f : N_{Z_1} \to N_{Z_2}$ such that, if we define $\mathsf{in}(f) = \{s \in S_1 : {}^\bullet f_S(s) - f_T({}^\bullet s) \neq \emptyset\}$ and $\mathsf{out}(f) = \{s \in S_1 : f_S(s)^\bullet - f_T(s^\bullet) \neq \emptyset\}$ then*

$$\textit{(i)}\ f_S^{-1}(O_2^+) \cup \mathsf{in}(f) \subseteq O_1^+ \qquad \textit{and} \qquad \textit{(ii)}\ f_S^{-1}(O_2^-) \cup \mathsf{out}(f) \subseteq O_1^-.$$

The morphism f is called an open net embedding *if both f_T and f_S are injective. We will denote by* **ONet** *the category of open nets and open net morphisms.*

Hereafter, to lighten the notation, we will omit the subscripts "S" and "T" in the place and transition components of morphisms, writing $f(s)$ for $f_S(s)$ and $f(t)$ for $f_T(t)$.

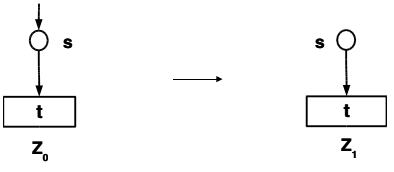
A morphism $f : Z_1 \to Z_2$ can be seen as an "insertion" of net Z_1 into a larger net Z_2, extending Z_1. In other words, Z_2 can be thought of as an instantiation of Z_1, where part of the unknown environment gets specified. Conditions (i) and (ii) first require that open places are reflected and hence that places which are "internal" in Z_1 cannot be promoted to open places in Z_2. Furthermore, the context in which Z_1 is inserted can interact with Z_1 only through the open places. To understand how this is formalized, observe that for each place s in $\mathsf{in}(f)$, its image $f(s)$ is in the post-set of a transition outside the image of ${}^\bullet s$. Hence we can think that in Z_2 new transitions are attached to s and can produce tokens in such place. This is the reason why condition (i) also asks any place in $\mathsf{in}(f)$ to be an input open place of Z_1. Condition (ii) is analogous for output places.

The above intuition better fits with open net embeddings, and indeed most of the constructions in the paper will be defined for this subclass of open net morphisms. However, for technical reasons (e.g., to characterize the composition of open nets as a pushout) the more general notion of morphism is useful.

Example. As an example of open net morphism, consider, in Fig. 2, the embedding of net Traveler into net Global. Observe that the constraints characterizing open nets morphisms have an intuitive graphical interpretation:

- the connections of transitions to their pre-set and post-set have to be preserved. New connections cannot be added;
- in the larger net, a new arc may be attached to a place only if the corresponding place of the subnet has a dangling arc in the same direction. Dangling arcs may be removed, but cannot be added in the larger net. E.g., without the outgoing dangling arc from place can in net Traveler, i.e., if place can were not output open, the mapping in from Traveler into Global would have not been a legal open net morphism.

We said that open net morphisms are designed to capture the idea of "insertion" of a net into a larger one. Hence it is natural to expect that they "reflect" the behaviour in the sense that given $f : Z_0 \to Z_1$, the behaviour of Z_1 can be projected along the morphism to the behaviour of Z_0 (this fact will be formalized later, in Construction 9). Instead, differently from most of the morphisms considered over Petri nets, open net morphisms cannot be thought of as simulations since they *do not* preserve the behaviour. For instance, consider the open nets Z_0 and Z_1 below and the obvious open net morphism between them.



Then the firing sequence $0\ [+_s\rangle\ s\ [t\rangle\ 0$ in Z_0 is not mapped to a firing sequence in Z_1.

3 Processes of Open Nets

Similarly to what happens for ordinary nets, a process of an open net, representing a concurrent computation of the net, is an open net itself, satisfying suitable acyclicity and conflict freeness requirements, together with a mapping to the original net.

The open net underlying a process is an open occurrence net, namely an open net K such that N_K is an ordinary occurrence net and satisfying some additional conditions over open places. The open places in K are intended to represent tokens which are produced/consumed by the environment in the considered computation. Consequently, every input open place is required to have an empty pre-set, i.e., to be minimal with respect to the causal order. In fact, an input open place in the post-set of some transition would correspond to a kind of generalized backward conflict: a token on this place could be generated in two different ways and this would prevent one to interpret the place as a token occurrence. Similarly, to avoid generalized forward conflicts, output open places are required to be maximal.

Definition 6 (open (deterministic) occurrence net). *An* open (deterministic) occurrence net *is an open net K such that*

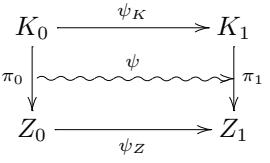
1. *N_K is an ordinary (deterministic) occurrence net, namely (i) for any $t \in T_K$, ${}^\bullet t$ and $t^\bullet$ are sets, rather than proper multisets; (ii) for any $t, t' \in T_K$, if $t \neq t'$ then ${}^\bullet t \cap {}^\bullet t' = \emptyset$ and $t^\bullet \cap t'^\bullet = \emptyset$; (iii) the causal relation $<_K$ defined as the least transitive relation such that $x <_K y$ if $y \in x^\bullet$, for $x, y \in S_K \cup T_K$, is a finitary strict partial order.*
2. *each input open place is minimal and each output open place is maximal w.r.t. $<_K$, i.e., $\forall s \in O_K^+.\ {}^\bullet s = \emptyset$ and $\forall s \in O_K^-.\ s^\bullet = \emptyset$.*

Definition 7 (open net process). *A* (deterministic) process *of an open net Z is a mapping $\pi : K \to Z$ where K is an open occurrence net and $\pi : N_K \to N_Z$ is a Petri net morphism, such that $\pi_S(O_K^+) \subseteq O_Z^+$ and $\pi_S(O_K^-) \subseteq O_Z^-$.*

Note that the process mapping π is *not*, in general, an open net morphism. In fact, the process mapping must be a simulation, i.e., it must preserve the behaviour. Moreover, the image of an open place in K must be an open place in Z, since tokens can be produced (consumed) by the environment only in input (output) open places of Z.

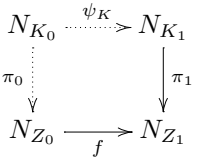
Example. A process for the open net Traveler can be found in the left part of Fig. 3. The morphism back to the original net Traveler is implicitly represented by the labeling (an item $: x$ is mapped to x). Observe that the requirements of minimality for input places and of maximality for output places of a process have a natural graphical interpretation: the absence of backward and forward conflicts extends to dangling arcs, i.e., in total, each place may have at most one ingoing and one outgoing arc.

Definition 8 (category of processes). *We denote by* **Proc** *the category where objects are processes and, given two processes* $\pi_0 : K_0 \to Z_0$ *and* $\pi_1 : K_1 \to Z_1$*, an arrow* $\psi : \pi_0 \to \pi_1$ *is a pair of open net morphisms* $\psi = \langle \psi_Z : Z_0 \to Z_1, \psi_K : K_0 \to K_1 \rangle$ *such that the following diagram (indeed the underlying diagram in* **Net***) commutes*



Let $f : Z_0 \to Z_1$ be an open net morphism. As mentioned before, it is natural to expect that each computation in Z_1 can be "projected" to Z_0, by considering only the part of the computation of the larger net which is visible in the smaller net. The above intuition is formalized, in the case of an open net embedding $f : Z_0 \to Z_1$, by showing how a process of Z_1 can be projected along f giving a process of Z_0.

Construction 9 (process projection). Let $f : Z_0 \to Z_1$ be an open net embedding and let $\pi_1 : K_1 \to Z_1$ be a process of Z_1. A *projection of* π_1 *along* f is a pair $\langle \pi_0, \psi \rangle$ where $\pi_0 : K_0 \to Z_0$ is a process of Z_0 and $\psi : \pi_0 \to \pi_1$ is an arrow in **Proc**, constructed as follows. Take the pullback of π_1 and f in **Net**, obtaining the net morphisms π_0 and ψ_K.



Then K_0 is obtained by taking N_{K_0} with the smallest sets of open places which make $\psi_K : N_{K_0} \to N_{K_1}$ an open net morphism, namely $O^+_{K_0} = {\psi_K}^{-1}(O^+_{K_1}) \cup \mathsf{in}(\psi_K)$ and $O^-_{K_0} = {\psi_K}^{-1}(O^-_{K_1}) \cup \mathsf{out}(\psi_K)$, and $\psi = \langle \psi_K, f \rangle$.

Example. The embedding of Traveler into Global in Fig. 2 induces a projection of open net processes in the opposite direction. For instance, the bottom part of Fig. 3 shows a process of Global. Its projection along the embedding of Traveler into Global is shown on the left part of the same figure. Notice how transition acknowledged, which consumes a token in place ack, is replaced in the projection by a dangling output arc: an internal action in the larger net becomes an interaction with the environment in the smaller one.

4 Composing Open Nets

We introduce a basic mechanism for composing open nets, characterized as a pushout construction in the category of open nets. Intuitively, two open nets Z_1

and Z_2 are composed by specifying a common subnet Z_0, and then by joining the two nets along Z_0. For instance, the open nets for the local workflows Traveler and Agency in the middle of Fig. 2 share the subnet Common, depicted in the top of the same figure, which represents the "glue" between the two components. The net Global resulting from the composition of Traveler and Agency over the shared subnet Common is shown in the bottom part of Fig. 2. This composition is only defined if the embeddings of the components into the resulting net satisfy the constraints of open net morphisms. For example, if we remove the ingoing dangling arc of the place ticket in the net Traveler, the embedding of Common into Traveler would still represent a legal open net morphism. However, in this case the embedding of Traveler into Global would become illegal because of the new arc from issueTicket (see condition (i) of Definition 5).

Formally, given two nets Z_1 and Z_2 and a span $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$, the composition operation constructs the corresponding pushout in **ONet**. Category **ONet** does not have all pushouts, while category **Net** does. This corresponds to the intuition that the composition operation can be performed in **Net** and then lifted to **ONet**, but only when it respects the interfaces specified by the various components, e.g., a new transition can be attached to a place only if such place is open (see also [1]).

We start by recalling that for any span $N_1 \overset{f_1}{\leftarrow} N_0 \overset{f_2}{\rightarrow} N_2$ in **Net** the pushout always exists. It can be defined as $N_1 \overset{\alpha_1}{\rightarrow} N_3 \overset{\alpha_2}{\leftarrow} N_2$, where the sets of places and transitions of N_3 are computed as the pushout in **Set** of the corresponding components, i.e., $S_3 = S_1 +_{S_0} S_2$ and $T_3 = T_1 +_{T_0} T_2$. The source and target functions are defined by: for all $t \in T_3$, if $t = \alpha_i(t_i)$ with $t_i \in T_i$ and $i \in \{1, 2\}$ then ${}^\bullet t = \alpha_i{}^{\oplus}({}^\bullet t_i)$ and $t^\bullet = \alpha_i{}^{\oplus}(t_i{}^\bullet)$. Next we formalize the condition which ensures the composability of a span in **ONet**.

Definition 10 (composable span). *Let $Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2$ be a span of open net morphisms. We say that f_1 and f_2 are* composable *if*

1. $f_2(\mathsf{in}(f_1)) \subseteq O^+_{Z_2}$ *and* $f_2(\mathsf{out}(f_1)) \subseteq O^-_{Z_2}$*;*
2. $f_1(\mathsf{in}(f_2)) \subseteq O^+_{Z_1}$ *and* $f_1(\mathsf{out}(f_2)) \subseteq O^-_{Z_1}$.

In words, f_1 and f_2 are composable if the places which are used as interfaces by f_1, namely the places $\mathsf{in}(f_1)$ and $\mathsf{out}(f_1)$, are mapped by f_2 to input and output open places in Z_2, and also the symmetric condition holds. If, and only if, this condition is satisfied the pushout of f_1 and f_2 can be computed in **Net** and then lifted to **ONet**.

Proposition 11 (pushouts in ONet). *Let $Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2$ be a span in* **ONet** *(see the diagram in Fig. 4). Compute the pushout of the corresponding diagram in the category* **Net** *obtaining the net N_{Z_3} and the morphisms α_1 and α_2, and then take as open places, for $x \in \{+, -\}$, $O^x_{Z_3} = \{s_3 \in S_3 \mid \alpha_1^{-1}(s_3) \subseteq O^x_{Z_1} \wedge \alpha_2^{-1}(s_3) \subseteq O^x_{Z_2}\}$. Then $(\alpha_1, Z_3, \alpha_2)$ is the pushout in* **ONet** *of f_1 and f_2 iff f_1 and f_2 are composable.*

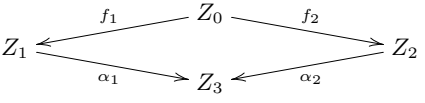


Fig. 4. Pushout in **ONet**.

5 Amalgamating Processes of Open Nets

Let $f_1 : Z_0 \to Z_1$ and $f_2 : Z_0 \to Z_2$ be a composable span of open net embeddings and consider the corresponding composition, i.e., the pushout in **ONet**, as depicted in Fig. 4. We would like to establish a clear relationship among the behaviours of the involved nets. Roughly speaking, we would like that the behaviour of Z_3 could be constructed "compositionally" out of the behaviours of Z_1 and Z_2.

In this section we show how this can be done for processes. Given two processes π_1 of Z_1 and π_2 of Z_2 which "agree" on Z_0, we construct a process π_3 of Z_3 by "amalgamating" π_1 and π_2. Vice versa, each process π_3 of Z_3 can be projected over two processes π_1 and π_2 of Z_1 and Z_2, respectively, which can be amalgamated to produce π_3 again. Hence, all and only the processes of Z_3 can be obtained by amalgamating the processes of the components Z_1 and Z_2. This is formalized by showing that, working up to isomorphism, the amalgamation and decomposition operations are inverse to each other. This leads to a bijective correspondence between the processes of Z_3 and pair of processes of the components Z_1 and Z_2 which agree on the common subnet Z_0.

As a first step towards the amalgamation of processes we identify a suitable condition which ensures that the pushout of occurrence open nets exists and produces a net in the same class. This condition will be used later to formalize the intuitive idea of processes of different nets which "agree" on a common part.

For a given span $K_1 \stackrel{f_1}{\leftarrow} K_0 \stackrel{f_2}{\rightarrow} K_2$ we introduce the notion of causality relation induced by K_1 and K_2 over K_0. When the two nets are composed the corresponding causality relations get "fused". Hence, to avoid the creation of cyclic causal dependencies in the resulting net, the induced causality will be required to be a partial order.

Definition 12 (induced causality and consistent span). *Let $K_1 \stackrel{f_1}{\leftarrow} K_0 \stackrel{f_2}{\rightarrow} K_2$ be a span in* **ONet***, where K_i ($i \in \{0,1,2\}$) are occurrence open nets. The relation of causality $<_{1,2}$ induced over K_0 by K_1 and K_2, through f_1 and f_2 is the least transitive relation such that for any x_0, y_0 in K_0, if $f_1(x_0) <_1 f_1(y_0)$ or $f_2(x_0) <_2 f_2(y_0)$ then $x_0 <_{1,2} y_0$.*

We say that the span is consistent*, written $f_1 \uparrow f_2$, if f_1 and f_2 are composable and the induced causality $<_{1,2}$ is a finitary strict partial order.*

The next proposition shows that the composition operation in **ONet**, when applied to a consistent span of occurrence nets, produces an occurrence net.

Proposition 13. *Let $K_1 \overset{f_1}{\leftarrow} K_0 \overset{f_2}{\rightarrow} K_2$ be a composable span in* **ONet**, *where K_i ($i \in \{0,1,2\}$) are occurrence open nets and let $K_1 \overset{\alpha_1}{\rightarrow} K_3 \overset{\alpha_2}{\leftarrow} K_2$ be the pushout in* **ONet**. *Then $f_1 \uparrow f_2$ if and only if the pushout object K_3 is a occurrence open net.*

Two processes π_1 of Z_1 and π_2 of Z_2 can be amalgamated only when they agree on the common subnet Z_0, an idea which is formalized by resorting to the notion of consistent span of occurrence open nets. In the rest of this section we will refer to a fixed pushout diagram in **ONet**, as represented in Fig. 4, where f_1 and f_2 are a composable span of *open net embeddings*.

Definition 14 (agreement of processes). *The processes $\pi_1 : K_1 \rightarrow Z_1$ and $\pi_2 : K_2 \rightarrow Z_2$* agree *on Z_0 if there exist projections $\langle \pi_0, \psi^i \rangle$ along f_i of π_i for $i \in \{1,2\}$ such that $\psi^1_K \uparrow \psi^2_K$ (i.e., the span $K_1 \overset{\psi^1_K}{\leftarrow} K_0 \overset{\psi^2_K}{\rightarrow} K_2$ is consistent). In this case $\langle \pi_0, \psi^1 \rangle$ and $\langle \pi_0, \psi^2 \rangle$ are called* agreement projections *for π_1 and π_2.*

Definition 15 (amalgamation of processes). *Let $\pi_i : K_i \rightarrow Z_i$ ($i \in \{0,1,2,3\}$) be processes and let $\langle \pi_0, \psi^1 \rangle$ and $\langle \pi_0, \psi^2 \rangle$ be agreement projections of π_1 and π_2 along f_1 and f_2 (see Fig. 5). We say that π_3 is an* amalgamation *of π_1 and π_2, written $\pi_3 = \pi_1 +_{\psi^1,\psi^2} \pi_2$, if there exist projections $\langle \pi_1, \phi^1 \rangle$ and $\langle \pi_2, \phi^2 \rangle$ of π_3 over Z_1 and Z_2, respectively, such that the upper square is a pushout in* **ONet**.

We next give a more constructive characterization of process amalgamation, which also proves that the result is unique up to isomorphism.

Proposition 16 (amalgamation construction). *Let $\pi_1 : K_1 \rightarrow Z_1$ and $\pi_2 : K_2 \rightarrow Z_2$ be processes that agree on Z_0, and let $\langle \pi_0, \psi^1 \rangle$ and $\langle \pi_0, \psi^2 \rangle$ be corresponding agreement projections. Then the* amalgamation *$\pi_1 +_{\psi^1,\psi^2} \pi_2$ is a process $\pi_3 : K_3 \rightarrow Z_3$, where the net K_3 is obtained as the pushout in* **ONet** *of $\psi^1_K : K_0 \rightarrow K_1$ and $\psi^2_K : K_0 \rightarrow K_2$ and the process mapping $\pi_3 : K_3 \rightarrow Z_3$ is determined by the universal property of the underlying pushout diagram in* **Net** *(see Fig. 5). Hence $\pi_1 +_{\psi^1,\psi^2} \pi_2$ is unique up to isomorphism.*

The amalgamation construction can be given a more elegant (but less constructive) characterization. In fact, process π_3 (and the process morphisms ϕ^1 and ϕ^2) can be obtained by taking the pushout in **Proc** of the arrows $\psi^1 : \pi_0 \rightarrow \pi_1$ and $\psi^2 : \pi_0 \rightarrow \pi_2$.

The next result shows how each process of a composed net can be constructed as the amalgamation of processes of the components.

Proposition 17 (decomposition of processes). *Let $\pi_3 : K_3 \rightarrow Z_3$ be a process of Z_3 and, for $i \in \{1,2\}$, let $\langle \pi_i, \phi^i \rangle$ be projections of π_3 along α_i. Then process π_3 can be recovered as a suitable amalgamation of π_1 and π_2.*

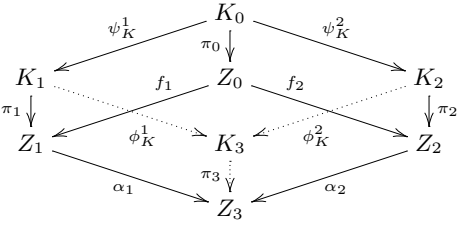


Fig. 5. Amalgamation of open net processes.

The amalgamation and decomposition results for open net processes are summarized in a theorem which establishes a bijective correspondence between the processes of Z_1 and Z_2 which agree on Z_0 and the processes of Z_3. Let Z be an open net and let $\pi : K \to Z$ be a process. We denote by $[\pi]$ the set of processes of Z isomorphic to π and by $\mathbf{DProc}(Z)$ the set of (isomorphism classes of) processes of Z. Given a span $Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2$ in **ONet**, the isomorphism classes of processes of Z_1 and Z_2 which agree on Z_0, denoted by $\mathbf{DProc}(Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2)$, is the set

$$\{[\pi_1 \overset{\psi^1}{\leftarrow} \pi_0 \overset{\psi^2}{\rightarrow} \pi_2] \mid \psi^1, \psi^2 \text{ agreement projections for } \pi_1, \pi_2 \text{ along } f_1, f_2\},$$

where isomorphism of process spans is defined in the obvious way.

Theorem 18 (amalgamation theorem). *Let Z_0, Z_1, Z_2, Z_3 be as in Fig. 4 and assume that the square is a pushout of two composable open net embeddings f_1 and f_2. Then there are composition and decomposition functions establishing a bijective correspondence between* $\mathbf{DProc}(Z_3)$ *and* $\mathbf{DProc}(Z_1 \overset{f_1}{\leftarrow} Z_0 \overset{f_2}{\rightarrow} Z_2)$.

Example. The amalgamation theorem is exemplified in Fig. 3. Two processes for the component nets Traveler and Agency which agree on the shared subnet Common, i.e., such that their projections over Common coincide, can be amalgamated to produce a process for the composed net Global. Vice versa, each process of the net Global can be reconstructed as amalgamation of compatible processes of the component nets.

6 Conclusions and Related Work

The compositionality result for the process semantics (Theorem 18) appears to be related to the amalgamation theorem for data-types in the framework of algebraic specifications [4]. There, an amalgamation construction allows one to "combine" any two algebras A_1 and A_2 of algebraic specifications $SPEC_1$ and $SPEC_2$ having a common subspecification $SPEC_0$, if and only if the restrictions of A_1 and A_2 to $SPEC_0$ coincide. The amalgamation construction produces a unique algebra A_3 of specification $SPEC_3$, union of $SPEC_1$ and $SPEC_2$. The fact

that the amalgamation of algebras is a pushout in the Grothendick's category of generalized algebras suggests the possibility of having a similar characterization for process amalgamation using fibred categories.

Open nets have been partly inspired by the notion of *open graph transformation system* [6], an extension of graph transformation for specifying reactive systems. In fact, P/T Petri nets can be seen as a special case of graph transformation systems [3] and this correspondence extends to open nets and open graph transformation systems. However, a compositionality result corresponding to Theorem 18 is still lacking in this more general setting.

In the field of Petri nets, several other approaches to net composition have been proposed in the literature. Most of them can be classified as algebraic approaches. A first family considers a category of Petri nets where morphisms arise by viewing a Petri net as the signature of a multisorted algebra, the sorts being the places. Then the semantics is expressed as a categorical adjunction, a fact which ensure its compositionality with respect to operations on nets defined in terms of universal constructions [19,10].

A second, more recent class of approaches to Petri net composition aims at defining a "calculus of nets", where a set of process algebra-like operators allows to build complex nets starting from a suitable set of basic net components. For instance, in the Petri Box calculus [2,9,8] a special class of nets, called *plain boxes* (safe and clean nets), provides the basic components. Plain boxes are then combined by means of operations which can all be seen as an instance of refinement over suitable nets. More precisely, the authors identify a special family of nets, called *operator boxes*. Once a set of operator boxes is fixed, the composition is realized by refining such operator boxes with plain boxes, an operation which produces a net still identifiable with a plain box. The calculus is given a compositional semantics (both interleaving and concurrent). Although based on some common ideas, like the use of interface places, this approach is quite different from ours, since it mainly relies on refinement and it focuses on a special class of nets and on the possibility of defining a kind of process algebra over such nets, with plain boxes as constants and operator boxes as operators.

Another relevant approach in the second family, closer to ours, is presented in the papers [12,15], which introduce a notion of Petri net with *interface*. The interface is partitioned into an input part, consisting of places, and an output part, consisting of transitions, and it is used to combine different nets, the most basic composition operation consisting of connecting the outputs of one net to the inputs of another net. Then the authors introduce a set of basic combinators which can be used to build terms corresponding to nets with an interface. The *pomset semantics* of nets with interfaces, defined by using a notion of universal context for a net, is shown to be compositional with respect to the net combinators [15]. Despite some technical differences and the different focus, which in these papers is more on the syntactical aspects of the Petri net algebra, Petri nets with interface appear to have several analogies with open nets, and their relationship surely deserves a deeper investigation.

Finally we recall the work in [7] which introduces *Petri net components*, a kind of Petri nets with distinguished input and output places. Components can be combined by means of an operation which connects the input places of a component to the output places of the other, and vice versa. A process semantics is introduced for components and it is proved to be compositional. Components can be viewed as special open nets and the composition operation on components can be defined in terms of the composition operation on open nets. A very interesting idea in [7], which we intend to explore also for open nets, is the definition of a temporal logic, interpreted over processes, which is used for reasoning in a modular way over distributed systems.

The notions of projection and of amalgamation of processes can be extended to general, possibly nondeterministic, processes. We are working on the generalization of the amalgamation theorem to nondeterministic processes, which could represent a first step towards an unfolding semantics for open nets, in the style of Winskel [11,19], still compositional with respect to our composition operation.

It would be also interesting to extend the constructions and results in this paper to open *high level nets*, which have been already studied on a conceptual level in [14]. Part of the technical background is already available — for instance it has been shown in [13] how to construct pushouts of algebraic high level nets — but a suitable formalization of high level processes is still missing.

Acknowledgement. We are grateful to Ugo Montanari for his insightful suggestions and to the anonymous referees for their helpful comments.

References

1. P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modeling of reactive systems using open nets [extended version]. The paper can be downloaded at the address `http://www.di.unipi.it/~baldan/Papers/Soft-copy-ps/open-ext.ps.gz`, 2001.
2. E. Best, R. Devillers, and J. G. Hall. The Petri box calculus: a new causal algebra with multi-label communication. In G. Rozemberg, editor, *Advances in Petri Nets*, volume 609 of *LNCS*, pages 21–69. Springer Verlag, 1992.
3. A. Corradini. Concurrent graph and term graph rewriting. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer Verlag, 1996.
4. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer Verlag, Berlin, 1985.
5. U. Golz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
6. R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*. PhD thesis, TU Berlin, 1998.
7. E. Kindler. A compositional partial order semantics for Petri net components. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 235–252. Springer Verlag, 1997.

8. M. Koutny and E. Best. Operational and denotational semantics for the box algebra. *Theoretical Computer Science*, 211(1–2):1–83, 1999.
9. M. Koutny, J. Esparza, and E. Best. Operational semantics for the Petri box calculus. In B. Jonsson and J. Parrow, editors, *Proceedings of CONCUR '94*, volume 836 of *LNCS*, pages 210–225. Springer Verlag, 1994.
10. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
11. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
12. M. Nielsen, L. Priese, and V. Sassone. Characterizing Behavioural Congruences for Petri Nets. In *Proceedings of CONCUR'95*, volume 962 of *LNCS*, pages 175–189. Springer Verlag, 1995.
13. J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5(2):217–256, 1995.
14. J. Padberg, L. Jansen, R. Heckel, and H. Ehrig. Interoperability in train control systems: Specification of scenarios using open nets. In *Proc. IDPT*, pages 17–28. Society for Design and Process Science, 1998.
15. L. Priese and H. Wimmel. A uniform approach to true-concurrency and interleaving semantics for Petri nets. *Theoretical Computer Science*, 206(1–2):219–256, 1998.
16. W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
17. W. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
18. W. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and Petri nets. *System Analysis and Modeling*, 34(3):335–367, 1999.
19. G. Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer Verlag, 1987.

Extended Temporal Logic Revisited

Orna Kupferman[1⋆], Nir Piterman[2], and Moshe Y. Vardi[3⋆⋆]

[1] Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
orna@cs.huji.ac.il, http://www.cs.huji.ac.il/~orna
[2] Weizmann Institute of Science, Department of Computer Science, Rehovot 76100, Israel
nirp@wisdom.weizmann.ac.il, http://www.wisdom.weizmann.ac.il/~nirp
[3] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu, http://www.cs.rice.edu/~vardi

Abstract. A key issue in the design of a model-checking tool is the choice of the formal language with which properties are specified. It is now recognized that a good language should extend linear temporal logic with the ability to specify all ω-regular properties. Also, designers, who are familiar with finite-state machines, prefer extensions based on automata than these based on fixed points or propositional quantification. Early extensions of linear temporal logic with automata use nondeterministic Büchi automata. Their drawback has been inability to refer to the past and the asymmetrical structure of nondeterministic automata. In this work we study an extension of linear temporal logic, called ETL_{2a}, that uses two-way alternating automata as temporal connectives. Two-way automata can traverse the input word back and forth and they are exponentially more succinct than one-way automata. Alternating automata combine existential and universal branching and they are exponentially more succinct than nondeterministic automata. The rich structure of two-way alternating automata makes ETL_{2a} a very powerful and convenient logic. We show that ETL_{2a} formulas can be translated to nondeterministic Büchi automata with an exponential blow up. It follows that the satisfiability and model-checking problems for ETL_{2a} are PSPACE-complete, as are the ones for LTL and its earlier extensions with automata. So, in spite of the succinctness of two-way and alternating automata, the advantages of ETL_{2a} are obtained without a major increase in space complexity. The recent acceptance of alternating automata by the industry and the development of symbolic procedures for handling them make us optimistic about the practicality of ETL_{2a}.

1 Introduction

In *formal verification*, we check that a system is correct with respect to a desired behavior by checking that a mathematical model of the system satisfies a

⋆ Supported in part by BSF grant 9800096.
⋆⋆ Supported in part by NSF grant CCR-9700061, NSF grant CCR-9988322, BSF grant 9800096, and by a grant from the Intel Corporation.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 519–535, 2001.
© Springer-Verlag Berlin Heidelberg 2001

formal specification of the behavior. Early formal-verification efforts considered terminating systems. There, the specification relates an initial condition about the system with a condition that is guaranteed to be satisfied upon its termination [Fra92]. In 1977, Pnueli suggested to use temporal logic in order to describe nonterminating and reactive systems [Pnu81]. Temporal logics augment propositional logics with *temporal modalities*, making it possible to describe a sequence of events in time. For example, using the temporal modalities *always* ($\Box$) and *eventually* ($\Diamond$), we can specify the behavior "if p holds in all future moments then there is a future moment in which q holds" ($\Box p \rightarrow \Diamond q$). Temporal logic has led to the development of algorithmic methods for reasoning about reactive systems [CGP99]. In particular, temporal logic *model checking* enjoys a substantial and growing use in industrial applications [BBG+94].

A key issue in the design of a model-checking tool is the choice of the formal language with which behaviors are specified. Almost two decades ago, Wolper argued that some very basic behaviors cannot be expressed by the linear temporal logic LTL. For example, he showed that the behavior "$\Box^2 p$", stating that an atomic proposition p is true in all even positions, cannot be expressed in LTL [Wol83]. Wolper suggested to extend linear temporal logic by grammar operators. It is more convenient to think about Wolper's extension in terms of ω-regular languages, as was later suggested in [VW94][1]. Intuitively, if the system is defined over a set AP of atomic propositions, then an infinite behavior of the system can be viewed as a word over the alphabet 2^{AP}, and a set of allowed behaviors can be described by an ω-regular automaton whose alphabet consists of Boolean formulas over AP. For example, the behavior $\Box^2 p$ can be described by an automaton whose language is $(\mathbf{true} \cdot p)^\omega$, and the behavior $\Box p \rightarrow \Diamond q$ can be described by an automaton whose language is $\mathbf{true}^* \cdot ((\neg p) \vee q) \cdot \mathbf{true}^\omega$.

It turned out that LTL can express precisely the star-free ω-regular behaviors [Tho81], and that its inability to express all ω-regular expressions makes LTL inadequate for numerous important tasks. For example, in *compositional model checking*, we verify a system by checking assume-guarantee specifications on its components. The specification $\langle\psi\rangle M \langle\varphi\rangle$ states that a composition that contains M and satisfies ψ, also satisfies φ. The assumption ψ can refer only to propositions observed by M, and LTL is not expressive enough to specify it [LPZ85] [2]. The recognition that the specification language should be able to specify all ω-regular properties has led to several other extensions of LTL. This includes augmenting LTL with quantification over atomic propositions, resulting in *QLTL* [LPZ85,SVW87,MP92], and augmenting LTL with fixed-point operators, resulting in the linear μ-calculus [BB87,Var88]. These suggestions, however, are not very appealing in practice: formulas of QLTL and the linear μ-calculus are very

[1] In [ET97,HT99] full ω-regularity is achieved by adding regular expressions over propositions and actions.

[2] The reason is that the assumption needs to refer to locations in the interaction between M and its environment, which cannot be done by a star-free ω-regular expression.

hard to understand, and the satisfiability problem for QLTL is non-elementary [SVW87,Mey75].

Recall that Vardi and Wolper suggested to use ω-automata as temporal connectives [VW94]. They study the usage of different types of automata. In particular, the logic ETL_r uses nondeterministic Büchi automata, and it enables the specification of all ω-regular properties. ETL_r combines two perspectives of system specification: the operational perspective (finite-state machines) and the behavioral perspective (temporal operators). This makes ETL_r, as well as related logics, appealing in practice (cf. [BBL98,AFG$^+$01]). Moreover, unlike QLTL, the satisfiability problem for ETL_r is PSPACE-complete.

The logic ETL_r still suffers from two limitations. First, it lacks temporal operators that can refer to the past. While past temporal operators do not add expressive power to LTL, they make the specification of many behaviors much more convenient[3] [LPZ85]. This convenience is reflected in the fact that the best known translation of PLTL, which extends LTL with past temporal operators, to LTL involves a non-elementary blow up [Gab87]. Also, as mentioned above, in assume-guarantee reasoning in compositional model checking, the assumptions refer only to propositions observed by the component. In PLTL we can refer to the history of the computation, which resembles using LTL with referring to locations in the interaction between the component and its environment [BK85,Pnu85,LPZ85]. To quote from Pnueli: "In order to perform compositional specification and verification, it is *convenient* to use the past operators but *necessary* to have the full power of ETL_r" [Pnu85]. The second limitation of ETL_r follows from the limited structure of nondeterministic automata. Unlike LTL, whose syntax contains both disjunctions and conjunctions, runs of nondeterministic automata are treated purely disjunctively. Modelling of conjunctions by nondeterministic automata involves a blow up of the state space and results in automata whose structure is different from the structure of equivalent LTL formulas. We would like to use automata that preserve as much as possible the structure of the formulas.

In this paper we describe and study the logic ETL_{2a}, which removes both limitations. The extension of temporal logic with past is analogous to an extension of automata with bidirectional movement. *Two-way* automata can traverse the input word back and forth (technically, the transition function of two-way automata maps a state and a letter to a set of pairs, where each pair specifies both the next state and the direction to which the reading head of the automaton proceeds). Just like PLTL is not more expressive than LTL, two-way automata are not more expressive than conventional one-way automata. Also, as in the temporal-logic paradigm, it is often more convenient to define languages using two-way automata, and the convenience is reflected in their succinctness. For example, the translation of nondeterministic two-way Büchi automata to nonde-

[3] For example, it is easy to specify the fact that grants are issued only upon requests using past temporal operators: $\Box(grant \rightarrow \ominus(\neg grant \mathcal{S} req))$, where $\ominus$ ("Yesterday") and $\mathcal{S}$ ("Since") are the past-time counterparts of "Next" and "Until"). The reader is encouraged to try and specify the behavior without past temporal operators.

terministic one-way Büchi automata involves an exponential blow-up [GH96]. So, our ETL_{2a} is going to have two-way Büchi automata as its temporal operators.

In addition, the automata are going to be *alternating*[4]. A deterministic automaton has a single run over an input word. A nondeterministic automaton may have many runs, and it accepts the word if one of them is accepting. This can be viewed as if the automaton operates in an existential mode. Dually, in a universal mode, a word is accepted if all the runs of the automaton on it are accepting. In an alternating automaton [BL80,CKS81], both existential and universal modes are allowed. The richer combinatorial structure of alternating automata makes them a convenient specification language. Formally, alternating Büchi automata are exponentially more succinct than nondeterministic Büchi automata [DH94]. In addition, the complementation of alternating Büchi automata is quadratic and simple [KV97].

Our interest in alternating automata is not merely theoretical. Alternating automata have recently been used in industrial projects. The Intel ForSpec compiler uses an intermediate language called *SPIF*, which is essentially a variant of ETL_a, using alteranting automata as temporal connectives. The ForSpec compiler translates *ForSpec Temporal Logic* (FTL) formulas [AFG+01] into SPIF, and from SPIF into nondeterministic Büchi automata [AFF+01]. We note, however, that the ability of FTL to refer to past events is very limited, because of the limitations of ETL_a. Using ETL_{2a}, it would be possible to extend FTL and SPIF to include reference to past. We also note that it has recently been shown how nondeterministic Boolean decision diagrams (BDDs) can be used for maintaining sets of states in order to reason about alternating automata [Fin01]. Thus, we believe that ETL_{2a} is interesting both theoretically and practically.

One may ask, why bother with the logic and not use two-way alternating automata directly. Indeed Boolean operators are easy to implement with alternating automata. We believe that explicit usage of Boolean connectives and nesting of formulas is more natural to users. Furthermore, the ability to name a formula and then refer to that name is much more convenient than dealing with the internals of alternating automata; indeed, this functionality is available in FTL [AFG+01].

We note that the succinctness of two-way automata holds also in the framework of alternating automata: Birget has shown that two-way alternating automata on finite words are exponentially more succinct than one-way alternating automata on finite words [Bir93], and it is not hard to extend his result to Büchi automata [Pit00]. Also, the succinctness of alternating automata is valid in the framework of two-way automata: two-way alternating Büchi automata are exponentially more succinct than two-way nondeterministic Büchi automata [GH96]. So, ETL_{2a} extends ETL_r in two important aspects. On the other hand, the two succinctness results are not additive: there is an exponential translation of two-

[4] An earlier attempt to extend ETL with alternating automata is reported in [VW94]. That attempt, however, was somewhat ad-hoc and could not handle alternating Büchi automata.

way alternating Büchi automata to one-way nondeterministic Büchi automata [Var98].

In the automata-theoretic approach to verification, we reduce questions about systems and their behavior to questions about automata [VW94]. Given a formal specification ψ, we construct a nondeterministic Büchi automaton $\mathcal{A}_\psi$ that accepts exactly the set of words that satisfy ψ. In order to check if ψ is satisfiable, we check whether the language of $\mathcal{A}_\psi$ is nonempty. In order to verify a system with respect to ψ, we check that the language of the system is contained in the language of $\mathcal{A}_\psi$. Following this approach, we would like to construct, given an ETL_{2a} formula ψ, a nondeterministic Büchi automaton that accepts exactly the set of words that satisfy ψ.

The construction proceeds in two stages. We first translate an ETL_{2a} formula ψ to a two-way alternating *hesitant* automaton. Alternating hesitant automata are an extension of alternating weak automata [MSS86], and they combine the Büchi and its dual co-Büchi acceptance condition. Recall that the complementation problem for alternating Büchi automata is quadratic. On the other hand, complementing an alternating Büchi automaton to a co-Büchi alternating automaton can be done by dualizing the transition function and the acceptance condition. Consequently, the combination of both conditions leads to a linear translation of ETL_{2a} to two-way alternating hesitant automata. In the second stage we translate the two-way alternating hesitant automaton to a one-way nondeterministic Büchi automaton. For that, we first remove the hesitation and get a Büchi automaton, and then combine techniques for removing alternation [MS95] with techniques for removing bidirectionality [Var88]. The fact we deal with hesitant word automata makes the procedure much simpler than the one required for the translation of two-way alternating parity tree automata to one-way nondeterministic parity tree automata [Var98]. All in all, given an ETL_{2a} formula ψ, the nondeterministic Büchi automaton $\mathcal{A}_\psi$ has $2^{O(|\psi|^2)}$ states. It follows that the model-checking and the satisfiability problems for ETL_{2a} can be solved in polynomial space. Matching lower bounds are easy to show, hence the problems are PSPACE-complete, as are the ones for ETL_r or LTL [SC85]. It follows that the in spite of the succinctness of two-way and alternating automata, the advantages of ETL_{2a} are obtained without a major increase in space complexity.

2 Definitions

For a finite alphabet Σ, a *word* $w \in \Sigma^\omega$ is an infinite sequence of letters from Σ. We denote by w_i the *i-th* letter of w.

Nondeterministic automata. A *nondeterministic automaton* is $A = \langle \Sigma, Q, Q_0, \rho, F \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\rho : Q \times \Sigma \to 2^Q$ is a transition function, and $F \subseteq Q$ is an acceptance condition. A *run* of A on a word $w \in \Sigma^\omega$ is an infinite sequence $r = q_0, q_1, \ldots$, where $q_0 \in Q_0$ and for all $i \geq 0$, we have $q_{i+1} \in \rho(q_i, w_i)$. Let $inf(r)$ denote the set of all states occurring infinitely often in r. Formally,

$inf(r) = \{s \mid s = q_i \text{ for infinitely many } i\text{'s}\}$. We consider two types of acceptance conditions *Büchi* and *co-Büchi*. A run of a Büchi automaton is *accepting* if it visits states from F infinitely often; i.e., $inf(r) \cap F \neq \emptyset$. A run of a co-Büchi automaton is *accepting* if it visits states from F only finitely often; i.e., $inf(r) \cap F = \emptyset$.

Hesitant automata combine the Büchi and the co-Büchi acceptance conditions. They extend *weak automata* [MSS86] by a richer acceptance condition. A hesitant automaton $A = \langle \Sigma, B, C, Q_0, \rho, F \rangle$ is a nondeterministic automaton such that the set of states $Q = B \cup C$ is the disjoint union of a set B of *Büchi states* and a set C of *co-Büchi states*. In addition, there is a partition of Q into disjoint sets, such that for each set S in the partition, either $S \subseteq B$, in which case S is a *Büchi set*, or $S \subseteq C$, in which case S is a *co-Büchi set*. For a state $q \in Q$, let $[q]$ denote the set of states in q's set in this partition. There exists a partial order $\leq$ on the collection of the sets such that for every two states q and q' for which q' occurs in $\delta(q, \sigma)$, for some $\sigma \in \Sigma$, we have $[q'] \leq [q]$. Thus, transitions from a state in a set S lead to states in either the same set or a lower one. It follows that a run r of a hesitant automaton ultimately gets trapped within some set S in the partition. The run r is *accepting* iff either $S \subseteq B$ is a Büchi set and $inf(r) \cap F \neq \emptyset$ or $S \subseteq C$ is a co-Büchi set and $inf(r) \cap F = \emptyset$. Thus, a run of a nondeterminisitic hesitant automaton may switch between Büchi and co-Büchi sets, yet eventually it stays forever in some set, and acceptance is determined according to the classification of this set. Note that if $C = \emptyset$, then A is a Büchi automaton, and that if $B = \emptyset$, then A is a co-Büchi automaton.

An automaton A *accepts* a word w if there exists an accepting run of A on w. Otherwise, A *rejects* w. The *language* of A, denoted $L(A)$, is the set of all words accepted by A. The *complementary language* of A is the set $\Sigma^\omega \setminus L(A)$ of all words w rejected by A.

Alternating automata. For a set Q, we denote by $B^+(Q)$ the set of all positive Boolean formulas over Q, where we also allow **true** and **false**. We say that a set $Q' \subseteq Q$ *satisfies* a formula $\theta \in B^+(Q)$ (denoted $Q' \models \theta$) if by assigning true to all members of Q' and false to all members of $Q \setminus Q'$, the formula θ evaluates to true. Note that the formula **true** is satisfied by the empty set and the formula **false** cannot be satisfied. Given a formula $\theta \in B^+(Q)$, the *dual* of θ, denoted by $\widetilde{\theta}$, is obtained from θ by switching $\wedge$ and $\vee$, and switching **true** and **false**.

A *tree* is a set $T \subseteq \mathbb{N}^*$ such that if $x \cdot c \in T$, where $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$, then also $x \in T$. The elements of T are called *nodes*, and the empty word ϵ is the *root* of T. For every $x \in T$, the nodes $x \cdot c$ where $c \in \mathbb{N}$ are the *children* of x, the nodes $x \cdot y$ where $y \in \mathbb{N}^*$ are the *successors* of x. A node is a *leaf* if it has no children. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\epsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $c \in \mathbb{N}$ such that $x \cdot c \in \pi$. Given an alphabet Σ, a *Σ-labeled tree* is a pair $\langle T, r \rangle$, where T is a tree and $r : T \to \Sigma$ maps each node of T to a letter in Σ.

An *alternating automaton* is $A = \langle \Sigma, Q, q_0, \rho, F \rangle$, where Σ, Q, and F are as in nondeterministic automata, q_0 is a unique initial state, and $\rho : Q \times \Sigma \to B^+(Q)$ is the transition function. We can say that a nondeterministic automaton accepts

a word $w = w_0 \cdot w_1 \cdot w_2 \cdots$ from state s if it accepts the suffix $w_1 \cdot w_2 \cdots$ from one of the states in $\rho(s, w_0)$. In alternating automata, we allow posing both existential and universal demands on the suffix of the word. For example, if $\rho(s, a) = s_1 \wedge s_2 \vee s_3$, then A accepts a word starting with a from state s if it accepts the suffix of the word from both s_1 and s_2, or it accepts the suffix from s_3. For that, A sends to the suffix either two copies of itself, in states s_1 and s_2, or a single copy, in state s_3.

A *run* of an alternating automaton on a word $w \in \Sigma^\omega$ is a Q-labeled tree $\langle T, r\rangle$, where $r(\epsilon) = q_0$ and for all $x \in T$ the (possibly empty) set $\{r(x \cdot c) \mid c \in \mathbb{N} \text{ and } x \cdot c \in T\}$ satisfies the formula $\rho(r(x), w_{|x|})$. For a path π in the tree T, let $inf(r|\pi)$ denote the set of all states occurring infinitely often along that path in r, formally $inf(r|\pi) = \{s \mid s = r(x) \text{ for infinitely many } x \text{ in } \pi\}$. We consider alternating Büchi and co-Büchi automata. A run of an alternating Büchi automaton is *accepting* if for all infinite paths π in T, we have $inf(r|\pi) \cap F \neq \emptyset$. A run of an alternating co-Büchi automaton is *accepting* if for all infinite paths π in T we have $inf(r|\pi) \cap F = \emptyset$.

Two-way alternating automata. A *2-way alternating automaton* is $A = \langle \Sigma, Q, q_0, \rho, F\rangle$, where Σ, Q, q_0, and F are as in alternating automata, and the transition function is $\rho : Q \times \Sigma \to B^+(\{-1, 0, 1\} \times Q)$. Alternating automata allowed us to pose both existential and universal demands on the suffix of the word. Two-way automata allow us to pose demands also on the prefix of the word. Technically, when the reading head of A is on the i-th position of w, it can move to locations $i-1$, i, and $i+1$. For example, $\rho(s_0, a) = (-1, s_1) \wedge (1, s_2) \vee (0, s_3)$ means that when the automaton is in state s_0 reading the letter a in location i, it can either send a copy in state s_1 to location $i-1$ and a copy in state s_2 to location $i+1$, or stay in location i in the state s_3. If $i = 0$, the automaton must choose the second option.

A *run* of A on a word $w \in \Sigma^\omega$ is a $(Q \times \mathbb{N})$-labeled tree $\langle T, r\rangle$, where $r(\epsilon) = (q_0, 0)$ and for all $x \in T$ with $r(x) = (r, k)$, the set $\{(q, \Delta) \mid c \in \mathbb{N},\ x \cdot c \in T, \text{ and } r(x \cdot c) = (q, k + \Delta\}$ satisfies the formula $\rho(r, w_k)$. For a path π, the set $inf(r|\pi)$ is defined as in alternating automata, thus $inf(r|\pi) = \{s \mid \text{there are infinitely many nodes } x \in \pi \text{ with } r(x) \in \{s\} \times \mathbb{N}\}$. A run of a 2-way alternating Büchi automaton is *accepting* if all infinite paths π in T have $inf(r|\pi) \cap F \neq \emptyset$ and a run of a *2-way alternating co-Büchi automaton* is accepting if all infinite paths π in T have $inf(r|\pi) \cap F = \emptyset$.

A 2-way alternating hesitant automaton $A = \langle \Sigma, B, C, q_0, \rho, F\rangle$ obeys the same restrictions as a nondeterministic hesitant automaton. Namely, the state set $Q = B \cup C$ is the union of Büchi and co-Büchi sets, there is a partition of the state set and a partial order that restricts the transition function. It follows that every infinite path in a run tree of a 2-way alternating hesitant automaton ultimately gets trapped within some $S \times \mathbb{N}$, for a set S in the partition. The run $\langle T, r\rangle$ is accepting if for every infinite path π in T, either $S \subseteq B$ and $inf(r|\pi) \cap F \neq \emptyset$, or $S \subseteq C$ and $inf(r|\pi) \cap F = \emptyset$.

Note that a 1-way alternating automaton can be viewed as a 2-way alternating automaton whose transition function is restricted to formulas from

$B^+(\{1\} \times Q)$. Also, a nondeterministic automaton can be viewed as an alternating automaton whose transitions are restricted to disjunctions over the set Q. Given an automaton $A = \langle \Sigma, Q, q_0, \rho, F \rangle$ and a state $q \in Q$, we denote by A^q the automaton with initial state q; i.e, is $A^q = \langle \Sigma, Q, q, \rho, F \rangle$.

Given a 2-way alternating Büchi automaton $A = \langle \Sigma, Q, q_0, \rho, F \rangle$, the dual of A is the co-Büchi automaton $\widetilde{A} = \langle \Sigma, Q, q_0, \widetilde{\rho}, F \rangle$, where $\widetilde{\rho}(s, a)$ is the dual of $\rho(s, a)$. The automata A and $\widetilde{A}$ accept complementary languages [MS87]; i.e. $L(\widetilde{A}) = \Sigma^\omega \setminus L(A)$. Given an alternating hesitant automaton $A = \langle \Sigma, B, C, q_0, \rho, F \rangle$, the dual of A is the alternating hesitant automaton $\widetilde{A} = \langle \Sigma, C, B, q_0, \widetilde{\rho}, F \rangle$, where the set of Büchi states and the set of co-Büchi states switch roles. Again, $\widetilde{A}$ accepts the complementary language of A. Clearly, $\widetilde{\widetilde{A}}$ is A again.

We denote the different types of automata by four-symbol acronyms in $\{1, 2\} \times \{D, N, A\} \times \{B, C, H\} \times \{W\}$, where the first symbol describes whether the automaton is 2-way or 1-way, (for 1-way automata we often omit the 1), the second symbol describes the branching mode of the automaton (deterministic, nondeterministic, or alternating), the third symbol describes the type of acceptance condition (Büchi, co-Büchi or hesitant), and the last symbol indicates that the automaton runs over words. For example, 1DBW denotes 1-way deterministic Büchi automata, as well as the set of ω-regular languages that can be recognized by a deterministic Büchi word automaton.

Linear Temporal Logic. The linear temporal logic LTL extends propositional logic by temporal operators like always ($\Box$), eventually ($\Diamond$), until (U), and next-time ($\bigcirc$) [Pnu81]. The semantics of LTL is defined with respect to infinite words in $(2^{AP})^\omega$, for a set AP of atomic propositions. For example, the formula $\Box p$ (always p) is satisfied by words all of whose letters contain the atom p. For full syntax and semantics see [Pnu81].

Extended Temporal Logic. As mentioned above, Vardi and Wolper suggested to increase the expressive power of LTL by using 1NBW as temporal connectives [VW94]. Suppose the alphabet of an 1NBW A is the set 2^{AP}. The 1NBW A defines a set of sequences of truth assignments to the propositions. We can view A as a formula that is satisfied by exactly all the words accepted by A. The formal definition is a bit more complex, as automata are allowed to use other formulas as part of their alphabet and not only propositions. Below we describe the definition of ETL_r as defined in [VW94].

We start with the syntax. Formulas are defined with respect to a set AP of atomic propositions as follows.

- Every proposition $p \in AP$ is a formula.
- If φ_1 and φ_2 are formulas, then $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, and $\varphi_1 \wedge \varphi_2$ are formulas.
- For every 1NBW $A = \langle \Sigma, Q, \rho, Q_0, F \rangle$ with $\Sigma = \{a_1, \ldots, a_n\}$, if $\varphi_1, \ldots, \varphi_n$ are formulas, then $A(\varphi_1, \ldots, \varphi_n)$ is a formula.

The *semantics* of ETL_r is defined with respect to pairs $(\pi, i) \in (2^{AP})^\omega \times \mathbb{N}$, of words and locations. Consider an 1NBW $A = \langle \Sigma, Q, Q_0, \rho, F \rangle$. A *run* of a formula $A(\varphi_1, \ldots, \varphi_n)$ over a word π starting at point i, is an infinite sequence

$\sigma = q_0, q_1, \ldots$ of states from Q, such that $q_0 \in Q_0$ and for all $k \geq 0$, there is some $a_j \in \Sigma$ such that $(\pi, i+k) \models \varphi_j$ and $q_{k+1} \in \rho(q_k, a_j)$. The run is *accepting* if $inf(r) \cap F \neq \emptyset$.

We use $(\pi, i) \models \psi$ to indicate that the word π in the location i satisfies the formula ψ. For a word $\pi \in (2^{AP})^\omega$ and a location $i \in \mathbb{N}$, the relation $\models$ is defined as follows.

- For a proposition $p \in AP$, we have $(\pi, i) \models p$ iff $p \in \pi_i$.
- $(\pi, i) \models \neg\varphi_1$ iff not $(\pi, i) \models \varphi_1$.
- $(\pi, i) \models \varphi_1 \vee \varphi_2$ iff $(\pi, i) \models \varphi_1$ or $(\pi, i) \models \varphi_2$.
- $(\pi, i) \models \varphi_1 \wedge \varphi_2$ iff $(\pi, i) \models \varphi_1$ and $(\pi, i) \models \varphi_2$.
- $(\pi, i) \models A(\varphi_1, \ldots, \varphi_n)$ iff there is an accepting run of $A(\varphi_1, \ldots, \varphi_n)$ over π starting at i.

Consider for example the 1NBW $A = \langle \Sigma, Q, Q_0, \rho, F \rangle$, where $\Sigma = \{a, b\}$, $Q = \{q_0, q_1\}$, $\rho(q_0, a) = \rho(q_1, a) = \{q_0\}$, $\rho(q_0, b) = \rho(q_1, b) = \{q_1\}$, and $Q_0 = F = \{q_1\}$. The state q_1 is visited exactly when A reads the letter b. A run of A is accepting if it visits state q_1 infinitely often. Hence, the ETL_r connective $A(\neg p, p)$, where p is a proposition, is true iff p is true infinitely often. It is equal to the LTL formula $\Box\Diamond p$. As another example, consider the formula $\psi = \Box(grant \rightarrow \ominus (\neg grant \mathcal{S} req))$ stating that grants are issued only upon requests. We describe an equivalent ETL_r formula for it. Consider the 1NBW $A = \langle \Sigma, Q, Q_0, \rho, F \rangle$, where $\Sigma = \{a, b, c, d\}$, $Q = \{q_0, q_1\}$, $\rho(q_0, a) = \{q_0\}$, $\rho(q_0, b) = \{q_1\}$, $\rho(q_1, c) = \{q_1\}$, $\rho(q_1, d) = \{q_0\}$, $Q_0 = \{q_0\}$, and $F = \{q_0, q_1\}$. Note that all the infinite runs of A are accepting. The state q_0 corresponds to a configuration in which no requests are pending. The state q_1 corresponds to a configuration in which there is at least one request pending. Accordingly, the ETL_r formula $A(\neg req \wedge \neg grant, req \wedge \neg grant, grant \rightarrow req, grant \wedge \neg req)$ is equivalent to ψ.

Extending temporal logic with 2-way alternating automata. We now define formally the logic ETL_{2a}. The logic ETL_{2a} extends ETL_r by having 2-way alternating automata as its temporal connectives. Complementing the transition function of alternating automata is very simple. Hence, by allowing both Büchi and co-Büchi acceptance conditions, we can make the complementation of the temporal connectives simple. Accordingly, ETL_{2a}, uses both 2ABW and 2ACW as automata connectives. Runs of formulas that are automata connectives are defined as follows.

Consider a 2-way alternating automaton $A = \langle \Sigma, Q, q_0, \rho, F \rangle$. A *run* of the formula $A(\varphi_1, \ldots, \varphi_n)$ over a word w starting at point i, is a finite or infinite $(Q \times \mathbb{N})$-labeled tree $\langle T, r \rangle$ such that $r(\epsilon) = (q_0, i)$ and for all $x \in T$ with $r(x) = (q, k)$, there is some $a_j \in \Sigma$ such that $(\pi, k) \models \varphi_j$ and the (possibly empty) set $P = \{(q', \Delta) \mid$ There is a child y of x in T such that $r(y) = (q', k + \Delta)\}$ satisfies the transition $\rho(r(x), a_j)$. Intuitively, the children of x are labeled by the states of A and the locations in w from which the copies of the automaton should start running. Note that as $\varphi_1, \ldots, \varphi_n$ are not mutually exclusive, different copies may choose different formulas. If the automaton is a 2ABW, the run is *accepting* if for all infinite paths π of T we have $inf(r|\pi) \cap F \neq \emptyset$. If the automaton is a 2ACW, the run is *accepting* if for all paths π of T we have $inf(r|\pi) \cap F = \emptyset$. When not

important or clear from the context, we often write the formula $A(\varphi_1, \ldots, \varphi_n)$ as A.

Recall the formula ψ stating that grants are issued only upon requests. We now describe an ETL_{2a} formula for it. Consider the 2ABW $A = \langle \Sigma, Q, q_0, \rho, F \rangle$, where $\Sigma = \{a, b, c, d\}$, $Q = \{q_0, q_1\}$, $\rho(q_0, a) = (q_0, 1)$, $\rho(q_0, b) = (q_0, 1) \wedge (q_1, -1)$, $\rho(q_1, b) = \mathbf{false}$, $\rho(q_1, c) = \mathbf{true}$, $\rho(q_1, d) = (q_1, -1)$, and $F = \{q_0\}$. The formula $A(\neg grant, grant, req, \neg req \wedge \neg grant)$ is equivalent to ψ. To see that, note that whenever A visits the state q_0 and reads a letter containing a grant, it sends a copy that goes backwards, expecting a request before it comes across a grant. Also, as $q_1 \notin F$, a request has to be eventually found. The ETL_{2a} formula has very much the same structure as the PLTL formula.

Similar to other logics, handling ETL_{2a} is easier in positive normal form, where negations are pushed inward using De-Morgan laws. In an ETL_{2a} formula in positive normal form, negations apply to automata and atomic propositions only. For a formula ψ, let $\widetilde{\psi}$ denote $\neg\psi$ in positive normal form[5].

Given an ETL_{2a} formula ψ in a positive normal form, the *closure* of ψ, denoted $cl(\psi)$ includes all the subformulas of ψ and their complements. This includes formulas of the form A^q, for an automata connective A and a state q of it. For simplicity, we assume that the state sets of the automata connectives in ψ are pairwise disjoint, thus we can denote the subformula $A(\psi_1, \ldots, \psi_n)$ by $q_0(\psi_1, \ldots, \psi_n)$, for the initial state q_0 of A. Similarly, we denote $A^q(\psi_1, \ldots, \psi_n)$ by $q(\psi_1, \ldots, \psi_n)$. When $\psi_1, \ldots, \psi_n$ are clear from the context, we write just q_0 or q, respectively. Formally, the set $cl(\psi)$ is the minimal set satisfying all the following.

- $\psi \in cl(\psi)$,
- If $\psi_1 \in cl(\psi)$, then $\widetilde{\psi_1} \in cl(\psi)$.
- if $\psi_1 \wedge \psi_2 \in cl(\psi)$ then $\{\psi_1, \psi_2\} \subseteq cl(\psi)$,
- if $\psi_1 \vee \psi_2 \in cl(\psi)$ then $\{\psi_1, \psi_2\} \subseteq cl(\psi)$, and
- if $q_0(\psi_1, \ldots, \psi_n) \in cl(\psi)$, for a 2ABW or a 2ACW $A = \langle \Sigma, Q, q_0, \rho, F \rangle$, then $\{\psi_1, \ldots, \psi_n\} \subseteq cl(\psi)$, and for all $q \in Q$, we have $q(\psi_1, \ldots, \psi_n) \in cl(\psi)$.

Note that the formulas in $cl(\psi)$ are in positive normal form. Thus, negation applies to atomic propositions and automata connectives only. Consider again the formula $\varphi = A(\neg grant, grant, req, \neg req \wedge \neg grant)$ discussed above. The closure of φ is $cl(\varphi) = \{q_0, q_1, \neg q_0, \neg q_1, grant, \neg grant, req, \neg req, \neg req \wedge \neg grant, req \vee grant\}$.

For a formula ψ, the *models* of the formula is the set $L(\psi)$ of all infinite words $w \in (2^{AP})^\omega$ that satisfy the formula.

[5] Consider an automaton A. Note that the formula $\psi = \neg A(\varphi_1, \ldots, \varphi_n)$ is not equivalent to the formula $\tilde{A}(\varphi_1, \ldots, \varphi_n)$, where $\tilde{A}$ is the dual of A. This is because both A and $\tilde{A}$ treat the formulas $\varphi_1, \ldots, \varphi_n$ existentially. Indeed, for both automata, the transition from a state to its successor involves a choice of some φ_i. In order for ψ to be false, all the runs of A should be rejected, thus $\tilde{A}$ should treat the formulas $\varphi_1, \ldots, \varphi_n$ universally. Universally in this case means that if φ_i holds then $\tilde{A}$ should take the transition corresponding to the letter a_i. This is why the positive normal form for ETL_{2a} allows the application of negation to automata connectives.

3 Decision Procedures for ETL_{2a}

In this section we solve the satisfiability and model-checking problems for ETL_{2a}. Given an ETL_{2a} formula ψ, we construct a 1NBW $\mathcal{A}_\psi$ such that $\mathcal{A}_\psi$ accepts exactly all the words satisfying ψ. The size of $\mathcal{A}_\psi$ is $2^{O(|\psi|^2)}$. Using $\mathcal{A}_\psi$, we show that both the satisfiability and the model-checking problems for ETL_{2a} are PSPACE-complete. The construction of $\mathcal{A}_\psi$ proceeds in two stages, with 2AHW serving as an intermediate formalism.

We describe now how to construct the intermediate 2AHW.

Theorem 1. *Given an ETL_{2a} formula ψ of length n, there is a 2AHW $\mathcal{H}_\psi$ such that $L(\mathcal{H}_\psi) = L(\psi)$ and $\mathcal{H}_\psi$ has $O(n)$ states.*

Proof: Given a set Q of states, let $\neg Q = \{\neg q \mid q \in Q\}$. We define the function $dual : B^+(\{-1,0,1\} \times Q) \to B^+(\{-1,0,1\} \times \neg Q)$ as follows. For a formula $\theta \in B^+(\{-1,0,1\} \times Q)$, the formula $dual(\theta)$ is obtained from $\tilde{\theta}$ (the dual of θ) by replacing every atom of the form $(\Delta, q) \in \{-1,0,1\} \times Q$ by the atom $(\Delta, \neg q)$. So, $dual(\theta)$ switches $\vee$ and $\wedge$, and **true** and **false**, and also adds negationa in front of states in Q. For example, $dual(((-1,s) \wedge (0,t)) \vee (1,q)) = ((-1,\neg s) \vee (0,\neg t)) \wedge (1,\neg q)$.

Now, given an ETL_{2a} formula ψ, we define $\mathcal{H}_\psi = \langle 2^{AP}, B, C, \psi, \eta, \alpha \rangle$, where $B \cup C = cl(\psi)$, and

- The set of Büchi states is

$$B = \begin{array}{l} \{q \mid A \text{ is a 2ABW connective in } \psi \text{ and } q \text{ is a state of } A\} \cup \\ \{\neg q \mid A \text{ is a 2ACW connective in } \psi \text{ and } q \text{ is a state of } A\}. \end{array}$$

  The set of co-Büchi states is $C = cl(\psi) \setminus B$. We note that the decision to include elements of $cl(\psi)$ that are not states of automata in C is arbitrary. Indeed, the transition from such states is to strict subformulas, and the automaton is not going to get trapped in a set associated with them.
  The partition of $cl(\psi)$ is as follows. For every state $s \in cl(\psi)$ such that s is not a state of an automaton, the Singleton $\{s\}$ is a set of the partition. For an automaton $A = \langle \Sigma, Q, q_0, \rho, F \rangle \in cl(\psi)$, all the states $\{q \mid q \in Q\}$ form a set in the partition, and all the states $\{\neg q \mid q \in Q\}$ form a set in the partition. The partial order $\leq$ on the sets is such that $[s'] \leq [s]$ iff $s' \in cl(s)$.
- The transition function $\eta : cl(\psi) \times 2^{AP} \to B^+(\{-1,0,1\} \times cl(\psi))$ is defined for every formula in $cl(\psi)$ and letter $a \in 2^{AP}$ as follows.
  - For a proposition $p \in AP$, we have $\eta(p,a) = \mathbf{true}$ and $\eta(\neg p, a) = \mathbf{false}$ if $p \in a$, and $\eta(p,a) = \mathbf{false}$ and $\eta(\neg p, a) = \mathbf{true}$ if $p \notin a$.
  - $\eta(\psi_1 \wedge \psi_2, a) = (\psi_1, 0) \wedge (\psi_2, 0)$
  - $\eta(\psi_1 \vee \psi_2, a) = (\psi_1, 0) \vee (\psi_2, 0)$
  - Let $A(\psi_1, \ldots, \psi_n) \in cl(\psi)$ such that $A = \langle \{a_1, \ldots, a_n\}, Q, q_0, \rho, F \rangle$. For every $q \in Q$ we have $\eta(q(\psi_1, \ldots, \psi_n), a) = \bigvee_{i=1}^{n} [(\psi_i, 0) \wedge \rho(q, a_i)]$
  - Let $A(\psi_1, \ldots, \psi_n) \in cl(\psi)$ such that $A = \langle \{a_1, \ldots, a_n\}, Q, q_0, \rho, F \rangle$. For every $q \in Q$ we have $\eta(\neg q(\psi_1, \ldots, \psi_n), a) = \bigwedge_{i=1}^{n} [(\widetilde{\psi_i}, 0) \vee dual(\rho(q, a_i))]$

The transition from states associated with a 2ABW or a 2ACW $A(\psi_1, \ldots, \psi_n)$ makes sure that there is indeed an accepting run of the formula. For that, when the automaton is in state q of A, it checks that there is a formula ψ_i in $\psi_1, \ldots, \psi_n$ such that ψ_i holds in the current location (checked by sending the copy $(\psi_i, 0)$), and that the formula A^q has an accepting run starting with the transition taken by reading a_i (checked by the copies sent by $\rho(q, a_i)$). The transition from states associated with a formula $\neg A(\psi_1, \ldots, \psi_n)$ are dual.

It is easy to see that η respects the partial order on the partition of $B \cup C$.

– The acceptance condition is

$$\alpha = \{q, \neg q \mid q \in F, \text{ for an automaton connective } A = \langle \Sigma, Q, q_0, \rho, F \rangle \text{ in } \psi\}.$$

For a 2ABW A and state $q \in F$, we would like to visit q infinitely often, and indeed q is a Büchi state in α. On the other hand, the transition from the state $\neg q$ is obtained by dualizing the transitions from q, we would like to visit it finitely often, and indeed it is a co-Büchi state in α. So, both q and $\neg q$ are members of α and the restriction as to whether they should be visited finitely or infinitely often is determined by their classification as Büchi and co-Büchi states, respectively.

□

We describe how to transform the intermediate 2AHW to 1NBW. In [Var98], Vardi translates 2-way alternating parity tree automata to 1-way nondeterministic parity tree automata. Since $\mathcal{H}_\psi$ can be defined as a parity automaton, and since words are a special case of trees, one could use the transformation in [Var98]. We describe here a simpler and more direct construction. We first need some notations.

Consider a 2AHW $A = \langle \Sigma, Q, q_0, \eta, F \rangle$. A *restriction* of A is a set $\xi \in 2^{Q \times \{-1,0,1\} \times Q}$. For a restriction $\xi \subseteq Q \times \{-1, 0, 1\} \times Q$, we define $state(\xi) = \{u : (u, \Delta, u') \in \xi\}$. A *strategy* for A is an infinite sequence $\tau = \xi_0, \xi_1, \ldots$ of restrictions. We sometimes denote ξ_i by $\tau(i)$. We say that the strategy τ is *on* a word w if $q_0 \in state(\xi_0)$, for all $i \in \mathbb{N}$ and states $q \in state(\xi_i)$, the set $\{(\Delta, q') | (q, \Delta, q') \in \xi_i\}$ satisfies $\eta(q, w_i)$, and for all $i \in \mathbb{N}$ and $(q, \Delta, q') \in \xi_i$ we have $q' \in state(\xi_{i+\Delta})$ (or $\eta(q', w_{i+\Delta}) = true$). Intuitively, a strategy suggests at each location i, a possible way for satisfying the transition function.

Lemma 1. *Consider a 2AHW $A = \langle \Sigma, Q, q_0, \eta, F \rangle$ with n states. There is a 1DBW A' over the alphabet $\Sigma \times 2^{Q \times \{-1,0,1\} \times Q}$ such that A' has $2^{O(n)}$ states and it accepts a word $(w_0, \xi_0) \cdot (w_1, \xi_1) \cdots$ iff $\xi_0, \xi_1, \cdots$ is a strategy for A on $w_0, w_1, \cdots$.*

Proof: The intuition is quite simple. When reading (w_i, ξ_i), the automaton A' has to remember two sets. The set of states that ξ_i restricts ($state(\xi_i)$) and the set of states that ξ_i promises that have a strategy from ξ_{i+1} (if $(q, 1, q') \in \xi_i$ then ξ_i promises that q' has a strategy from ξ_{i+1}). It then checks that the states

that are promised by ξ_{i+1} that have a strategy from ξ_i are indeed restricted by ξ_i and that all promises of ξ_i are indeed fulfilled. The local requirements, that the strategy fulfills the transition of A and that states that should be restricted by ξ_i are indeed restricted need no memory in order to be checked. The formal proof is ommitted. □

A *path* in a strategy τ is a finite or infinite sequence $(0, q_0), (i_1, q_1), (i_2, q_2), \ldots$ of pairs from $\mathbb{N} \times Q$ such that either the path is infinite, in which case for all $j \geq 0$, there is $\Delta_j \in \{-1, 0, 1\}$ such that $(q_j, \Delta_j, q_{j+1}) \in \tau(i_j)$ and $i_{j+1} = i_j + \Delta_j$, or the path is finite $(0, q_0), \ldots, (i_m, q_m)$, in which case for all $0 \leq j < m$, there is $\Delta_j \in \{-1, 0, 1\}$ such that $(q_j, \Delta_j, q_{j+1}) \in \tau(i_j)$, $i_{j+1} = i_j + \Delta_j$, and $\eta(q_m, w_{i_m}) = \mathit{true}$. An infinite path is *accepting* if it gets trapped in B and visits $\mathbb{N} \times F$ infinitely often or if it gets trapped in C and visits $\mathbb{N} \times F$ finitely often. A finite path is always *accepting*. We say that τ is *winning* if all infinite paths in τ are accepting. Otherwise, τ is *losing*. It is not hard to see that a 2AHW accepts a word iff it has a winning strategy on the word (c.f., [MS95,Var98,KV00]).

Lemma 2. *Consider a 2AHW $A = \langle \Sigma, Q, q_0, \eta, F \rangle$ with n states. There is a 2NBW A' over the alphabet $2^{Q \times \{-1,0,1\} \times Q}$ such that A' has $O(n)$ states and it accepts exactly all the losing strategies for A.*

Proof: We first define a 2NHW A'' that accepts exactly all the losing strategies of A. The automaton $A'' = \langle 2^{Q \times \{-1,0,1\} \times Q}, Q, q_0, \eta', F \rangle$, where the co-Büchi set of A'' is the Büchi set of A, the Büchi set of A'' is the co-Büchi set of A, and η' is defined for all $q \in Q$ and $\xi \in 2^{Q \times \{-1,0,1\} \times Q}$ as follows. Let $\mathit{options}(q, \xi) = \{(\Delta, q') \mid (q, \Delta, q') \in \xi\}$. Then,

$$\eta'(q, \xi) = \begin{cases} \mathbf{false} & \text{If } \mathit{options}(q, \xi) = \emptyset \\ \mathit{options}(q, \xi) & \text{Otherwise.} \end{cases}$$

Intuitively, when the automaton A'' reads a strategy τ, it guesses a path in τ that is not accepting. Accordingly, A'' rejects when the strategy reaches a location in which the set of restrictions is empty (this corresponds to the case where the candidate path is finite). When the candidate path is infinite, it is not accepting in τ iff it does not satisfy the acceptance condition of A, which is why A'' dualizes A.

Now, it is easy to translate the 2NHW A'' to a 2NBW A' with a linear blow up: whenever we are in a co-Büchi set S, we can nondeterministically move to a copy of the set in which only states from $S \setminus F$ are present. □

The automaton A' in Lemma 2 uses its bidirectionality in order to follow the strategy τ. This enables us to remove alternation, but still leaves us with bidirectionality. To remove the latter, we have to blow up the state space:

Lemma 3. [Var88] *Given a 2NBW A with n states, we can construct a 1NBW A' with $2^{O(n^2)}$ states such that $L(A') = \Sigma^\omega \setminus L(A)$.*

Intuitively, the $2^{O(n^2)}$ blow up follows from the fact we have to remember, for each pair of states $\langle q, q' \rangle$, the set of states visited between subsequent visits of the automaton in q and q'. We can now combine Lemmas 1, 2, and 3 to obtain our goal.

Theorem 2. *Given a 2AHW $\mathcal{H}$ over Σ, we can construct a 1NBW $\mathcal{A}$ with $2^{O(n^2)}$ states such that $L(\mathcal{H}) = L(\mathcal{A})$.*

Proof: Let $\mathcal{H} = \langle \Sigma, Q, q_0, \eta, F \rangle$. By Lemma 1, we can construct a 1DBW A_1 over the alphabet $\Sigma \times 2^{Q \times \{-1,0,1\} \times Q}$ such that A_1 has $2^{O(n)}$ states and it accepts a word $(w_0, \xi_0) \cdot (w_1, \xi_1) \cdots$ iff $\xi_0, \xi_1, \ldots$ is on $w_0, w_1, \ldots$. Also, by Lemmas 2 and 3, we can construct a 1NBW A_2 such that A_2 accepts a word over the alphabet $\Sigma \times 2^{Q \times \{-1,0,1\} \times Q}$ iff its projection on $2^{Q \times \{-1,0,1\} \times Q}$ is an accepting strategy. The automaton $\mathcal{A}$ is the intersection of A_1 and A_2, projected on Σ. □

We combine now the constructions described above and use the resulting 1NBW for solving the satisfiability and model-checking problems for ETL_{2a}. First, Theorems 1 and 2 immediately imply the following.

Theorem 3. *Given an ETL_{2a} formula ψ of length n, there is a 1NBW $\mathcal{A}_\psi$ such that $L(\mathcal{A}_\psi) = L(\psi)$ and $\mathcal{A}_\psi$ has $2^{O(n^2)}$ states.*

Once we construct $\mathcal{A}_\psi$, we can reduce satisfiability of ψ to nonemptiness of $\mathcal{A}_\psi$, and we can reduce model checking of a system S with respect to ψ to the language inclusion problem $L(S) \subseteq L(\mathcal{A}_\psi)$. (The system S is given as a finite state-transition graph, $L(S)$ is the set of all the words that S generates, and we say that S satisfies ψ if $(\pi, 0) \models \psi$ for all the words π that S generate.) As in LTL, we can use the fact that ETL_{2a} is closed under negation and check the latter by checking the emptiness of the intersection $S \times \mathcal{A}_{\neg\psi}$ [VW94]. Since the nonemptiness problem for Büchi automata can be solved in NLOGSPACE, we have the following (the lower bounds follow immediately from the lower bounds on LTL [SC85] and the linear translation of LTL to 1ABW [Var96]).

Theorem 4. *The satisfiability and the model-checking problems for ETL_{2a} are PSPACE-complete.*

It follows that in spite of the succinctness of two-way and alternating automata, the advantages of ETL_{2a} are obtained without a major increase in space complexity.

4 Discussion

We studied an extension of linear temporal logic with two-way alternating automata. The resulting logic ETL_{2a}, is as expressive as previous extensions of linear temporal logic with ω-regular automata, but the added strength of bidirectionality and alternation makes the logic substantially more convenient. The satisfiability and model-checking problems for ETL_{2a} are PSPACE-complete,

as is the case with LTL or weaker extensions of LTL with automata. There have been two recent developments that make us optimistic about the practicality of ETL_{2a}: the development of symbolic procedures for handling alternating automata [Fin01], and the usage of alternating automata as an intermediate formalism at Intel [AFF+01]. Using ETL_{2a}, it would be possible to extend this intermediate formalism to include convenient reference to past.

In this paper we considered the linear framework to verification. Branching temporal logic extends linear temporal logic with the path quantifiers A ("for all path") and E ("there exists a path"), and its formulas describe computation trees. The same limitation of LTL applies to its branching-time extension CTL*. Similar suggestions to extend the expressiveness of CTL* are studied in the literature. This includes both the extensions of the path formulas of CTL* with ω-regular word automata [VW84,CGK92], and the extension of the state formulas with ω-regular tree automata [MS85]. As in the linear framework, one can strengthen these extensions by using more powerful automata, in particular two-way and alternating ones. Since it is possible to remove bidirectionality and alternation also in the branching framework [Var98], our treatment of ETL_{2a} should work here as well. Its implementation, however, is going to be much more complicated in the branching framework.

References

[AFF+01] R. Armoni, L. Fix, A. Flaisher, R. Gerth, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, M.Y. Vardi, and Y. Zber. The ForSpec compiler. Submitted, 2001.

[AFG+01] R. Armoni, L. Fix, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, A. Tiemeyer, E. Singerman, and M.Y. Vardi. The ForSpec temporal logic: A new temporal property-specification logic. Submitted, 2001.

[BB87] B. Banieqbal and H. Barringer. Temporal logic with fixed points. In *Temporal Logic in Specification*, *LNCS* 398, 62–74. Springer-Verlag, 1987.

[BBG+94] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *6th CAV*, *LNCS* 818, 182–193, Springer-Verlag, 1994.

[BBL98] I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *10th CAV*, *LNCS* 1427, 184–194. Springer-Verlag, 1998.

[Bir93] J.C. Birget. State-complexity of finite-state devices, state compressibility and incompressibility. *Mathematical Systems Theory*, 26(3):237–269, 1993.

[BK85] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a framework. In *Seminar in Concurrency*, LNCS 197 , 35–61. Springer-Verlag, 1985.

[BL80] J.A. Brzozowski and E. Leiss. Finite automata and sequential networks. *TCS*, 10:19–35, 1980.

[CGK92] E.M. Clarke, O. Grumberg, and R.P. Kurshan. A synthesis of two approaches for verifying finite state concurrent systes. *Logic and Computation*, 2(5):605–618, 1992.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of the ACM*, 41(3):517–539, 1994.

[ET97] E.A. Emerson and R.J. Trefler. Generalized quantitative temporal reasoning: An automata theoretic approach. In *TAPSOFT*, *LNCS* 1214, 189–200. Springer, 1997.

[Fin01] B. Finkbeiner. Symbolic refinement checking with nondeterministic BDDs. In *TACAS*, LNCS 2031. Springer-Verlag, 2001.

[Fra92] N. Francez. *Program verification*. Int. Computer Science. Addison-Weflay, 1992.

[Gab87] D. Gabbay. The declarative past and imperative future. In *Temporal Logic in Specification*, *LNCS* 398 , 407–448. Springer-Verlag, 1987.

[GH96] N. Globerman and D. Harel. Complexity results for two-way and multi-pebble automata and their logics. *TCS*, 143:161–184, 1996.

[HT99] J.G. Henriksen and P.S. Thiagarajan. Dynamic linear time temporal logic. *Annals of Pure and Applied Logic*, 96(1–3):187–207, 1999.

[KV97] O. Kupferman and M.Y. Vardi. Weak alternating automata are not that weak. In *5th ISTCS*, 147–158. IEEE Computer Society Press, 1997.

[KV00] O. Kupferman and M.Y. Vardi. μ-calculus synthesis. In *25th MFCS*, *LNCS* 1893, 497–507. Springer-Verlag, 2000.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, *LNCS 193*, 196–218, Springer-Verlag, 1985.

[Mey75] A. R. Meyer. Weak monadic second order theory of successor is not elementary recursive. In *Proc. Logic Colloquium*, Vol. 453 of *Lecture Notes in Mathematics*, 132–154. Springer-Verlag, 1975.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[MS85] D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *TCS*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *TCS*, 54:267–276, 1987.

[MS95] D.E. Muller and P.E. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of theorems of Rabin, McNaughton and Safra. *TCS*, 141:69–107, 1995.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *13th ICALP*, *LNCS* 226, 1986.

[Pit00] N. Piterman. Extending temporal logic with ω-automata. M.Sc. Thesis, The Weizmann Institute of Science, Israel, 2000, `http://www.wisdom.weizmann.ac.il/home/nirp/public_html/publications/msc_thesis.ps`.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *TCS*, 13:45–60, 1981.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144. Springer-Verlag, 1985.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *TCS*, 49:217–237, 1987.

[Tho81] W. Thomas. A combinatorial approach to the theory of ω-automata. *Information and Computation*, 48:261–283, 1981.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In *15th POPL*, pages 250–259, 1988.

[Var96] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, *LNCS* 1043, 238–266, 1996.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *25th ICALP LNCS* 1443, 628–641. Springer-Verlag, 1998.

[VW84] M.Y. Vardi and P. Wolper. Yet another process logic. In *Logics of Programs*, *LNCS* 164, 501–512. Springer-Verlag, 1984.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

Symbolic Algorithms for Infinite-State Games*

Luca de Alfaro Thomas A. Henzinger Rupak Majumdar

Electrical Engineering and Computer Sciences, University of California, Berkeley
{dealfaro,tah,rupak}@eecs.berkeley.edu

Abstract. A procedure for the analysis of state spaces is called *symbolic* if it manipulates not individual states, but sets of states that are represented by constraints. Such a procedure can be used for the analysis of *infinite* state spaces, provided termination is guaranteed. We present symbolic procedures, and corresponding termination criteria, for the solution of *infinite-state games*, which occur in the control and modular verification of infinite-state systems. To characterize the termination of symbolic procedures for solving infinite-state games, we classify these game structures into four increasingly restrictive categories:

1. Class 1 consists of infinite-state structures for which all safety and reachability games can be solved.
2. Class 2 consists of infinite-state structures for which all ω-regular games can be solved.
3. Class 3 consists of infinite-state structures for which all nested positive boolean combinations of ω-regular games can be solved.
4. Class 4 consists of infinite-state structures for which all nested boolean combinations of ω-regular games can be solved.

We give a structural characterization for each class, using *equivalence relations* on the state spaces of games which range from game versions of trace equivalence to a game version of bisimilarity. We provide infinite-state examples for all four classes of games from control problems for *hybrid systems*. We conclude by presenting symbolic algorithms for the *synthesis* of winning strategies ("controller synthesis") for infinite-state games with arbitrary ω-regular objectives, and prove termination over all class-2 structures. This settles, in particular, the symbolic controller synthesis problem for rectangular hybrid systems.

1 Introduction

While algorithmic methods ("model checking") were originally invented for the analysis of finite-state systems, much recent interest has concerned the application of such methods to *infinite-state* systems. There are two kinds of approaches. Approaches of the first kind reduce an infinite-state system to an "equivalent" finite-state system, and then explore the resulting finite state space (e.g., the region-graph method for timed automata [2]). We call these approaches *reductionist*. Approaches of the second kind explore the infinite state space directly, by

* This research was supported in part by the AFOSR MURI grant F49620-00-1-0327, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the NSF Theory grant CCR-9988172, and the NSF ITR grant CCR-0085949.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 536–550, 2001.
© Springer-Verlag Berlin Heidelberg 2001

manipulating constraints that may represent infinite state sets (e.g., the clock-zone method for timed automata [12]). We call these approaches *symbolic*. While perhaps optimal in theoretical complexity, reductionist approaches usually experience state explosion, and are typically outperformed in practice by symbolic approaches. In fact, the state-space partition induced by the execution of a symbolic method is often much coarser than the partition corresponding to the "equivalent" finite-state system. As they operate on infinite sets of states and constraints, the main concern with symbolic approaches is *termination*. We refer to procedures that may or may not terminate as *semi-algorithms*.

The control and modular verification of systems can be studied as games played on state spaces, where the players represent controller vs. plant, or individual processes (see, e.g., [3]). The control and modular verification of infinite-state systems, accordingly, give rise to infinite-state games. In this paper, we present symbolic semi-algorithms for solving two-player *concurrent games on infinite state spaces*, and for synthesizing the corresponding winning strategies. A concurrent game is played in rounds. In each round, both players simultaneously and independently choose moves, and the choice of moves determines a set of possible next states (games in which the players take turns are a special case [3]). We consider ω-regular as well as nested winning conditions, such as "Player 1 has a strategy to reach an observable p from which player 2 cannot reach an observable q." We establish a set of criteria for the termination of the semi-algorithms, leading to a classification of infinite-state games.

Symbolic methods for games are based on the *controllable precondition* operator $CPre_i$, for $i = 1, 2$ [3]: for a set σ of states, $CPre_i(\sigma)$ contains those states from which player i can force the game into σ in a single round by choosing an appropriate move. We show that termination of $CPre_i$-based semi-algorithms can be studied by reasoning about various equivalence relations on the states of an infinite game structure, ranging from two-player versions of trace equivalence to a two-player version of bisimilarity. First, we argue that the semi-algorithms for solving games with specific winning conditions can be seen as instances of generic closure semi-algorithms, which refine a partition of the state space by applying the $CPre_i$ operators together with various boolean operators (such as set union, intersection, and difference). Hence, if the closure semi-algorithm terminates, so do the semi-algorithms for solving the corresponding games. Second, we show that the closure semi-algorithms terminate exactly when certain equivalence relations on the infinite state space have finite index. Thus, to obtain symbolic decision procedures for infinite-state games, it suffices that the corresponding equivalence relations have finite index.

Accordingly, we propose a classification of infinite-state game structures, depending on which equivalence relations have finite index. The classification parallels the classification of infinite-state transition systems presented in [11]. The first class of infinite-state game structures are those with finite *i-bounded-reach equivalence* quotients, for $i = 1, 2$: two states are i-bounded-reach equivalent if from either state, player i can force the game to the same observables in the same number of rounds. On these infinite-state structures, we can symbolically solve

games with safety and reachability objectives, by iterating $CPre_i$ a finite number of times. Game structures of the second class have finite *i-trace equivalence* quotients: state s is i-trace contained by state t if for every player-i strategy from s, there is a player-i strategy from t such that every possible outcome of the game (i.e., sequence of observables) from t is also a possible outcome from s (this is the *player-i alternating trace containment* of [4]). On these infinite-state structures, we can symbolically solve all games with ω-regular winning conditions, by appropriately iterating $CPre_i$, set union, and restricted intersection with observables. Game structures of the third class have finite *i-similarity* (or *alternating similarity* [4]) quotients, which permits symbolic model checking for all negation-free properties of the *game calculus*, a fixpoint logic with $CPre_i$, union, and (unrestricted) intersection operators. Finally, the fourth class contains the game structures with finite *i-bisimilarity* (or *alternating bisimilarity* [4]) quotients. They permit symbolic model checking for the full game calculus (with negation). Examples of infinite-state games from all four classes can be drawn from real-time and hybrid systems: networks of timed games, rectangular games [10], 2D rectangular games, and timed games [15] fall into the classes 1 to 4, in that order.

The termination criteria for solving games are insufficient if we wish to synthesize the corresponding winning strategies, which is important in control applications [18]. This is because for different states in $CPre_i(\sigma)$, player i may have to choose different moves to force the game into σ. However, if the set of possible moves is finite, then this problem can be overcome. We show how winning strategies can be synthesized symbolically over all class-2 game structures (finite i-trace equivalence) for all ω-regular winning conditions. Previously, symbolic infinite-state controller synthesis has been solved only for the special case of *timed games* [15], which fall into the more restrictive class 4 (finite i-bisimilarity). In particular, as an instance of our results, we obtain symbolic algorithms also for the control and controller synthesis of *rectangular hybrid systems*, a problem that was left open in [10] (where a reductionist solution is given). These symbolic algorithms can be executed directly by symbolic model checkers for hybrid systems, such as HyTech [9].

2 Symbolic Game Structures

A (two-player) *game structure*[1] $G = (S, A, \Gamma_1, \Gamma_2, \delta, P, \ulcorner\cdot\urcorner)$ consists of a (possibly infinite) set S of *states*, a finite set A of *actions*, two action assignments $\Gamma_1, \Gamma_2 : S \to 2^A \setminus \emptyset$ which define for each state nonempty sets of actions available to player 1 and player 2, a partial *transition* function $\delta : S \times A \times A \to S$ which associates with each state s and each pair of actions $a_1 \in \Gamma_1(s)$ and $a_2 \in \Gamma_2(s)$ a successor state, a finite set P of *observables*, and an *observation* function $\ulcorner\cdot\urcorner : P \to 2^S$ which associates with each observable a set of states. We require that for each observable $p \in P$, there is a complementary observable $\overline{p} \in P$ such that $\ulcorner\overline{p}\urcorner = S \setminus \ulcorner p\urcorner$. Intuitively, at state s, player 1 chooses an action a_1

[1] The multiple-player case is an immediate generalization.

from $\Gamma_1(s)$ and, simultaneously and independently, player 2 chooses an action a_2 from $\Gamma_2(s)$. Then, the game proceeds to $\delta(s, a_1, a_2)$.

Given two states $s, t \in S$ and actions $a_1 \in \Gamma_1(s)$ and $a_2 \in \Gamma_2(s)$, the state $\delta(s, a_1, a_2)$ is called the (a_1, a_2)-*successor* of s. A *source-s run* of the game structure G is an infinite sequence $s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$ of alternating states and action pairs such that $s_0 = s$ and for all $j \geq 0$, the state s_{j+1} is the (a_j, b_j)-successor of s_j. A *source-s trace* of G is an infinite sequence $P_0P_1P_2 \ldots$ of sets of observables for which there is a source-s run $s_0(a_0, b_0)s_1(a_1, b_1) \ldots$ such that $P_j = \{p \in P \mid s_j \in \ulcorner p \urcorner\}$ for all $j \geq 0$. A *strategy of player i*, for $i = 1, 2$, is a function $f_i : S^+ \to 2^A$ such that $\emptyset \subsetneq f_i(w \cdot s) \subseteq \Gamma_i(s)$ for every state sequence $w \in S^*$ and every state $s \in S$. Let f_1 and f_2 be a pair of strategies for player 1 and player 2. The *outcome* $\mathcal{L}_{f_1,f_2}(s)$ from state $s \in S$ of strategies f_1 and f_2 is a subset of the source-s runs of G: a run $s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$ belongs to $\mathcal{L}_{f_1,f_2}(s)$ if $s_0 = s$ and for all $j \geq 0$, we have $a_j \in f_1(s_0s_1 \cdots s_j)$ and $b_j \in f_2(s_0s_1 \cdots s_j)$ and $s_{j+1} = \delta(s_j, a_j, b_j)$. We write $L_{f_1,f_2}(s)$ for the set of source-s traces that correspond to runs in $\mathcal{L}_{f_1,f_2}(s)$.

2.1 Region algebras for game structures

A *symbolic theory* for the game structure G consists of a (possibly infinite) set R of *regions* together with a function $\ulcorner \cdot \urcorner : R \to 2^S$ which maps each region σ to the (possibly infinite) set of states represented by σ, such that the following four conditions hold:

1. Each observable is a region; that is, $P \subseteq R$. Furthermore, the function $\ulcorner \cdot \urcorner$ agrees on P with the definition of G. There are regions $\mathit{True}, \mathit{False} \in R$ such that $\ulcorner \mathit{True} \urcorner = S$ and $\ulcorner \mathit{False} \urcorner = \emptyset$.
2. For each pair $\sigma, \tau \in R$ of regions, there are regions $\mathit{And}(\sigma, \tau) \in R$, $\mathit{Or}(\sigma, \tau) \in R$, and $\mathit{Diff}(\sigma, \tau) \in R$ such that $\ulcorner \mathit{And}(\sigma, \tau) \urcorner = \ulcorner \sigma \urcorner \cap \ulcorner \tau \urcorner$, $\ulcorner \mathit{Or}(\sigma, \tau) \urcorner = \ulcorner \sigma \urcorner \cup \ulcorner \tau \urcorner$, and $\ulcorner \mathit{Diff}(\sigma, \tau) \urcorner = \ulcorner \sigma \urcorner \setminus \ulcorner \tau \urcorner$. Furthermore, the functions $\mathit{And}, \mathit{Or}, \mathit{Diff} : R \times R \to R$ are computable.
3. For each region $\sigma \in R$ and each pair $a, b \in A$ of actions, there is a region $\mathit{Pre}^{a,b}(\sigma) \in R$ such that $\ulcorner \mathit{Pre}^{a,b}(\sigma) \urcorner = \{s \in S \mid a \in \Gamma_1(s) \text{ and } b \in \Gamma_2(s) \text{ and } \delta(s, a, b) \in \ulcorner \sigma \urcorner\}$. Furthermore, the function $\mathit{Pre} : R \times A \times A \to R$ is computable. Using boolean operations and Pre, we can compute the functions $\mathit{CPre}_I : R \to R$ on regions, for $I = 1, 2, \{1, 2\}$, such that
$$\ulcorner \mathit{CPre}_1(\sigma) \urcorner = \{s \in S \mid \exists a \in \Gamma_1(s). \forall b \in \Gamma_2(s). \delta(s, a, b) \in \ulcorner \sigma \urcorner\};$$
$$\ulcorner \mathit{CPre}_2(\sigma) \urcorner = \{s \in S \mid \exists a \in \Gamma_2(s). \forall b \in \Gamma_1(s). \delta(s, b, a) \in \ulcorner \sigma \urcorner\};$$
$$\ulcorner \mathit{CPre}_{\{1,2\}}(\sigma) \urcorner = \{s \in S \mid \exists a \in \Gamma_1(s). \exists b \in \Gamma_2(s). \delta(s, a, b) \in \ulcorner \sigma \urcorner\}.$$
In particular, the region $\mathit{CPre}_1(\sigma)$ represents the states from which player 1 can force the game in one step into the region σ, no matter which action player 2 chooses. The region $\mathit{CPre}_{\{1,2\}}(\sigma)$ represents the states from which the two players can collaborate to force the game in one step into σ.
4. All emptiness and membership questions about regions can be decided; that is, there are computable functions $\mathit{Empty} : R \to \mathbb{B}$ and $\mathit{Member} : S \times R \to \mathbb{B}$ such that (a) $\mathit{Empty}(\sigma)$ iff $\ulcorner \sigma \urcorner = \emptyset$, and (b) $\mathit{Member}(s, \sigma)$ iff $s \in \ulcorner \sigma \urcorner$.

The tuple $(R, P, And, Or, Diff, Pre, Empty)$ is called a *region algebra* for G. A *symbolic semi-algorithm* on game structures takes as input a region algebra for a game structure G and generates, starting from the observables P and constants *True*, *False*, regions in R by repeatedly applying the operations *And*, *Or*, *Diff*, *Pre*, and *Empty*.

Example 1. Consider the symbolic semi-algorithm Reach_1:

$$T_0 := p; \textbf{ for } j = 0, 1, 2, \ldots \textbf{ do } T_{j+1} := Or(T_j, CPre_1(T_j)) \textbf{ until } T_{j+1} \subseteq T_j$$

which computes, for an observable $p \in P$, the region $CPre_1^*(p)$ of states from which player 1 can force the game in some number of steps into a p-state. The termination test $T \subseteq T'$ is decided by checking that $Empty(And(T, Diff(True, T')))$. While each individual operation is computable, depending on G, the iteration of operations may or may not terminate. □

2.2 Equivalences on game structures

State equivalences. A *state equivalence* $\cong$ is a family of relations which contains for each game structure G an equivalence relation $\cong_G$ on the states of G. The $\cong$*-equivalence problem* for a class C of game structures asks, given two states s and t of a game structure G from the class C, whether $s \cong_G t$. The state equivalence $\cong$ is *as coarse as* the state equivalence $\cong'$ if $s \cong_G t$ implies $s \cong'_G t$ for all game structures G. The equivalence $\cong$ is *coarser than* $\cong'$ if $\cong$ is as coarse as $\cong'$, but $\cong'$ is not as coarse as $\cong$. Given a game structure $G = (S, A, \Gamma_1, \Gamma_2, \delta, P, \ulcorner\cdot\urcorner)$ and a state equivalence $\cong$, the *quotient structure* is the game structure $G/_\cong = (S/_\cong, A, \Gamma_1, \Gamma_2, \delta/_\cong, P, \ulcorner\cdot\urcorner/_\cong)$, where $G/_\cong$ is the set of equivalence classes of $\cong_G$, and $\tau \in \delta/_\cong(\sigma, a_1, a_2)$ if there is a state $s \in \sigma$ and a state $t \in \tau$ such that $t = \delta(s, a_1, a_2)$, and $\sigma \in \ulcorner p \urcorner/_\cong$ if there is a state $s \in \sigma$ such that $s \in \ulcorner p \urcorner$. The quotient construction is of particular interest to us when it transforms an infinite-state structure G into a finite-state structure $G/_\cong$.

Simulation-based equivalences. A binary relation $\preceq \subseteq S \times S$ is a *1-simulation*[2] if $s \preceq t$ implies the following two conditions:

(1) For each observable $p \in P$, if $s \in \ulcorner p \urcorner$, then $t \in \ulcorner p \urcorner$.
(2.1) For each $a_1 \in \Gamma_1(s)$, there is $a_2 \in \Gamma_1(t)$ such that for all $b_2 \in \Gamma_2(t)$ there is $b_1 \in \Gamma_2(s)$ with $\delta(s, a_1, b_1) \preceq \delta(t, a_2, b_2)$.

By exchanging the subscripts 1 and 2 in condition (2.1), we obtain condition (2.2). The relation $\preceq$ is a *2-simulation* if $s \preceq t$ implies the dual conditions (1) and (2.2). The relation $\preceq$ is a $\{1,2\}$*-simulation* if $s \preceq t$ implies all three conditions (1), (2.1), and (2.2). For $I = 1, 2, \{1, 2\}$, the state s *is* I*-simulated* by t, in symbols $s \preceq_I^S t$, if there is an I-simulation $\preceq$ such that $s \preceq t$. We write $s \cong_I^S t$ if both $s \preceq_I^S t$ and $t \preceq_I^S s$. The state equivalence $\cong_I^S$ is called I*-similarity.* We note that two states may be both 1-similar and 2-similar, but not $\{1,2\}$-similar (see Figure 1). A binary relation $\cong \subseteq S \times S$ is an I*-bisimulation* if $\cong$ is a symmetric

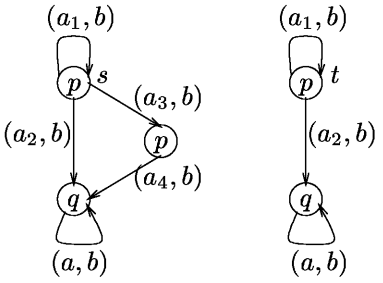


Fig. 1. The states s and t are both 1-similar and 2-similar (hence 1-trace and 2-trace equivalent), but not equivalent with respect to all DGμ formulas (hence not $\{1,2\}$-similar).

I-simulation. We define $s \cong_I^B t$ if there is an I-bisimulation $\cong$ such that $s \cong t$. The state equivalence $\cong_I^B$ is called *I-bisimilarity.*

Trace-based equivalences. A binary relation $\preceq \subseteq S \times S$ is a *1-trace containment*[3] if $s \preceq t$ implies that for all strategies f_1 of player 1, there exists a strategy g_1 of player 1 such that for all strategies g_2 of player 2, there exists a strategy f_2 of player 2 such that

(3) $L_{g_1,g_2}(t) \subseteq L_{f_1,f_2}(s)$.

Given a trace $\xi = P_0 P_1 P_2 \ldots$ and an observation $p \in P$, let $bnd(\xi, p)$ be the smallest $j \geq 0$ such that $p \in P_j$, and undefined if no such j exists. The relation $\preceq$ is a *1-bounded-reach containment* if condition (3) is replaced by

(4) for every trace $\xi \in L_{g_1,g_2}(t)$ and observation $p \in P$, if $bnd(\xi, p)$ is defined, then there is a trace $\xi' \in L_{f_1,f_2}(s)$ with $bnd(\xi', p) = bnd(\xi, p)$.

We define $s \preceq_1^L t$ (respectively, $s \preceq_1^R t$) if there is a 1-trace containment (respectively, 1-bounded-reach containment) $\preceq$ such that $s \preceq t$. We write $s \cong_1^L t$ if both $s \preceq_1^L t$ and $t \preceq_1^L s$, and $s \cong_1^R t$ if both $s \preceq_1^R t$ and $t \preceq_1^R s$. The state equivalences $\cong_1^L$ and $\cong_1^R$ are called *1-trace equivalence* and *1-bounded-reach equivalence*, respectively. The 1-bounded-reach equivalence characterizes termination of reachability questions on a game structure: it can be shown that the symbolic semi-algorithm Reach_1 terminates on a region algebra of G for all observables $p \in P$ iff the 1-bounded-reach equivalence of G has finite index.

The expected relationships between these state equivalences hold. For example, 1-bounded-reach equivalence is coarser than 1-trace equivalence, which is coarser than 1-similarity, which is coarser than 1-bisimilarity [4]. Also, standard trace equivalence (respectively, similarity; bisimilarity), as interpreted on the transition structure that underlies G, is coarser than 1-trace equivalence (respectively, 1-similarity; 1-bisimilarity) [4].

[2] This is the *alternating simulation* of [4].

[3] This is the *alternating trace containment* of [4].

2.3 Fixpoint calculi for game structures

State logics. A *state logic* Φ is a logic whose formulas are interpreted over the states of game structures; that is, for every Φ-formula φ and every game structure G, there is a set $[\![\varphi]\!]_G$ of states of G which satisfy φ. The *Φ model-checking problem* for a class C of game structures asks, given a Φ-formula φ and a state s of a game structure G from the class C, whether $s \in [\![\varphi]\!]_G$. Two formulas φ and ψ of state logics are *equivalent* if $[\![\varphi]\!]_G = [\![\psi]\!]_G$ for all game structures G. The state logic Φ is *as expressive as* the state logic Φ' if for every Φ'-formula φ, there is a Φ-formula ψ which is equivalent to φ. The logic Φ is *more expressive than* Φ' if Φ is as expressive as Φ', but Φ' is not as expressive as Φ. Every state logic Φ *induces* a state equivalence, denoted $\cong^{\Phi}$: for all states s and t of a game structure G, define $s \cong^{\Phi} t$ if for all Φ-formulas φ, we have $s \in [\![\varphi]\!]_G$ iff $t \in [\![\varphi]\!]_G$. The state logic Φ *admits abstraction* if for every Φ-formula φ and every game structure G, we have $[\![\varphi]\!]_G = \bigcup\{\sigma \mid \sigma \in [\![\varphi]\!]_{G/\cong^{\Phi}}\}$; that is, a state s of G satisfies an Φ-formula φ iff the $\cong^{\Phi}$ equivalence class of s satisfies φ in the quotient structure. Consequently, if Φ admits abstraction, then every Φ model-checking question on a game structure G can be reduced to an Φ model-checking question on the induced quotient structure $G/\cong^{\Phi}$. Below, we shall repeatedly prove the Φ model-checking problem for a class C to be decidable by observing that for every game structure G from C, the quotient structure $G/\cong^{\Phi}$ has finitely many states and can be constructed effectively.

Example 2. Given an observation $p \in P$, let $\Diamond p$ be the set of traces ξ such that $bnd(\xi, p)$ is defined; that is, p occurs in ξ. The *controllability formula* $\langle\langle 1 \rangle\rangle \Diamond p$ is true at the states from which player 1 has a strategy to control the game to reach a p-state; that is, there is a strategy f_1 of player 1 such that for all strategies f_2 of player 2, we have $L_{f_1,f_2}(s) \subseteq \Diamond p$. Both safety and reachability control problems can be expressed as boolean combinations of controllability formulas. The semi-algorithm Reach_1 of Example 1 provides a symbolic model-checking procedure for controllability formulas. From the characterization of Section 2.2 we conclude that the model-checking problem for controllability formulas is decidable for all game structures that have symbolic theories and 1-bounded-reach equivalences with finite index. An example of infinite-state game structures with symbolic theories and finite 1-bounded-reach equivalences are *networks of timed games*, a two-player version of networks of timed automata [1]. □

Game calculus. The *formulas* are generated by the grammar

$$\varphi \ ::= \ p \mid \neg p \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\langle I \rangle\rangle \bigcirc \varphi \mid [\![I]\!] \bigcirc \varphi \mid (\mu x \colon \varphi) \mid (\nu x \colon \varphi),$$

for constants p from some set Π, variables x from some set X, and teams $I = 1, 2, \{1, 2\}$. Let $G = (S, A, \Gamma_1, \Gamma_2, \delta, P, \ulcorner \cdot \urcorner)$ be a game structure whose observables include all constants; that is, $\Pi \subseteq P$. Let $\mathcal{E} : X \to 2^S$ be a mapping from the variables to sets of states. We write $\mathcal{E}[x \mapsto \rho]$ for the mapping that agrees with $\mathcal{E}$ on all variables, except that $x \in X$ is mapped to $\rho \subseteq S$. Given G and $\mathcal{E}$, every formula φ defines a set $[\![\varphi]\!]_{G,\mathcal{E}} \subseteq S$ of states:

$$[\![p]\!]_{G,\mathcal{E}} = \ulcorner p \urcorner;$$
$$[\![\neg p]\!]_{G,\mathcal{E}} = \ulcorner \overline{p} \urcorner;$$
$$[\![x]\!]_{G,\mathcal{E}} = \mathcal{E}(x);$$
$$[\![\varphi_1 \{{}^{\vee}_{\wedge}\} \varphi_2]\!]_{G,\mathcal{E}} = [\![\varphi_1]\!]_{G,\mathcal{E}} \{{}^{\cup}_{\cap}\} [\![\varphi_2]\!]_{G,\mathcal{E}};$$
$$[\![\{{}^{\langle\!\langle 1 \rangle\!\rangle}_{[\![1]\!]}\} \bigcirc \varphi]\!]_{G,\mathcal{E}} = \{s \in S \mid \{{}^{\exists a \in \Gamma_1(s).\ \forall b \in \Gamma_2(s).}_{\forall a \in \Gamma_1(s).\ \exists b \in \Gamma_2(s).}\} \delta(s,a,b) \in [\![\varphi]\!]_{G,\mathcal{E}}\};$$
$$[\![\{{}^{\langle\!\langle 1,2 \rangle\!\rangle}_{[\![1,2]\!]}\} \bigcirc \varphi]\!]_{G,\mathcal{E}} = \{s \in S \mid \{{}^{\exists a \in \Gamma_1(s).\ \exists b \in \Gamma_2(s).}_{\forall a \in \Gamma_1(s).\ \forall b \in \Gamma_2(s).}\} \delta(s,a,b) \in [\![\varphi]\!]_{G,\mathcal{E}}\};$$
$$[\![\{{}^{\mu}_{\nu}\} x\colon \varphi]\!]_{G,\mathcal{E}} = \{{}^{\cap}_{\cup}\}\{\rho \subseteq S \mid \rho = [\![\varphi]\!]_{G,\mathcal{E}[x \mapsto \rho]}\}.$$

Note that the team operator $\langle\!\langle 1,2 \rangle\!\rangle \bigcirc$ corresponds to the existential next operator $\exists \bigcirc$, as interpreted on the transition structure that underlies G. If we restrict ourselves to the closed formulas, then we obtain a state logic, called *game calculus*[4] and denoted Gμ: define $[\![\varphi]\!]_G$ as $[\![\varphi]\!]_{G,\mathcal{E}}$ for any $\mathcal{E}$. The player-1 fragment of Gμ, which restricts all teams to $I = 1$, is called the *1-game calculus* and denoted 1-Gμ. The fragment $\{1,2\}$-Gμ, which restricts all teams to $I = \{1,2\}$, is the standard μ-calculus [14].

Proposition 1. *The state equivalence induced by* Gμ *(respectively, 1-*Gμ*) is* $\{1,2\}$*-bisimilarity (respectively, 1-bisimilarity).*

It can be shown that the game calculus Gμ admits abstraction. The definition of Gμ naturally suggests a model-checking method over finite-state game structures, where each fixpoint can be computed by successive approximation. The symbolic semi-algorithm ModelCheck of Figure 2 applies this method to infinite-state game structures. Suppose that the input given to ModelCheck is the region algebra of a game structure G, the Gμ-formula φ, and any mapping $E\colon X \to 2^R$ from the variables to sets of regions. Then for each recursive call of ModelCheck, each T_j, for $j \geq 0$, is a region from R, and each recursive call returns a region from R. Furthermore, if it terminates, then ModelCheck returns a region $[\varphi]_E$ such that $[\varphi]_E = [\![\varphi]\!]_{G,\mathcal{E}}$, where $\mathcal{E}(x) = \bigcup\{\ulcorner \sigma \urcorner \mid \sigma \in E(x)\}$ for all $x \in X$. In particular, if φ is closed, then a state s of G satisfies φ iff $Member(s, [\varphi]_E)$.

Negation-free game calculus. The formulas of the *negation-free game calculus*, denoted NGμ, are the boolean combinations of Gμ-formulas generated by the grammar

$$\varphi ::= p \mid \neg p \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\!\langle I \rangle\!\rangle \bigcirc \varphi \mid (\mu x\colon \varphi) \mid (\nu x\colon \varphi).$$

No negations are permitted in the scope of team operators, all of which have the form $\langle\!\langle I \rangle\!\rangle$, except in front of observables. Consequently, team operators can be nested only if they have the same force (either $\langle\!\langle\rangle\!\rangle$ or $[\![]\!]$ force). The player-1 fragment of NGμ, which restricts all teams to $I = 1$, is called the *negation-free 1-game calculus* and denoted 1-NGμ. The fragment $\{1,2\}$-NGμ, which restricts all teams to $I = \{1,2\}$, is equivalent to the boolean combinations of existential and universal μ-calculus formulas, which include $\exists$CTL and $\forall$CTL.

Proposition 2. *The state equivalence induced by* NGμ *(respectively, 1-*NGμ*) is* $\{1,2\}$*-similarity (respectively, 1-similarity).*

[4] This is the *alternating-time μ-calculus* of [3].

Symbolic semi-algorithm ModelCheck
Input: a region algebra $(R, P, And, Or, Diff, Pre, Empty)$, a formula $\varphi \in \text{G}\mu$, and a mapping E with domain X.

Output: $[\varphi]_E :=$
if $\varphi = p$ **then return** p;
if $\varphi = \neg p$ **then return** $\overline{p}$;
if $\varphi = x$ **then return** $E(x)$;
if $\varphi = (\varphi_1 \vee \varphi_2)$ **then return** $Or([\varphi_1]_E, [\varphi_2]_E)$;
if $\varphi = (\varphi_1 \wedge \varphi_2)$ **then return** $And([\varphi_1]_E, [\varphi_2]_E)$
if $\varphi = \langle\langle I \rangle\rangle \bigcirc \varphi'$ **then return** $CPre_I([\varphi']_E)$;
if $\varphi = [\![I]\!] \bigcirc \varphi'$ **then return** $Diff(True, CPre_I(Diff(True, [\varphi']_E)))$;
if $\varphi = (\mu x \colon \varphi')$ **then**
  $T_0 := False$;
  for $j = 0, 1, 2, \ldots$ **do** $T_{j+1} := [\varphi']_{E[x \mapsto T_j]}$ **until** $T_{j+1} \subseteq T_j$;
  return T_j;
if $\varphi = (\nu x \colon \varphi')$ **then**
  $T_0 := True$;
  for $j = 0, 1, 2, \ldots$ **do** $T_{j+1} := [\varphi']_{E[x \mapsto T_j]}$ **until** $T_{j+1} \supseteq T_j$;
  return T_j.

Fig. 2. Model checking.

Deterministic game calculus. The formulas of the *deterministic game calculus*, denoted $\text{DG}\mu$, are the boolean combinations of $\text{G}\mu$-formulas generated by the grammar

$$\varphi \ ::= \ p \mid \neg p \mid x \mid \varphi \vee \varphi \mid p \wedge \varphi \mid \langle\langle I \rangle\rangle \bigcirc \varphi \mid (\mu x \colon \varphi) \mid (\nu x \colon \varphi).$$

Note that in deterministic formulas, one argument of each conjunction is an observable. The player-1 fragment of $\text{DG}\mu$, which restricts all teams to $I = 1$, is called the *deterministic 1-game calculus* and denoted 1-$\text{DG}\mu$. The fragment $\{1,2\}$-$\text{DG}\mu$, which restricts all teams to $I = \{1, 2\}$, corresponds to the boolean combinations of existential and universal ω-regular trace properties [11], which include LTL. We can characterize the expressive power of 1-$\text{DG}\mu$ similarly. Let 1-$\text{G}\omega$ be the state logic that consists of all formulas of the form $\langle\langle 1 \rangle\rangle K$, where K is an ω-regular expression with constants from Π [19]. We identify K with the set of infinite words over the alphabet 2^Π that satisfy K. Let G be a game structure whose observables contain Π. A state s of G is in $[\![\langle\langle 1 \rangle\rangle K]\!]_G$ if player 1 has a strategy f_1 such that for all strategies f_2 of player 2, we have $L_{f_1, f_2}(s) \subseteq K$. Given a formula φ of 1-$\text{DG}\mu$, we can inductively construct an ω-regular expression K_φ such that $\langle\langle 1 \rangle\rangle K_\varphi$ and φ are equivalent [8]. In Section 4, we will show conversely that every 1-$\text{G}\omega$ formula can be translated into an equivalent formula of 1-$\text{DG}\mu$.

Theorem 1. *The state logics 1-*$\text{DG}\mu$ *and 1-*$\text{G}\omega$ *are equally expressive.*

Corollary 1. *The state equivalence induced by 1-*$\text{DG}\mu$ *is 1-trace equivalence.*

It is an open problem to characterize the state equivalence induced by the full deterministic game calculus DGμ, which is strictly finer than the intersection of $\cong_1^L$ and $\cong_2^L$ (see Figure 1).

3 Three Symbolic Semi-algorithms on Game Structures

We define three closure semi-algorithms, and we characterize their termination in terms of three state equivalences. The three closure semi-algorithms compute regions that are also computed by the symbolic semi-algorithm ModelCheck on certain inputs. Thus, the closure semi-algorithms enable us to study the termination of ModelCheck on classes of input structures and input formulas. They enable us to separate termination concerns from partial-correctness concerns, such as the solution of LTL games using ModelCheck. In particular, partial-correctness arguments can often follow the corresponding proofs for finite-state games.

3.1 Observation refinement

The symbolic semi-algorithm OR, called *observation refinement*, on a region algebra starts from the finite set $T_0 := P$ of observables and generates inductively the finite sets of regions

$$T_{j+1} = T_j \cup \{CPre_1(\sigma), CPre_2(\sigma), CPre_{\{1,2\}}(\sigma) \mid \sigma \in T_j\} \\ \cup \{Or(\sigma,\tau) \mid \sigma,\tau \in T_j\} \cup \{And(\sigma,p) \mid \sigma \in T_j \text{ and } p \in P\}$$

for $j \geq 0$. Note that OR applies only a restricted form of the *And* operation: one argument is always an observable. Let $\ulcorner T \urcorner$ denote the set $\{\ulcorner \sigma \urcorner \mid \sigma \in T\}$. The semi-algorithm OR terminates iff there is a j such that $\ulcorner T_{j+1} \urcorner \subseteq \ulcorner T_j \urcorner$. Termination can be decided as follows: for each region $\sigma \in T_{j+1}$ we check that there is a region $\tau \in T_j$ such that both $Empty(Diff(\sigma,\tau))$ and $Empty(Diff(\tau,\sigma))$. The symbolic semi-algorithm OR_1 closes P under the operations $CPre_1$, union, and intersection with observables (but not under $CPre_2$ and $CPre_{\{1,2\}}$). If we close P under $CPre_{\{1,2\}}$ and intersection with observables, then the result characterizes trace equivalence on the underlying transition structure [11]. Suppose that the input given to OR_1 is the region algebra of a game structure G. It can be seen by induction that for all $j \geq 0$, every region in T_j, as computed by OR_1, represents a $\cong_G^{1\text{-DG}\mu}$-block (i.e., a union of equivalence classes). Thus, if $\cong_G^{1\text{-DG}\mu}$ has finite index, then OR_1 terminates. Conversely, suppose that OR_1 terminates with $\ulcorner T_{j+1} \urcorner \subseteq \ulcorner T_j \urcorner$. It can be shown that if two states are not $\cong_G^{1\text{-DG}\mu}$-equivalent, then there is a region in T_j which contains one state but not the other. This implies that $\cong_G^{1\text{-DG}\mu}$ has finite index.

Theorem 2. *The symbolic semi-algorithm* OR_1 *terminates on the region algebra of a game structure* G *iff the* 1*-trace equivalence of* G *has finite index.*

All regions generated by the symbolic semi-algorithm ModelCheck for input formulas from 1-DGμ are also generated by the observation-refinement semi-algorithm OR_1. Therefore, if OR_1 terminates, so does ModelCheck on inputs from 1-DGμ.

Corollary 2. *The model-checking problems for 1-*DGμ *and 1-*Gω *are decidable on all game structures that have symbolic theories and 1-trace equivalences with finite index.*

The *rectangular games* [10] are a class of infinite-state game structures with symbolic theories and finite 1-trace equivalences. While in [10] rectangular hybrid games are solved by translation to timed games, which is impractical, the results of this section and Section 4 suggest a direct symbolic semi-algorithm for solving rectangular games, which is guaranteed to terminate. Such an algorithm has been implemented in the tool HyTech.

3.2 Intersection refinement

The symbolic semi-algorithm IR, called *intersection refinement*, on a region algebra starts from the finite set $T_0 := P$ of observables and generates inductively the finite sets of regions

$$T_{j+1} = T_j \cup \{CPre_1(\sigma), CPre_2(\sigma), CPre_{\{1,2\}}(\sigma) \mid \sigma \in T_j\} \\ \cup \{Or(\sigma,\tau) \mid \sigma,\tau \in T_j\} \cup \{And(\sigma,\tau) \mid \sigma,\tau \in T_j\}$$

for $j \geq 0$. The semi-algorithm IR terminates iff there is a j such that $\ulcorner T_{j+1} \urcorner \subseteq \ulcorner T_j \urcorner$. The symbolic semi-algorithm IR_1 closes P under the operations $CPre_1$, union, and intersection. If we close P under $CPre_{\{1,2\}}$, union, and intersection, then the result characterizes similarity on the underlying transition structure [11]. Suppose that the input given to IR_1 is the region algebra of a game structure G. For $j \geq 0$ and a state s of G, define $Sim_j(s) = \bigcap\{\ulcorner \sigma \urcorner \mid \sigma \in T_j \text{ and } s \in \ulcorner \sigma \urcorner\}$, where the set T_j of regions is computed by IR_1. By induction it is easy to check that for all $j \geq 0$, if t 1-simulates s, then $t \in Sim_j(s)$. Thus, every region in T_j represents a block of the 1-similarity for G. Conversely, suppose that IR_1 terminates with $\ulcorner T_{j+1} \urcorner \subseteq \ulcorner T_j \urcorner$. From the definition of 1-simulations, it follows that if $t \in Sim_j(s)$, then t 1-simulates s.

Theorem 3. *The symbolic semi-algorithm* IR *(respectively,* IR_1*) terminates on the region algebra of a game structure G iff the $\{1,2\}$-similarity (respectively, 1-similarity) of G has finite index.*

Corollary 3. *The model-checking problem for* NGμ *(respectively, 1-*NGμ*) is decidable on all game structures that have symbolic theories and $\{1,2\}$-similarity (respectively, 1-similarity) equivalences with finite index.*

An example of infinite-state game structures with symbolic theories and finite $\{1,2\}$-similarity equivalences are the *2-dimensional rectangular games* [10].

3.3 Partition refinement

The symbolic semi-algorithm PR, called *partition refinement*, on a region algebra starts from the finite set $T_0 := P$ of observables and generates inductively the

finite sets of regions

$$\begin{aligned} T_{j+1} = T_j \cup \{&CPre_1(\sigma), CPre_2(\sigma), CPre_{\{1,2\}}(\sigma) \mid \sigma \in T_j\} \\ \cup \{&Or(\sigma,\tau) \mid \sigma,\tau \in T_j\} \cup \{And(\sigma,\tau) \mid \sigma,\tau \in T_j\} \\ \cup \{&Diff(\sigma,\tau) \mid \sigma,\tau \in T_j\} \end{aligned}$$

for $j \geq 0$. The semi-algorithm **PR** terminates iff there is a j such that $\ulcorner T_{j+1} \urcorner \subseteq \ulcorner T_j \urcorner$. The symbolic semi-algorithm PR_1 closes P under $CPre_1$ and the boolean operations union, intersection, and set difference. If we close P under $CPre_{\{1,2\}}$ and all boolean operations, then the result characterizes bisimilarity on the underlying transition structure [5, 13]. The following is shown similar to the analysis of intersection refinement.

Theorem 4. *The symbolic semi-algorithm* **PR** *(respectively,* PR_1*) terminates on the region algebra of a game structure G iff the $\{1,2\}$-bisimilarity (respectively, 1-bisimilarity) of G has finite index.*

Corollary 4. *The model-checking problem for* Gμ *(respectively, 1-*Gμ*) is decidable on all game structures that have symbolic theories and $\{1,2\}$-bisimilarity (respectively, 1-bisimilarity) equivalences with finite index.*

An example of infinite-state game structures with symbolic theories and finite $\{1,2\}$-bisimilarity equivalences are the *timed games* [15].

4 Symbolic Controller Synthesis

We present symbolic semi-algorithms for solving the ω-regular control and control synthesis problems, and we provide conditions for the termination of these semi-algorithms. Consider a game structure G and an ω-regular expression K whose constants are observables of G. Player 1 can *control the state s of G w.r.t. K* if there exists a strategy f_1 of player 1 such that for every strategy f_2 of player 2, we have $L_{f_1,f_2}(s) \subseteq K$. In this case, we say that the strategy f_1 is a *control strategy for K from s*. The *ω-regular control problem* asks, given G and K, which states of G can be controlled w.r.t. K. The *ω-regular control synthesis problem* asks, in addition, for the construction of the control strategy.

Following [7], we use deterministic *Rabin-chain automata* (also called *parity automata*) for encoding the ω-regular property K. Deterministic Rabin-chain automata can encode all ω-regular properties [17], and they lead to compact Gμ formulas for solving the corresponding control problems.[5] A *Rabin-chain automaton of index n* is a tuple $\mathcal{C} = (Q, Q_0, \Delta, \Psi, \ell, \Omega)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\Delta : Q \to 2^Q$ is the transition relation, Ψ is the input alphabet, $\ell : Q \to \Psi$ is a state labeling, and $\Omega \colon Q \to \{0, \ldots, n-1\}$ is

[5] The solution of the ω-regular control problem on game structures requires deterministic ω-automata (see, e.g., [19]), whereas nondeterministic (and hence Büchi) ω-automata suffice for the ω-regular verification problem on the underlying transition structures, as in [11].

the acceptance condition. An *execution* of $\mathcal{C}$ on the infinite word $w_0 w_1 w_2 \ldots \in \Psi^\omega$ is an infinite sequence $e = q_0 q_1 q_2 \ldots$ of states such that $q_0 \in Q_0$ and for all $j \geq 0$, both $\ell(q_j) = w_j$ and $q_{j+1} \in \Delta(q_j)$. Let $\inf(e)$ denote the set of states that occur infinitely often along e. The execution e is *accepting* if the maximum index in the set $\{\Omega(q) \mid q \in \inf(e)\}$ is even. The automaton *accepts* the input word w if it has an accepting execution on w. The *language* of $\mathcal{C}$ is the set $L(\mathcal{C}) = \{w \in \Psi^\omega \mid \mathcal{C} \text{ accepts } w\}$. The automaton $\mathcal{C}$ is *deterministic and total* if (1a) for all states $q', q'' \in Q_0$, if $q' \neq q''$, then $\ell(q') \neq \ell(q'')$; (1b) for all input letters $\psi \in \Psi$, there is a state $q' \in Q_0$ such that $\ell(q') = \psi$; (2a) for all states $q \in Q$ and $q', q'' \in \Delta(q)$, if $q' \neq q''$, then $\ell(q') \neq \ell(q'')$; (2b) for all states $q \in Q$ and input letters $\psi \in \Psi$, there is a state $q' \in \Delta(q)$ such that $\ell(q') = \psi$. If $\mathcal{C}$ is deterministic and total, then we write $\Delta(q, \psi)$ for the unique state $q' \in \Delta(q)$ with $\ell(q') = \psi$.

Let $G = (S, A, \Gamma_1, \Gamma_2, \delta, P, \ulcorner \cdot \urcorner)$ be a game structure and $\mathcal{C} = (Q, Q_0, \Delta, \Psi, \ell, \Omega)$ a Rabin-chain automaton of index n such that $\Psi \subseteq 2^P$. To solve the ω-regular control problem for G and $\mathcal{C}$, we first construct a 1-DGμ formula χ' that computes the controllable states of the game structure $\mathcal{C} \times G$, obtained by taking the synchronous product between $\mathcal{C}$ and G. From χ', we construct a 1-DGμ formula χ that solves the ω-regular control problem directly on G. The product game structure $\mathcal{C} \times G = (S', A, \Gamma_1', \Gamma_2', \delta', (Q \times P) \cup \{c_0, \ldots, c_{n-1}\}, \ulcorner \cdot \urcorner')$ is defined as follows. For a state $s \in S$, let $P_s = \{p \in P \mid s \in \ulcorner p \urcorner\}$ be the set of observables at s. Define $S' = \{(q, s) \in Q \times S \mid \ell(q) = P_s\}$, with $\Gamma_i'(q, s) = \Gamma_i(s)$ for $i = 1, 2$, and $\delta'((q, s), a_1, a_2) = (\Delta(q, P_{\delta(s, a_1, a_2)}), \delta(s, a_1, a_2))$. Furthermore, $\ulcorner (q, p) \urcorner' = \{(q, s) \mid s \in \ulcorner p \urcorner\}$, and $\ulcorner c_i \urcorner' = \{(q, s) \mid \Omega(q) = i\}$ for all $0 \leq i < n$. Given a symbolic theory for G with the set R of regions, we define a symbolic theory for $\mathcal{C} \times G$ using as regions all functions of the form $R'\colon Q \to R$, with $\ulcorner R' \urcorner = \bigcup_{q \in Q} \{(q, s) \mid s \in \ulcorner R'(q) \urcorner\}$. From this representation, it is clear that the operations $\widetilde{CPre}_I$ for $I = 1, 2, \{1, 2\}$, *And*, *Or*, *Diff*, *Empty*, and *Member* are computable.

We give the formula χ' in equational form; it is straightforward to convert it to a formula of 1-DGμ by unrolling the equations and binding variables with μ or ν fixpoints. The formula χ' is composed of n blocks $B_0', \ldots, B_{n-1}'$; block B_0' is the innermost, and block B_{n-1}' the outermost. The block B_0' is a ν-block, and consists of the single equation $x_0 = \bigvee_{j=0}^{n-1} (c_j \wedge \langle\langle 1 \rangle\rangle \bigcirc x_j)$. For $1 \leq i < n$, the block B_i' is a μ-block if i is odd, a ν-block if i is even, and consists of the single equation $x_i = x_{i-1}$. The output variable is x_{n-1}. From the construction, it follows that player 1 can control a state s of G w.r.t. $\mathcal{C}$ iff $(q, s) \in [\![\chi']\!]_{\mathcal{C} \times G}$ for the unique $q \in Q_0$ such that $(q, s) \in S'$. The formula χ mimics on G the evaluation of χ' on $\mathcal{C} \times G$. It contains for each variable x_i of χ', for $0 \leq i < n$, the set $\{x_i^q \mid q \in Q\}$ of variables: the value of x_i^q at s keeps track of the value of x_i at (q, s). The formula χ is composed of n blocks $B_0, \ldots, B_{n-1}$. For $0 \leq i < n$, the block B_i consists of the set $\{e_i^q \mid q \in Q\}$ of equations. The equation e_i^q is derived by replacing in the equation of block B_i' on the l.h.s. the variable x_i with x_i^q, and by replacing on the r.h.s. c_j with *true* if $\Omega(q) = j$ and *false* otherwise, and by replacing $\langle\langle 1 \rangle\rangle \bigcirc x_j$ with $\langle\langle 1 \rangle\rangle \bigcirc \bigvee_{r \in \Delta(q)} x_j^r$; the r.h.s. is then conjuncted with the

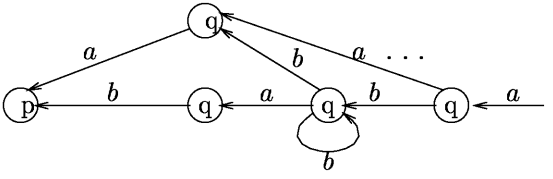


Fig. 3. Game structure on which OR_1 terminates, but CR_1 does not. (Player 2 has only one move enabled at each state.)

formula $(\bigwedge_{p\in\ell(q)} p) \wedge (\bigwedge_{p\in P\setminus\ell(q)} \neg p)$, which characterizes the observables of q. The block B_{n-1} contains the additional equation $x_{out} = \bigvee_{q\in Q_0} x^q_{n-1}$, which defines the output variable x_{out}. Then, player 1 can control a state s of G w.r.t. $\mathcal{C}$ iff $s \in [\![\chi]\!]_G$.

Lemma. *Each 1-*Gω *formula can be translated into an equivalent 1-*DGμ *formula.*

To solve the ω-regular control synthesis problem, assume that the semi-algorithm OR_1 terminates, let U be the resulting finite set of regions that define 1-trace equivalence classes for $\mathcal{C} \times G$, and let U' be the regions that define unions of regions from U.While computing χ', we can determine a *region strategy* $\hat{f}\colon U \to U'$ for the product structure $\mathcal{C} \times G$ following the algorithm for finite games [7, 20]. When the game is in $\ulcorner\sigma\urcorner$, for a region $\sigma \in U$, player 1 must choose an action that forces the game into $\ulcorner\hat{f}(\sigma)\urcorner$. Note that $\ulcorner\sigma\urcorner \subseteq \ulcorner CPre_1(\hat{f}(\sigma))\urcorner$ for all $\sigma \in U$ (if σ cannot be controlled, set $\hat{f}(\sigma) = \mathit{True}$). From the region strategy $\hat{f}$, we can obtain a memoryless strategy $f\colon Q \times S \to 2^A$ for $\mathcal{C} \times G$ by recovering which actions player 1 can choose at each state to force the game from $\ulcorner\sigma\urcorner$ to $\ulcorner\hat{f}(\sigma)\urcorner$. To this end, we define the function $Pre_1\colon R^Q \times A \to R^Q$ such that

$$\ulcorner Pre_1^a(\sigma)\urcorner = \{(q,s) \in S' \mid q \in Q \wedge a \in \Gamma_1(s) \wedge \forall b \in \Gamma_2(s).\, \delta'((q,s),a,b) \in \ulcorner\sigma\urcorner\}$$

for all $a \in A$ and $\sigma \in R^Q$.[6] For $\sigma \in U$ and $a \in A$, let $\sigma_a = Pre_1^a(\hat{f}(\sigma))$. Then $\ulcorner\sigma\urcorner \subseteq \ulcorner\bigcup_{a\in A} \sigma_a\urcorner$, because there is always at least one controlling action. If the game is at state $(q,s) \in \ulcorner\sigma\urcorner$ for a region $\sigma \in U$, define $f(q,s) = \{a \in A \mid (q,s) \in \ulcorner\sigma_a\urcorner\}$. Unlike a control strategy for $\mathcal{C} \times G$, a control strategy for G may need memory [6, 16]. We can construct such a strategy f' as follows. As the game goes on, the strategy f' feeds the observables of the visited states to a copy of the Rabin chain automaton $\mathcal{C}$, remembering the current state of the automaton. Upon reaching a state s, player 1 chooses an action in $f(q,s)$, where q is the current state of the automaton.

Theorem 5. *The ω-regular control synthesis problem can be solved on all game structures that have symbolic theories and 1-trace equivalences with finite index.*

Using the above construction, control strategies can be obtained symbolically. The construction uses the function Pre_1 to split the regions computed by the

[6] Note that the function Pre_1 can be computed from Pre using boolean operations.

semi-algorithm OR_1. However, we do not use Pre_1 to refine the region algebra into an algebra that is closed with respect to Pre_1. In fact, even if the semi-algorithm OR_1 terminates, a refinement based on Pre_1 may not. More precisely, let CR_1 be the semi-algorithm obtained from OR_1 by replacing $CPre_1$ with Pre_1. As the example of Figure 3 demonstrates, there are game structures on which OR_1 terminates, but CR_1 does not. The construction given above uses Pre_1 only once to refine the regions returned by OR_1, thus avoiding the problem.

References

1. P. Abdulla and B. Jonsson. Verifying networks of timed automata. In *TACAS 98*, LNCS 1384, pp. 298–312. Springer-Verlag, 1998.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *FOCS 97*, pp. 100–109. IEEE Computer Society Press, 1997.
4. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR 97*, LNCS 1466, pp. 163–178. Springer-Verlag, 1998.
5. A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In *CAV 90*, LNCS 531, pp. 197–203. Springer-Verlag, 1990.
6. J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the AMS*, 138:295–311, 1969.
7. E. Emerson and C. Jutla. Tree automata, mu-calculus, and determinacy. In *FOCS 91*, pp. 368–377. IEEE Computer Society Press, 1991.
8. E. Emerson, C. Jutla, and A. Sistla. On model checking for fragments of μ-calculus. In *CAV 93*, LNCS 697, pp. 385–396. Springer-Verlag, 1993.
9. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
10. T. Henzinger, B. Horowitz, and R. Majumdar. Rectangular hybrid games. In *CONCUR 99*, LNCS 1664, pp. 320–335. Springer-Verlag, 1999.
11. T. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *STACS 2000*, LNCS 1770, pp. 13–35. Springer-Verlag, 2000.
12. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
13. P. Kanellakis and S. Smolka. CCS expressions, finite-state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.
14. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
15. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *STACS 95*, LNCS 900, pp. 229–242. Springer-Verlag, 1995.
16. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149–184, 1993.
17. A. Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Symp. Comp. Theory*, LNCS 208, pp. 157–168. Springer-Verlag, 1984.
18. P. Ramadge and W. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25:206–230, 1987.
19. W. Thomas. Automata on infinite objects. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volume B, pp. 133–191. Elsevier, 1990.
20. W. Thomas. On the synthesis of strategies in infinite games. In *STACS 95*, LNCS 900, pp. 1–13. Springer-Verlag, 1995.

A Game-Based Verification of Non-repudiation and Fair Exchange Protocols

Steve Kremer and Jean-François Raskin*

Département d'Informatique - Faculté des Sciences
Université Libre de Bruxelles, Belgium
{skremer, Jean-Francois.Raskin}@ulb.ac.be

Abstract. In this paper, we report on a recent work for the verification of non-repudiation protocols. We propose a verification method based on the idea that non-repudiation protocols are best modeled as games. To formalize this idea, we use alternating transition systems, a game based model, to model protocols and alternating temporal logic, a game based logic, to express requirements that the protocols must ensure. This method is automated by using the model-checker MOCHA, a model-checker that supports the alternating transition systems and the alternating temporal logic. Several optimistic protocols are analyzed using MOCHA.

1 Introduction

Non-repudiation protocols. During the last decade, open networks, above all the Internet, have seen an impressive growth. As a consequence, new security issues, like non-repudiation have to be considered. Repudiation is defined as the *denial* of an entity of having participated in all or part of a communication. Consider for instance the following scenario: Alice wants to send a message to Bob; after having sent the message, Alice may deny having sent it (repudiation of origin), or Bob may deny having received it (repudiation of receipt). Therefore specific protocols have been designed, generating both a non-repudiation of origin (NRO) evidence, destined to Bob, and a non-repudiation of receipt (NRR) evidence, intended to Alice. These evidences are based on digital signatures that provide proofs of the origin of a message or a receipt. In case of a dispute Alice or Bob can present their evidences to an adjudicator, who can take a decision in favor of one of the two entities without ambiguity. The major problem in these protocols is to handle the fact that at one moment an entity will come into an advantageous position. For instance, if Alice starts sending her message, Bob has received all expected information and may stop the protocol being in an advantageous position. Different solutions have been proposed: they are generally divided into two classes, according to whether they use a trusted third party (TTP) or not. The approach without TTP is either based on a gradual

* This author was partially supported by a "Crédit aux chercheurs" granted by the Belgian National Fund for Scientific Research.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 551–565, 2001.
© Springer-Verlag Berlin Heidelberg 2001

release of knowledge or on a probabilistic protocol. It generally requires that all involved parties have equivalent computational power. In most cases these protocols are inefficient due to the large number of messages that need to be sent. The second approach is the one using a TTP. Thus, the message is first sent to the TTP, who acts as an intermediary to assure delivery. The major problem of this approach is the network and communication bottleneck, created at the TTP. To avoid the performance decrease created by this bottleneck, Asokan et al. introduced the optimistic approach for fair exchanges [2]. In an optimistic protocol one supposes that in general the involved entities are honest and the network is well functioning. The rationale is that the TTP only intervenes in case of a problem (a cheating entity or a network failing a delivery at a critical moment). Afterwards Zhou et al. applied the optimistic approach to the non-repudiation protocols [16]. Optimistic protocols are the ones that received most of the attention in recent literature.

A non-repudiation protocol has to respect several properties. The first one is *fairness*[1]: fairness must ensure that if at least one entity is honest, either both entities receive the expected non-repudiation evidence or none of them receives it. This property can also be split in the following two properties. *Fairness for Alice* ensures that, if Alice is honest, Bob receives the non-repudiation of origin evidence only if Alice receives her non-repudiation of receipt evidence. *Fairness for Bob* ensures that, if Bob is honest, Alice receives her non-repudiation of receipt evidence only if Bob receives his non-repudiation of origin evidence. The fairness requirement is the conjunction of these two properties. Another property we require is *timeliness*: we want that the protocol finishes for each honest player after a finite amount of time. A third property that is desirable but not necessary is viability. A protocol is *viable* if two honest players always succeed in exchanging the expected evidences. In general viability can only be realized by strengthening the requirements we make on the communication channels. We define three classes of channels: unreliable channels, resilient channels and operational channels. No assumptions have to be made about *unreliable channels*: data may be lost. A *resilient channel* delivers data after a finite, but unknown amount of time. Data may be delayed, but will eventually arrive. When using an *operational channel* data arrive before a known, constant amount of time. Operational channels are however rather unrealistic in heterogeneous networks. A formal definition of both these properties and the channels, will be found later.

In comparison to other security issues, such as privacy or authenticity of communications, non-repudiation has not been studied intensively. However, the more studied authentication protocols have shown that the design of security protocols is an error prone process. Some flaws have only been found after years, e.g. the Needham-Schroeder public-key authentication protocol. Hence, a need for formal verification methods has been identified. These methods include an adequate specification, more precise than the traditional, informal and sometimes

[1] The term *fairness* will be used in two different contexts in this paper that must not be confused: on one hand it is used to denote the fairness property that a protocol must respect, on the other hand we use it when discussing fair computations in the context of temporal logic.

ambiguous one often used in literature. Another aim is the automatic verification. Several works have been engaged: believe logics, such as BAN [5], general purpose model checkers, e.g. FDR [9], theorem provers as Isabelle for instance [12] and also special purpose verification tools as the NRL protocol analyzer [11] have successfully been applied to authentication protocols. To the best of our knowledge only four attempts have been made to verify non-repudiation and fair exchange protocols. First works have been done on non-repudiation protocols using CSP [13], where the proofs were generated by hand, and Zhou and Gollman briefly considered using the belief logic SVO [15]. Some work on fair exchange protocols has been realized using the model-checker Murφ [14] as well as the animation tool Possum [4].

Non-repudiation protocols as games. There are some fundamental differences between authentication protocols and exchange protocols, e.g. non-repudiation protocols. Generally one of the most difficult problems in authentication protocols is to deal with the presence of an intruder. In non-repudiation protocols we do not need to model an intruder, but we have to consider that either Alice or Bob, the two entities taking part in the protocol, may *cheat* (cf also [13] and [4]). The most important difference is that exchange protocols, above all optimistic ones, are not linear. Generally authentication protocols are ping pong protocols and only allow very few different traces. On the other hand, non-repudiation and fair exchange protocols are divided in several subprotocols (e.g. a main and a recovery protocol), making branching possible, although they are intended to be executed in a given order by a honest entity. Changing the order of execution could result in subtle errors. This is the reason why we propose a new method for the specification and the verification of exchange protocols. First, we want to model the actions that are possible in the course of the protocol and not stick to a given predefined order of execution. In that way, we give a malicious entity the potential not to follow the protocol, but to construct an attack against the honest entity. Second, we consider the execution of the protocol as a game: each entity (Alice, Bob, TTP) and each communication channel are players. We can think of designing a protocol as finding a *strategy*: the strategy proposed by the protocol has to defend a honest entity against all possible strategies of malicious parties that are trying to cheat. This point of view also allows us to express formally the required properties as strategies. For instance, a property such as fairness for Alice can be expressed as follows: "a coalition of Bob and all the communication channels does not have a strategy to obtain a non-repudiation of origin evidence without Alice having a strategy to obtain a non-repudiation of receipt evidence". Here, we have rephrased the property as the existence of a strategy. The main advantage of modeling such protocols as games is that we *directly* and *formally* take into account the possibility of *adversarial* behaviors. As the communication channels are also modeled as players, they can cooperate with protocol entities. This means that either the channels are not well-functioning or that they are controlled by a player of the coalition. For instance, when a message is lost on an unreliable channel, this can be due to a network failure, or a dishonest player who removed the message by cooperating with the communication channels. As a second example, consider the game view of the viability

property: Alice, in cooperation with Bob must have a strategy against the coalition of the communication channels, such that both Alice and Bob possess their expected evidences at the end of the protocol. This example illustrates the facility using the strategies not only to express adversarial behaviors, but also *cooperative* behaviors between several players. The trust in the TTP is modeled by giving the TTP a unique strategy: even if he makes part of one coalition or the other, his behavior is deterministic and thus *cannot choose* to help anyone. By using alternating transition systems and alternating-time temporal logic of Alur et al. [1], we are able to formalize the non-repudiation protocols and their requirements in a direct way.

Structure of the paper. We organize the paper as follows. In section 2, we introduce the *alternating transition systems* and the *alternating-time temporal logic* of Alur et al. [1]. Section 3 shows how exchange protocols, and more specifically non-repudiation protocols, can be modeled *naturally* and *accurately* as games. In section 4, we report on results about the automatic verification of several non-repudiation protocols using the model-checker MOCHA. In section 5, we compare our techniques to some related works. Finally, we report on plans for future works in section 6 and draw some conclusions.

2 A Formal Model of Games and Its Logic

Alternating transition systems. The formalism that we use to model exchange protocols as games is the model of alternating transition systems, ATS for short. The formal definition of this model is given in [1]. Here we only give an intuitive introduction. ATS are a game variant of usual Kripke structures. An ATS is composed of a set of states Q that represents the possible game configurations, a finite set of propositions P, a labeling function $L : Q \to 2^P$ that labels states with propositions, a set of players Σ and a game transition function δ. The game transition function defines for every player a and state q the set of choices $\delta(q, a) = \{Q_1, Q_2, \ldots, Q_n\}$, with $Q_i \subseteq Q$, that the player a can make in q. A choice is a set of possible next states. One step of the game at a state q is played as follows : each player $a \in \Sigma$ makes his choice Q_a and the next state q' of the game is the intersection (that is required to be a singleton) of the choices made by all the players of Σ, i.e. $\{q'\} = \bigcap_{a \in \Sigma} Q_a$. A computation is an infinite sequence $\lambda = q_0 q_1 \ldots q_n \ldots$ of states, such that for every $i \geq 0$, there exists $Q_{a_1}, \ldots, Q_{a_n}$ with $\Sigma = \{a_1, \ldots, a_n\}$ and such that for every j, $1 \leq j \leq n$, $Q_{a_j} \in \delta(q_i, a_j)$ and $q_{i+1} = \bigcap_{1 \leq j \leq n} Q_{a_j}$.

Alternating-time temporal logic. We now introduce the alternating-time temporal logic [1], ATL for short. For a set of players $A \subseteq \Sigma$, a set of computations Λ, and a state q, consider the following game between a protagonist and an antagonist. The game starts at state q. At each step, to determine the next state, the protagonist chooses among the choices controlled by the players in the set A, while the antagonist chooses among the remaining choices. If the resulting

infinite computation belongs to the set Λ, then the protagonist wins. If the protagonist has a winning strategy, we say that the ATL formula $\langle\langle A\rangle\rangle\Lambda$ is satisfied in state q. Here, $\langle\langle A\rangle\rangle$ is a path quantifier, parameterized by the set A of players, which ranges over all computations that the players in A can force the game into, irrespective of how the players in $\Sigma \setminus A$ proceed. The set Λ is defined using temporal logic formulas. If the reader is familiar with the branching time temporal logics, he may see that the parameterized path quantifier $\langle\langle A\rangle\rangle$ is a generalization of the path quantifiers of CTL: the existential path quantifier $\exists$ corresponds to $\langle\langle\Sigma\rangle\rangle$, and the universal path quantifier $\forall$ corresponds to $\langle\langle\emptyset\rangle\rangle$. We now illustrate the expressive power of ATL. Consider the set of players $\Sigma = \{a, b, c\}$ and the following formulas with their verbal reading:

- $\langle\langle a\rangle\rangle\Diamond p$, player a has a strategy against players b and c to eventually reach a state where the proposition p is true;
- $\neg\langle\langle b, c\rangle\rangle\Box p$, the coalition of players b and c does not have a strategy against a to reach a point where the proposition p will be true for ever;
- $\langle\langle a, b\rangle\rangle\bigcirc (p \wedge \neg\langle\langle c\rangle\rangle\Box p)$, a and b can cooperate so that the next state satisfies p and from there c does not have a strategy to impose p for ever.

Those three formulas are a good illustration of the great expressive power of ATL to express cooperative as well as adversarial behaviors between players.

Fairness. As in the usual temporal logic setting, fairness can be used to rule out computations. For example, to evaluate a formula of the form $\langle\langle a\rangle\rangle\Diamond\phi$, we only consider computations where the antagonists, that is the agents in $\Sigma \setminus \{a\}$, respect their fairness constraints. A *fairness constraint* is a function $\gamma : Q \times \Sigma \to 2^{2^Q}$ such that for each $q \in Q$ and $a \in \Sigma$, we have $\gamma(q, a) \subseteq \delta(q, a)$. Given a computation $\lambda = q_0 q_1 \dots q_n \dots$, we say that γ is *a-enabled* at position $i \geq 0$ if $\gamma(q_i, a) \neq \emptyset$. We say that γ is *a-taken* at position $i \geq 0$ if there exists a set $Q' \in \gamma(q_i, a)$ such that $q_{i+1} \in Q'$. Finally given a set $A \subseteq \Sigma$, we say that λ is *weakly* $\langle\gamma, A\rangle$*-fair* [2] if for each agent $a \in A$, either there are infinitely many positions of λ at which γ is not a-enabled, or there are infinitely many positions of λ at which γ is a-taken. To check ATL formulas under weak fairness constraints, we use the script language of MOCHA and a symbolic adaptation of the techniques described in [1].

Game guarded command language. Instead of modeling protocols directly with ATS we will use a more user-oriented notation: a guarded command language "à la Dijkstra". The details about the syntax and semantics of this language (given in terms of ATS) can be found in [6]. Here follows an intuitive presentation. Each player $a \in \Sigma$ disposes of a set of guarded commands of the form

$$guard_\xi(X) \to update_\xi(X, Y') \tag{1}$$

[2] Note that there exists also, as in the usual temporal logic setting a notion of strong fairness, the interested reader is referred to [1] for the definition. We will not need the notion of strong fairness in this paper.

where X denotes the set of all variables and Y' the set of variables, controlled by a in the next state. A computation-step is defined as follows: each player $a \in \Sigma$ chooses one of his commands whose guard evaluates to true, and the next state is obtained by taking the conjunction of the effects of each update part of the commands selected by the players.

3 Formal Modelization of Non-repudiation Protocols

Modeling of the protocols. Non-repudiation and fair exchange protocols have several particularities that allow us to make some simplifications. As also noted in [4] all messages are generally protected by digital signatures or an equivalent mechanism. Hence we consider that only well formed messages can be sent. Moreover each protocol execution can be uniquely identified by the identity of the participating entities and a special label. When the label contains some well chosen information related to an execution, as proposed for instance in [16], it is possible to show that different parallel executions cannot interfere. Therefore we do not need to verify the execution of several parallel runs. The notation, classically used in literature to describe cryptographic protocols ($A \rightarrow B : m$, to denote that Alice sends a message m to Bob), has several drawbacks. The protocols are presented as a linear sequence of message exchanges, with a predefined order. In the case of optimistic exchange protocols often subprotocols can be invoked at different moments. But, running a subprotocol at a time not foreseen by the designer, may have unexpected side-effects, threatening the security of a protocol if one of the participating entities tries to cheat the other entities. We use the modeling language described above to model the exchange protocols. To execute a given action ξ, $guard_\xi$ must evaluate to true. The guard $guard_\xi$ is used to represent the elements (such as keys, messages, ...) necessary for a participating entity, that is a player in our modelization, to execute the action ξ. The variables controlled by a player represent his current knowledge and thus determine which actions he/she can execute. This specification takes into account all possible executions of subprotocols corresponding to a given initial knowledge, as we do not give any predefined order to these guarded commands. At each point in the protocol execution all messages that could possibly be sent are determined. Thus, the specification enables a malicious entity to choose a different order of execution and construct a possible attack. We model the two players Alice and Bob with this approach.

The TTP is a special player and has to be modeled in a particular way. The TTP must be *impartial*, it may not help one or the other player. To make sure that the TTP does not have a strategy to help one of the players to cheat, we model the TTP such that it is deterministic: at each stage of the execution of the protocol, the TTP executes the action requested by the protocol.

The communication channels are also modeled as players. Each transmission is modeled as a guarded command. We consider three different kinds of communication channels: unreliable, resilient, and operational. To specify *unreliable channels* we add an action that the channel can always take and has no effect, i.e. this is simply an idle action. As a consequence, a message sent on an unreliable

channel may never be delivered. A *resilient channel* can be specified in a similar way, but we have to add fairness conditions on the computations in order to force the transmission before a finite amount of time. More precisely, assume that a resilient channel c has to deliver a message m_1 when the proposition $\texttt{SendM}_1$ is set to true and the delivery of the message is modeled by setting $\texttt{M}_1$ to true. We impose the following (weak) fairness constraint on c: for each $q \in Q$ such that $q \models \texttt{SendM}_1$, we have

$$\gamma(c, q) = \{Q' \mid Q' \in \delta(c, q) \text{ and } \forall q' \in Q' : q' \models \texttt{M}_1\}.$$

This fairness constraint rules out trajectories where c can deliver a message and never delivers it. *Operational channels* do not have any additional action and thus immediately transmit the messages that have been sent on it. By immediately transmitting messages we alter the channel definition, which requires messages to be transmitted before a known constant time. This is however acceptable as the sender can always wait before sending the message to obtain the desired delay.

When starting to model an informally described protocol we apply the following method. First, we determine the initial knowledge of each player. Initial knowledge may for instance include nonces, public and shared keys as well as initially known messages. All these items are represented by boolean variables initialized to true. All other variables are set to false, indicating that there value is not known yet. Then we model all useful cryptographic operations that can be applied on the initial knowledge. By useful we mean that the operation generates an item that can be used later in the protocol. Now the following steps are applied to each message $\texttt{X} \rightarrow \texttt{Y} : \texttt{M}_\texttt{i} = \texttt{m}_{\texttt{i1}}, \texttt{m}_{\texttt{i2}}, \ldots, \texttt{m}_{\texttt{in}}$ of the protocol. To represent the sending of $\texttt{M}_\texttt{i}$, a guarded command $\texttt{m}_{\texttt{i1}} \wedge \cdots \wedge \texttt{m}_{\texttt{in}} \rightarrow \texttt{SENDm}_\texttt{i} := \texttt{true}$ is added to player X's description. The guard contains the conjunction of all the elements of the message. This means that a player is able to send a message if and only if he has knowledge of all the required elements. The update relation expresses the intention to send the message. The transmission of the message is represented by a guarded command added to the description of the communication channel between X and Y. The command will be of the form $\texttt{SENDm}_\texttt{i} \rightarrow \texttt{M}_\texttt{i} := \texttt{true}$. Note that in order to delay this message a communication channel may choose to execute a different command. Reception of the message is modeled by adding the following command to Y's description $\texttt{M}_\texttt{i} \rightarrow \texttt{m}_{\texttt{i1}} := \texttt{true}; \ldots; \texttt{m}_{\texttt{in}} := \texttt{true}$. For each element of $\texttt{M}_\texttt{i}$ being set to true for the first time in Y's description, we also add all useful cryptographic operations that can be applied on it.

The above presented method describes how to model Alice, respectively Bob having arbitrary behavior. To restrict their behavior to the honest protocol execution, we can easily reinforce the guards so that the order of execution corresponds to the one dictated by the protocol. We get two different descriptions of Alice, as well as of Bob: the first description models arbitrary behaviors, the second one only the honest behavior.

Modeling of the requirements. We show here how the main requirements that an exchange protocol must fulfill, can be *naturally* rephrased as the existence of

strategies for the participating entities to reach their goal. The logic ATL is used to formalize those requirements. We will concentrate here on properties of non-repudiation protocols. The properties described are general properties that do not apply to a given protocol. They may need to be instantiated when studying a protocol.

In a first approximation, we introduced fairness as the property that either both entities receive all their desired evidences or none of them receives any valuable evidence. We can now formulate this requirement as the existence of strategies. We say that the protocol is fair for Alice if "Bob and the communication channels do not have a strategy to reach a state where Bob has his proof of origin and Alice has no more a strategy to obtain her proof of receipt". This can be formally expressed by the following ATL formula:

$$\neg\langle\langle B, \mathsf{Com}\rangle\rangle\Diamond(\mathsf{NRO} \wedge \neg\langle\langle A\rangle\rangle\Diamond\mathsf{NRR}) \tag{2}$$

Here, Com denotes the set of all communication channels. NRR and NRO respectively denote the non-repudiation of receipt and origin evidences. This is a generic notation as the form of those evidences depends on a particular protocol. The expressive power of the logic ATL is well illustrated by this example. Cooperation and adversarity are naturally expressed. To better understand the advantages of using a game logic to formalize this requirement, let us consider the following CTL formula:

$$\neg\exists\Diamond(\mathsf{NRO} \wedge \neg\exists\Diamond\mathsf{NRR}) \tag{3}$$

that may look as an appropriate CTL candidate for the formalization of fairness for Bob. Note that we have obtained this formula by replacing the two teams in (2) by the entire set of players Σ. Note also that the formula (3) can be rewritten in the equivalent and more readable positive form:

$$\forall\Box(\mathsf{NRO} \rightarrow \exists\Diamond\mathsf{NRR}) \tag{4}$$

That says "on every state of every run of the protocol, if Bob has his evidence of origin then there should exist a continuation of the protocol on which Alice eventually receives her evidence of receipt". This way, we have lost however the ability of distinguishing between cooperative and adversarial behaviors. Let us show what are some consequences of this, so to say, "loss of precision". For example, it may be the case that the formula (4) is verified because in the course of the protocol a *resilient* channel helps Alice to obtain her proof by delivering a message within a given bound (which is not generally the case for such a channel). As this assumption of cooperative behavior of the resilient channels cannot be made in general, in order to be fair the protocol should guarantee to Alice that she obtains her proof even if the resilient channel does not deliver the message within a certain bound. This fact is not ensured by the CTL formalization but is ensured by our ATL formalization. In fact, in formula (2), it is explicitly stated that the communication channels play against Alice. A resilient channel has only the obligation to deliver the message after a finite amount of time and not within a given bound, even if that helps Alice to obtain her proof. This

non-cooperative aspect is precisely formalized in ATL. Also formula (3) may be verified because there exists some paths, where Bob is honest and allows Alice to obtain her proof. Again, the protocol should not make the hypothesis that Bob is honest, and should ensure fairness for Alice even if Bob is trying to cheat her. This adversarial behavior is formally modeled by our ATL formula and can not be formalized in CTL. We can also have that the CTL formula is false, even if the ATL formula holds. Such a situation can occur if Alice's specification contains some non-determinism. For instance, if the specification does not contain at which moment Alice can stop the protocol, she may stop it at a "wrong" moment. The CTL formula will fail, while the ATL formula may hold, as not stopping the protocol at a given moment would have lead to a winning strategy. Using the notion of strategy, we "automatically" exclude behaviors of Alice that do not serve her objective. Note that this last comment also rules out the following CTL formula as a candidate for fairness:

$$\forall\Box(\mathsf{NRO} \rightarrow \forall\Diamond\mathsf{NRR}) \tag{5}$$

Situations where the protocol is not well defined, occur frequently when integrating the use of formal methods as an aid in the protocol designing process. In the first step while designing the protocol, we generally do not want to deal with all details: the aim of the verification is to determine whether the given specification contains a winning strategy that leads to a correct protocol. If Alice is specified in a deterministic way, one may use formula 5 to verify fairness for Alice. However, we believe that at a specification level, as well as during the design process, the fairness requirement should explicitly contain the adversarial as well as the cooperative behaviors.

Viability is expressed by the following formula:

$$\langle\langle A, B\rangle\rangle\Diamond(\mathsf{NRO} \wedge \mathsf{NRR}) \tag{6}$$

that says "Alice and Bob can cooperate, so they are honest (i.e. they are following the protocol), and in that case the protocol should allow them, even in presence of non-cooperating channels, to obtain their respective evidences". And finally timeliness is formalized by :

$$\langle\langle A\rangle\rangle\Diamond(\mathsf{stop}_A \wedge (\neg\mathsf{NRR} \rightarrow \neg\langle\langle B\rangle\rangle\Diamond\mathsf{NRO})) \tag{7}$$

which expresses that "Alice has a strategy to finish the protocol and if she does not have her evidence at that point, Bob will not be able to obtain his evidence neither". The same requirement can be expressed for Bob.

4 Verification with MOCHA

We used the model-checker MOCHA to verify several protocols. We report in detail on our verification of the Zhou-Gollmann optimistic protocol [16] and illustrate on this example our verification methodology. We also checked the Asokan-Shoup-Waidner certified mail protocol [3], the Kremer-Markowitch non-repudiation protocol [7], as well as the Markowitch-Kremer multi-party non-repudiation protocol [10]. For these protocols, due to lack of space, we only briefly show our results without giving a complete description of the protocols.

4.1 The ZG Optimistic Non-repudiation Protocol

We first give an informal description of the protocol using the following notation:

- $X \to Y$: transmission from entity X to entity Y
- $X \leftrightarrow Y$: ftp get operation performed by X at Y
- $h()$: a collision resistant one-way hash function
- $E_k()$: a symmetric-key encryption function under key k
- $D_k()$: a symmetric-key decryption function under key k
- $S_X()$: the signature function of entity X
- m: the message sent from A to B
- k: the message key A uses to cipher m
- $c = E_k(m)$: the cipher of m under the key k
- $l = h(m, k)$: a label that in conjunction with the entities (A,B) identifies a protocol run
- f: a flag indicating the purpose of a message
- t: the time-out chosen by A

The protocol generates the following evidences:

- $\mathsf{EOO} = S_A(f_{\mathsf{EOO}}, B, l, t, c)$: the evidence of origin of c
- $\mathsf{EOR} = S_B(f_{\mathsf{EOR}}, A, l, t, c)$: the evidence of receipt of c
- $\mathsf{EOO}_k = S_A(f_{\mathsf{EOO}_k}, B, l, t, k)$: the evidence of origin of k
- $\mathsf{EOR}_k = S_B(f_{\mathsf{EOR}_k}, A, l, t, k)$: the evidence of receipt of k
- $\mathsf{Sub}_k = S_A(f_{\mathsf{Sub}_k}, B, l, k)$: the evidence of submission of k
- $\mathsf{Con}_k = S_{\mathsf{TTP}}(f_{\mathsf{Con}_k}, A, B, l, t, k)$: the evidence of confirmation of k issued by the TTP

The protocol is divided into two subprotocols: a main protocol and a recovery protocol. As the protocol is based on the optimistic approach, the trusted third party (TTP) does only intervene in the recovery protocol. We start by describing the main protocol.

1. $A \to B$: $f_{\mathsf{EOO}}, B, l, t, c, \mathsf{EOO}$
2. $B \to A$: $f_{\mathsf{EOR}}, A, l, \mathsf{EOR}$
3. $A \to B$: $f_{\mathsf{EOO}_k}, B, l, k, \mathsf{EOO}_k$
4. $B \to A$: $f_{\mathsf{EOR}_k}, A, l, \mathsf{EOR}_k$

First Alice sends the digitally signed cipher $c = E_k(m)$ to Bob. In the second message Bob responds with the evidence of receipt for this cipher (EOR). If Alice does not receive the second transmission she stops the protocol, otherwise she sends the signed decryption key k to Bob. Bob answers by sending the receipt EOR_k for the key. The label l, present in each transmission, identifies the protocol run. The time-out t specified in message 1 is used in the recovery protocol. Alice may initiate the recovery protocol if Bob does not send the receipt for the key.

The steps of the recovery protocol are the following:

1. $A \to \mathsf{TTP}$: $f_{\mathsf{Sub}_k}, B, l, t, k, \mathsf{Sub}_k$
2. $B \leftrightarrow \mathsf{TTP}$: $f_{\mathsf{Con}_k}, A, B, l, t, k, \mathsf{Con}_k$
3. $A \leftrightarrow \mathsf{TTP}$: $f_{\mathsf{Con}_k}, A, B, l, t, k, \mathsf{Con}_k$

Alice sends the signed key, together with the deadline t to the TTP. If the key arrives after t, the TTP does not accept the recovery. Otherwise the TTP publishes the key together with a confirmation Con_k for the key in a read-only accessible directory, where both Alice and Bob can fetch the key as well as Con_k. Con_k serves to Bob as the evidence of origin of the key, and to Alice as the evidence of receipt of the key as it is accessible to Bob. The deadline t is necessary for Bob to know the moment when either the key is published or will not be published anymore. In this protocol, the non-repudiation of origin evidence NRO is composed of EOO and EOO_k or of EOO and Con_k. The non-repudiation of receipt evidence NRR is composed of EOR and EOR_k or of EOR and Con_k.

Zhou et al. suppose that the channels between Alice and Bob are unreliable and the channels between the TTP and both Alice and Bob are resilient. In our model we do not model the ftp get operation, but assume that the TTP takes the initiative to send messages 2 and 3 of the recovery protocol to Alice respectively Bob, as soon as these actions are possible.

To verify the protocol we instantiate the properties defined in section 3 for two versions of both Alice and Bob. The first version allows arbitrary behavior[3], the second one restricts the behavior to the honest protocol execution. The specification of Alice, respectively Bob, allowing arbitrary behavior will be denoted with A, respectively B, while the specification of the honest behavior of Alice and Bob will be denoted $\mathsf{A_h}$, respectively $\mathsf{B_h}$. When we want to check a property, for instance fairness for Alice, we first verify that Alice has a strategy to assure the fairness requirements, when she is allowed to behave in an arbitrary way. We check that

$$\neg\langle\langle \mathsf{B}, \mathsf{Com}\rangle\rangle\Diamond(\mathsf{EOO} \wedge (\mathsf{EOO_k} \vee \mathsf{Con_k}) \wedge \neg\langle\langle \mathsf{A}\rangle\rangle\Diamond(\mathsf{EOR} \wedge (\mathsf{EOR_k} \vee \mathsf{Con_k})))$$

holds. If the formula does not hold, even with arbitrary behavior of Alice, the protocol is flawed and can not be fixed without introducing new 'mechanisms' in the protocol. If the formula holds we additionally need to check that Alice cannot be cheated when following the protocol. It is possible that the formula holds because Alice adopted a strategy, that is not the one proposed by the protocol. Therefore we check the following formula:

$$\neg\langle\langle \mathsf{B}, \mathsf{Com}\rangle\rangle\Diamond(\mathsf{EOO} \wedge (\mathsf{EOO_k} \vee \mathsf{Con_k}) \wedge \neg\langle\langle \mathsf{A_h}\rangle\rangle\Diamond(\mathsf{EOR} \wedge (\mathsf{EOR_k} \vee \mathsf{Con_k})))$$

In this formula we restricted Alice's behavior to the actions that are dictated by the protocol. If the formula evaluates to false, it means that the protocol, as originally described, is flawed. However, it is possible to easily correct it, as a strategy was found by the model-checking algorithm while checking the previous formula. This means that we do not have to fundamentally change the protocol, by adding or changing messages.

While investigating the protocol, we succeeded in finding a flaw in the protocol, even if Alice's behavior is not restricted to the honest protocol execution.

[3] We call arbitrary behavior, all behavior that is *interesting* with respect to the protocol, i.e. behavior that results in sending or obtaining acceptable messages related to the protocol.

Remember that the communication channels between the TTP and both Alice and Bob are resilient and that a finite time-out must be chosen by Alice at the begin of the protocol. In our model, the time-out is a boolean variable controlled by a player *Clock*. We restrict the time-out to become true after a finite amount of time, using a similar technique as used for transmission on resilient channels. In fact, when the main protocol has been executed until step 3, Bob has received all of his evidences and may decide to stop the protocol in order to try to cheat Alice. At this point we check whether Alice in cooperation with the clock does have a strategy against Bob who is cooperating with the communication channels, to receive her non-repudiation of receipt evidences. As Alice cooperates with the clock, she entirely controls the triggering of the time-out. Intuitively, it means that Alice can choose the time-out to occur as late as she wants, which represents the fact, that she chooses the time-out at the first step of the protocol. As Alice does not receive her evidence, she has to initiate a recovery. For the recovery protocol to be successful, the request needs to arrive before the time-out. However, as communication channels can help Bob, this message can be delayed until the time-out occurs. Intuitively, a resilient channel has the capacity to delay messages: thus for each possible finite time-out value chosen by Alice, Bob (with the help of the communication channels) can delay long enough the delivery of the message in order for the TTP to reject the recovery request. However, when we change the communication channel to be operational (messages are transmitted immediately) the protocol becomes fair. Unfortunately, operational channels are rather unrealistic in practice.

We can also use our method to find the origin of a flaw. We can add players to the cooperation to detect the player(s) that make the property fail. As an example, take a look at the flaw we described in the previous paragraph. We asked whether Bob, together with all the communication channels, has a strategy to obtain the NRO evidence, such that Alice, in cooperation with the clock, does not have any strategy to receive the NRR evidence. Using MOCHA we showed the existence of a flaw, by validating this property. However, it may be difficult to determine the failure's origin. Thus we try to change the coalitions to find the player responsible for the flaw. By adding the communication channel between Alice and the TTP to Alice's coalition we succeeded in avoiding the property to be validated. Once we know that the communication channel is at the origin of the error it may be interesting to alter the channel's quality. In that way we see which modifications are needed to recover the fairness property. In this example we concluded that replacing the resilient channel by an operational channel was enough to prevent protocol failure. Hence, we have a rather easy and quick investigation method to locate and possibly prevent flaws.

4.2 Other Verified Protocols

We also verified the ASW certified e-mail protocol [3]. In this protocol we found errors due to the fact that sub-protocols can be executed out of order, in a way not foreseen by the authors. We verified a variant of the KM non-repudiation protocol [7], where the abort protocol has been removed. This results in a fair protocol, which however does not respect timeliness anymore and hence is not

secure. Finally we analyzed the MK multi-party non-repudiation protocol [10]. We verified an instance with two recipients and showed that problems can arise due to a race condition, if we do not correctly handle a clock synchronization problem, as proposed in the original paper. These results can be found in an extended version of this paper, available as a technical report [8].

5 Related Works

First efforts to apply formal methods to the verification of non-repudiation protocols have been presented by Zhou et al. in [15]. SVO, a "BAN-like" belief logic has been used to study a non-repudiation protocol. The aim of that study differs however from our study. In the context of non-repudiation protocols, belief logics are useful to reason about the validity of the evidences. They deal with questions of what a judge has to believe when Alice or Bob present their respective evidences. Belief logics cannot be applied to verify properties such as fairness or timeliness. In [13] Schneider uses CSP to prove the correctness of a non-repudiation protocol. These proofs are not automated and require great efforts. Recently Boyd and Kearny [4] discussed a method using specification animation to analyse fair exchange protocols using the Possum animation tool. The tool gives the possibility to step through the protocol and examine the consequences of various actions. Boyd et al. use a high level abstraction that only allows to find errors due to sending messages out of order. However they succeed in finding errors even on these very simplified versions of the protocols. The most extensive studies of fair exchange protocols using formal methods have been presented by Shmatikov and Mitchell in [14]. They use Murφ, a finite-state model-checker to analyze a fair exchange and two contract signing protocols. Their approach differs from ours on some major points. First, they use an intruder model. To model the fact that a party is malicious and could try to cheat, they make that party share all its knowledge with the intruder. The same intruder model has been used to analyse other security protocols. However we believe that there is no need to use an intruder model. Therefore we directly model Alice as well as Bob to be potentially malicious. The analysis using Murφ is based on invariant checking. Channel resilience is obtained by checking all invariants in final protocol states. Protocol runs where messages are lost do not reach the final protocol states and are discarded. To achieve resilience in our model we need to add fairness conditions. Invariant checking is sufficient to verify fairness as fairness is a *monotonic* property: if fairness is broken at one point of the protocol, the protocol will remain unfair. Checking only final states is thus sufficient to verify fairness. However with this approach timeliness cannot be verified. Timeliness is a liveness property guaranteeing that we can always reach a final state in the protocol. A protocol run, not respecting timeliness, would be discarded for the invariant check as the protocol would not be finished. Note that Shmatikov et al. did not aim to verify timeliness. In our model the timeliness property can easily be verified. As shown before timeliness is of crucial importance for the security of a protocol and should be verified.

6 Conclusions and Future Works

We have shown that exchange protocols, due to their particularities, are best modeled as games. First, the adversarial and cooperative behaviors that occur during the execution of those protocols are naturally and precisely expressed in term of strategies. Second, the main requirements that an exchange protocol must ensure are easily phrased as existence of strategies for the participating entities to reach well defined goals. We have proposed to use the framework of alternating transition systems and alternating-time temporal logic to formalize this view of exchange protocols as games. The great expressive power of alternating temporal logic in this context has been illustrated. The practical interest of the method has been demonstrated by analyzing several protocols. For the analysis, we have used the tool MOCHA which is able to automatically verify alternating temporal formula on alternating transition systems. In the future, we will try to show that this approach is applicable to the analysis and verification of more specific properties, such as abuse-freeness in contract signing protocols for instance. We will also investigate the problem of strategy synthesis. Strategy synthesis is a very powerful tool in protocol design and can also be used to synthesize attacks, showing the origin of flaws.

Acknowledgement. The authors would like to thank Giorgio Delzanno, Olivier Markowitch and Thierry Massart for carefully reading previous versions of this paper, as well as Freddy Mang for his assistance while using MOCHA and the anonymous referees for their helpful comments.

References

1. R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.
2. N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In T. Matsumoto, editor, *4th ACM Conference on Computer and Communications Security*, pages 6, 8–17, Zurich, Switzerland, Apr. 1997. ACM Press.
3. N. Asokan, V. Shoup, and M. Waidner. Asynchronous protocols for optimistic fair exchange. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 86–99, Oakland, CA, May 1998. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.
4. C. Boyd and P. Kearney. Exploring fair exchange protocols using specification animation. In *The Third International Workshop on Information Security - ISW2000*, Lecture Notes in Computer Science, Australia, Dec. 2000. Springer-Verlag.
5. M. Burrows, M. Abadi, and R. Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996*. 1996.
6. T. Henzinger, R. Manjumdar, F. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In *SAS 2000: Intertional Symposium on Static Analysis*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

7. S. Kremer and O. Markowitch. Optimistic non-repudiable information exchange. In J. Biemond, editor, *21th Symp. on Information Theory in the Benelux*, pages 139–146. Werkgemeenschap Informatie- en Communicatietheorie, Enschede, may 2000.
8. S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. Technical Report 451, ULB, 2001.
9. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Lecture Notes in Computer Science*, 1055:147–157, 1996.
10. O. Markowitch and S. Kremer. A multi-party optimistic non-repudiation protocol. In D. Won, editor, *Proceedings of The 3rd International Conference on Information Security and Cryptology (ICISC 2000)*, volume 2015 of *Lecture Notes in Computer Science*, Seoul, Korea, 2000. Springer-Verlag.
11. C. A. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. *Lecture Notes in Computer Science*, 1146:351–365, 1996.
12. L. C. Paulson. Proving properties of security protocols by induction. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, June 1997.
13. S. Schneider. Formal analysis of a non-repudiation protocol. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW '98)*, pages 54–65, Washington - Brussels - Tokyo, June 1998. IEEE.
14. V. Shmatikov and J. Mitchell. Analysis of abuse-free contract signing. In *Financial Cryptography '00*, Anguilla, 2000.
15. J. Zhou and D. Gollmann. Towards verification of non-repudiation protocols. In *Proceedings of 1998 International Refinement Workshop and Formal Methods Pacific*, pages 370–380, Canberra, Australia, Sept. 1998. Springer.
16. Y. Zhou and D. Gollmann. An efficient non-repudiation protocol. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

The Control of Synchronous Systems, Part II*

Luca de Alfaro, Thomas A. Henzinger, and Freddy Y.C. Mang

Electrical Engineering and Computer Sciences
University of California at Berkeley.
{dealfaro,tah,fmang}@eecs.berkeley.edu

Abstract. A controller is an environment for a system that achieves a particular control objective by providing inputs to the system without constraining the choices of the system. For synchronous systems, where system and controller make simultaneous and interdependent choices, the notion that a controller must not constrain the choices of the system can be formalized by type systems for composability. In a previous paper, we solved the control problem for static and dynamic types: a static type is a dependency relation between inputs and outputs, and composition is well-typed if it does not introduce cyclic dependencies; a dynamic type is a set of static types, one for each state. Static and dynamic types, however, cannot capture many important digital circuits, such as gated clocks, bidirectional buses, and random-access memory. We therefore introduce more general type systems, so-called *dependent* and *bidirectional* types, for modeling these situations, and we solve the corresponding control problems.
In a system with a dependent type, the dependencies between inputs and outputs are determined gradually through a game of the system against the controller. In a system with a bidirectional type, also the distinction between inputs and outputs is resolved dynamically by such a game. The game proceeds in several rounds. In each round the system and the controller choose to update some variables dependent on variables that have already been updated. The solution of the control problem for dependent and bidirectional types is based on algorithms for solving these games.

1 Introduction

The *control problem* asks, given an open system P and a property ϕ, to construct a controller Q such that the controlled system $P||Q$ satisfies ϕ (or to answer "uncontrollable," if no such Q exists). The control problem has applications in the synthesis of reactive programs and sequential circuits [PR89], in discrete-event control [RW87], in modular verification [AdAHM99], and in early error detection [dAHM00b]. An important special case is the *single-step* control problem, for properties of the form $\phi = \bigcirc f(X)$, where $\bigcirc$ is the temporal "next" operator,

* This research was supported in part by the SRC contract 99-TJ-683.003, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, and the NSF Theory grant CCR-9988172.

K.G. Larsen and M. Nielsen (Eds.): CONCUR 2001, LNCS 2154, pp. 566–581, 2001.
© Springer-Verlag Berlin Heidelberg 2001

and $f(X)$ is a predicate on a set X of state variables. In this case, the objective of the controller Q is to ensure that the system P enters a state satisfying $f(X)$ in one transition. While "next" properties are, *per se*, of limited interest as control objectives, algorithms for all ω-regular control objectives, such as invariance properties (specifically, $\phi = \Box f(X)$), use as subroutines algorithms that solve the single-step control problem [BL69,GH82,McN93,Tho95].

We study the single-step control problem under two very general assumptions: the closed-loop assumption, and the input-enabling assumption. The *closed-loop assumption* has that the state variables which are not given values by the system P are given values by the controller Q, and vice versa. In particular, in the *unidirectional* case, the set X of state variables is partitioned into a set O of system outputs, which are also controller inputs, and a set I of system inputs, which are also controller outputs. Later, in the *bidirectional* case, what is input and output may change from one state to the next. The *input-enabling assumption* has that the state variables that are given values by the system P cannot be in any way constrained by the controller Q, and vice versa. More precisely, for all legal environments E of P, every state of the composite system $P||E$ must have a successor state, and analogously for Q. It follows, in particular, that the controlled system $P||Q$ cannot dead-lock.

The solution and hardness of the single-step control problem depends on the precise definition of the composition operator $||$. The case most commonly considered in the literature is *unidirectional turn-based composition*, where the system P and the controller Q take turns to proceed. With each turn of Q, the system outputs O remain unchanged, and with each turn of P, the controller outputs I remain unchanged. Assume that the states in which the controller outputs can change are defined by the predicate $ctr(X)$, and that the transition relation of P is defined by the predicate $\tau(X, X')$, where unprimed state variables refer to the source state of a transition, and primed state variables refer to the target state. Then, the states for which the control objective $\bigcirc f(X)$ can be met are characterized by the quantified formula

$$(ctr(X) \rightarrow (\exists X')(O' = O \wedge \tau(X, X') \wedge f(X'))) \wedge$$
$$(\neg ctr(X) \rightarrow (\forall X')(I' = I \wedge \tau(X, X') \rightarrow f(X'))).$$

Another simple case is that of *unidirectional Moore composition*, where both the system P and the controller Q can update their respective outputs in the same transition, but they cannot do so dependent on each other. In this case, the appropriate formula is

$$(\exists I')(\forall O')(\tau(X, X') \rightarrow f(X')) \quad \text{or} \quad (\forall O')(\exists I')(\tau(X, X') \wedge f(X')).$$

If $\tau(X, X')$ is the transition relation of an input-enabling Moore system, then these two formulas are equivalent.

In the Mealy case, where some system outputs may depend on simultaneous controller outputs, and some controller outputs may depend on simultaneous system outputs, there is not a single formula that characterizes the controllable states. Instead, all possible dependencies between variables must be considered

one by one. There are, however, multiple ways in which the notion of "dependency" can be formalized. For this purpose, we introduced *type systems* for composability [dAHM00a]. In essence, such a type system offers a syntactic criterion for ensuring that all well-typed composite systems are input-enabling. The simplest type system is based on static types. A *static type* is a dependency relation between input variables and output variables. Two statically typed components can be composed if the union of the component types is acyclic. Examples of statically typed components include Reactive Modules [AH99], and sequential circuits without combinational loops. As there are meaningful systems that cannot be typed statically, we generalized static types to dynamic types. A *dynamic type* is a set of static types, one for each state. In a dynamically typed component, which outputs depend on which inputs may change from state to state. An example of a dynamically typed component that cannot be typed statically is the transparent latch.

Once a type system is adopted, we can distinguish between the fixed-type control problem and the unknown-type control problem. The *fixed-type control problem* assumes that the type of the controller is known: given a typed system P, a property ϕ, and a controller type t, construct a controller Q of type t such that $P||Q$ satisfies ϕ. The *unknown-type control problem* requires only that the controller and the resulting closed-loop system are well-typed: given a typed system P and a property ϕ, construct a typed controller Q such that $P||Q$ has a type and satisfies ϕ. In case of both static and dynamic types, the fixed-type controllable states can be characterized by formulas with partially ordered (Henkin) quantifiers [Hen61]. For example, if controller output i_1 depends on system output o_1, and controller output i_2 depends on system output o_2, and there are no other variables or dependencies, then the appropriate formula is

$$\begin{matrix}(\forall o_1')(\exists i_1')\\(\forall o_2')(\exists i_2')\end{matrix}(\tau(X, X') \wedge f(X')).$$

The unknown-type controllable states can be characterized by disjunctions of fixed-type solutions of a particularly simple kind, namely, those which contain only linearly ordered quantifiers. So, perhaps surprisingly, the unknown-type control problem is computationally simpler (in the case of boolean variables, PSPACE-complete) than the fixed-type control problem (NEXP-complete).

This concludes our review of [dAHM00a]. Following [PRSV98], we attempted to use our theory to automatically synthesize protocol converters, specifically, an interface between a PCI bus and a component using a two-phase commit protocol for communication. We found, however, that even dynamic types are too restrictive, and more general notions of synchronous composition need to be considered (cf. [BG92]). The common cases where dynamic types prove insufficient for hardware modeling fall into two classes. First, the dependencies between inputs and outputs may depend not only on the state of the system, but also on the partial successor state, as it unfolds. For example, in a digital circuit where gated clocks are used to enable or disable the latches, whether there is a dependency of the output of a latch on its input depends on whether its clock signal is asserted or not. This in turn may depend on other parts of

the system such as the current primary inputs. Second, in many digital circuits, the complete classification of ports into input and output signals is not done *a priori*. In hardware description languages, such as VHDL and Verilog, ports can be specified as bidirectional, or *IO ports*, indicating that they are sometimes used for input and at other times for output. Examples of systems that make extensive use of IO ports include bidirectional data-buses, random access memory, and the control subsystem of PCI buses, to name just a few. The use of IO ports, however, is restricted to circuit simulation; they are not included in the synthesizable fragment of HDLs.

We present a model of synchronous system composition that includes both dependent types and bidirectional types. A *dependent type* chooses the dependencies between variables based on the next-state values of other variables; a *bidirectional type* classifies variables into input and output dynamically. We then solve the single-step fixed-type and unknown-type control problems for systems with dependent and bidirectional types. Using our solution for the single-step control problem as a subroutine, controllers for more general temporal properties can be obtained in the standard way. In particular, our algorithms can be used to synthesize sequential circuits with gated clocks and IO ports.

The modeling and controller synthesis for dependent and bidirectional types is based on game-theoretic notions. In a *dependent type*, the input-output dependencies unfold in steps as the next-state variables acquire values. In a *bidirectional type*, also the commitments of variables to represent output unfold in steps as the next-state variables acquire values. These unfoldings can be viewed as a game in which the system and the controller, starting from a state, proceed to assign values to individual variables until the next state is completely specified. Our solutions of the resulting control problems are based on algorithms for solving such multi-step finite games. Although dependent and bidirectional types are significantly more general than the static and dynamic types studied previously, our results show that the computational complexity of the resulting control problems does not increase. In particular, we prove that for the most general, bidirectional types, the fixed-type control problem for boolean systems is NEXP-complete, and the unknown-type control problem is PSPACE-complete, as is already the case for static types. We also show that while the composability check for dependent types is difficult (coNP-complete), it is no more difficult than in the case of dynamic types, and simpler than checking if an untyped system is input-enabled (Π_2-complete).

2 Types for Synchronous Composition

Let V be a set of variables. In this paper all variables range over the set $\mathbb{B}$ of booleans. We denote by $\mathit{PStates}(V)$ the set of partial functions from V to $\mathbb{B}$, and by $\mathit{States}(V)$ the set of total functions. Given $v \in \mathit{PStates}(V)$, we write $\mathcal{V}ar(v) \subseteq V$ for the set of variables on which v is defined. For $X \subseteq V$, we write $v[X]$ for the restriction of v to the variables in X. For a boolean formula φ over V, we write $\varphi[\![v]\!] = \varphi[v(x_1)/x_1, \ldots, v(x_n)/x_n]$ for the formula obtained

by replacing each variable $x_i \in \mathcal{V}ar(v)$ in φ with the truth value $v(x_i)$. We write $V' = \{x' \mid x \in V\}$ for the set of corresponding primed variables, and for $v \in PStates(V)$, we write v' for the partial function in $PStates(V')$ such that $v'(x') = v(x)$ for all $x \in \mathcal{V}ar(v)$, and $v'(x')$ is undefined otherwise.

Modules. A *module* M consists of the following two components:

- A finite set V_M of *module variables*. The *states* of M are $S_M = States(V_M)$, and the *partial* (*next*) *states* of M are $R_M = PStates(V'_M)$. Unprimed variables represent current-state values; primed variables, next-state values. A pair $\langle s, t' \rangle \in S_M \times R_M$ is called an *extended state*.
- A boolean formula τ_M, called *transition predicate*, over the set $V_M \cup V'_M$ of variables; it relates the current-state and next-state values of the module variables. The state $t \in S_M$ a *macro-step successor* of the state $s \in S_M$ if $\tau_M[\![s \cup t']\!]$ is true. For a variable $x \in V_M$, the extended state $\langle s, u' \rangle \in S_M \times R_M$ is a (*micro-step*) (x, τ_M)-*successor* of the extended state $\langle s, t' \rangle$ if $x' \notin \mathcal{V}ar(t')$ and there exists $b \in \mathbb{B}$ such that $u' = t' \cup \{(x', b)\}$ and $\tau_M[\![s \cup u']\!]$ is satisfiable.

Given two modules M and N, the (*synchronous*) *composition* $M||N$ is the module with $V_{M||N} = V_M \cup V_N$ and $\tau_{M||N} = \tau_M \wedge \tau_N$. A module M is *nonblocking* if every state has a macro-step successor; that is, for all states $s \in S_M$, there exists a state $t \in S_M$ such that $\tau_M[\![s \cup t']\!]$ is true. Synchronous composition is problematic because it may cause blocking even if both components are nonblocking. This is particularly undesirable in control applications, where we usually want to rule out controllers that achieve the control objective simply by blocking the plant from progressing.

Proposition 1 *It is Π_2-complete to check if the composition of two nonblocking modules is nonblocking.*

In order to simplify the check that synchronous composition is nonblocking, we augment modules with *types*. We indicate by (M, γ) the pair consisting of a module M and its type γ. We will consider several classes of module types. For each such class T, we will define a notion of T-*composability*, which specifies whether two typed nonblocking modules (M, γ) and (N, γ') with $\gamma, \gamma' \in T$ can be composed into a single nonblocking module.

Single-step control. Given a class T of types and a typed module (M, γ) with $\gamma \in T$, a T-*controller* for (M, γ) is a typed module (N, γ') with $\gamma' \in T$ which is T-composable with (M, γ). The *single-step control problem* for T asks, given a typed module (M, γ), a state $s \in S_M$, and a boolean formula φ over V_M, if there is a T-controller (N, γ') for (M, γ) such that for all states $t \in S_M$, if $\tau_{M\|N}[\![s \cup t']\!]$ is true, then $\varphi[\![t]\!]$ is true. The controller N ensures that starting from s, the predicate φ holds after one step of the *closed-loop system* $M\|N$, and it does so without blocking the progress of M. If the answer is Yes, then the state s is *single-step* T-*controllable* by (N, γ') w.r.t. the *control objective* φ. We distinguish two kinds of control problems. In the *unknown-type* control problem $((M, \gamma), s, \varphi)$, we are free to choose the type γ' of the controller; in the *fixed-type*

control problem $((M,\gamma), s, \varphi, \gamma')$, the type γ' of the controller is specified as part of the problem statement.

IO-type modules. We partition the module variables into input and output, when composing modules, we disallow variables to be output from more than one module. An *IO-type module* (M, π_M), or simply *IO-module*, consists of a module M and a partition $\pi_M = (V_M^i, V_M^o)$ of the module variables V_M into a set V_M^i of input variables and a set $V_M^o = V_M \setminus V_M^i$ of output variables. We refer to π_M as an *IO-type* for M. Two IO-modules (M, π_M) and (N, π_N) are *IO-composable* if their output variables are disjoint; that is, $V_M^o \cap V_N^o = \emptyset$. The *IO-composition* is the IO-module $(M \| N, \pi_{M\|N})$, where $V_{M\|N}^o = V_M^o \cup V_N^o$ and $V_{M\|N}^i = (V_M^i \cup V_N^i) \setminus V_{M\|N}^o$. The fact that two IO-modules are IO-composable does not suffice to guarantee that their composition is nonblocking. Therefore, we either restrict the transition predicate (as in the case of Moore modules), or we augment IO-types with additional information (such as dependency relations).

Moore modules. A *Moore module* (M, π_M) is an IO-module such that (a) the module M is non-blocking, and (b) the next values of output variables V_M^o do not depend on the next values of input variables; that is, for all states $s, t, u \in S_M$, if $\tau_M[\![s \cup t']\!]$ and $t[V_M^o] = u[V_M^o]$, then $\tau_M[\![s \cup u']\!]$. The composition of two IO-composable Moore modules is nonblocking.

Statically typed modules. A *dependency relation* for an IO-module (M, π_M) is an acyclic binary relation $\succ \subseteq V_M^o \times V_M$ between the output variables and the module variables (acyclicity means that the transitive closure is irreflexive). The IO-module (M, π_M) *respects* the dependency relation $\succ$ at state $s \in S_M$ if for all states $t \in S_M$ with $\tau_M[\![s \cup t']\!]$, for each subset $Y^i \subseteq V_M^i$ of input variables, and for each truth-value assignment u^i to the variables in Y^i, there is a state u with $\tau_M[\![s \cup u']\!]$ such that $u[Y^i] = u^i$, and $u[Y] = t[Y]$ for $Y = \{z \in V_M \mid (\text{not } z \succ^* y) \text{ for all } y \in Y^i\}$, where $\succ^*$ is the reflexive-transitive closure of $\succ$. A *statically typed module* $(M, \pi_M, \succ_M)$ consists of an IO-module (M, π_M) and a dependency relation $\succ_M$ for (M, π_M) such that (a) the module M is nonblocking, and (b) the IO-module (M, π_M) respects the dependency relation $\succ_M$ at all states in S_M. We refer to the pair $(\pi_M, \succ_M)$ as a *static type* for M. Two statically typed modules $(M, \pi_M, \succ_M)$ and $(N, \pi_N, \succ_N)$ are *statically composable* if (1) they are IO-composable and (2) the relation $\succ_M \cup \succ_N$ is acyclic. The composition of two statically composable modules is nonblocking. These modules can be used to model sequential circuits without combinational loops.

Dynamically typed modules. A *composite dependency relation* for an IO-module (M, π_M) is a set $D = \{(\psi^1, \succ^1), \ldots, (\psi^m, \succ^m)\}$ of pairs, where each ψ^i is a boolean formula over the module variables V_M, and each $\succ^i$ is a dependency relation for (M, π_M), such that for every state $s \in S_M$, there is exactly one formula ψ^i, $1 \le i \le m$, such that $\psi^i[\![s]\!]$ is true. If $\psi^i[\![s]\!]$, then we write $\succ^s$ for the corresponding dependency relation $\succ^i$. A *dynamically typed module* (M, π_M, D_M) consists of an IO-module (M, π_M) and a composite dependency relation $D_M = \{(\psi_M^i, \succ_M^i) \mid 1 \le i \le m\}$ for (M, π_M) such that (a) the module M is nonblocking, and (b) at every state $s \in S_M$, the module M respects the dependency relation $\succ_M^s$. We refer to the pair (π_M, D_M) a dynamic type for the

module M. Two dynamically typed modules (M, π_M, D_M) and (N, π_N, D_N) are *dynamically composable* if (1) they are IO-composable and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$ for which the conjunction $\psi_M^i \wedge \theta_N^j$ is satisfiable. The composition of two dynamically composable modules is nonblocking. These modules can be used to model circuits with transparent latches that may contain combinational loops [dAHM00a].

Theorem 1 [dAHM00a] *It can be checked in linear time if two statically typed modules are statically composable. It is coNP-complete to check if two dynamically typed modules are dynamically composable.*

Theorem 2 [dAHM00a] *The single-step control problem for Moore modules is Σ_2-complete. The single-step unknown-type control problems for statically and dynamically typed modules are PSPACE-complete. The single-step fixed-type control problems for statically and dynamically typed modules are NEXP-complete.*

The goal of this paper is to extend our type systems to capture wider classes of nonblocking synchronous composition, and study the resulting control problems. In particular, we add to our list of type classes *dependent types* and *bidirectional types*. Dependent types generalize dynamic types by allowing the dependency relation to be a function not only of the current state, but also of the partial next state. Bidirectional types further generalize the dependent types by removing the requirement that module variables are partitioned into input and output variables a priori. Rather, the choice of which variables are used as inputs and outputs is performed dynamically, while the values of the variables themselves are chosen.

3 Macro-Steps as Micro-Step Graphs

The following notions will be used in the definition of both dependent and bidirectional types. The variable dependency relation establishes the possible orders in which the variables can be assigned a value in order to determine the next state, and the dependencies among the values chosen. The micro-step graph makes explicit the partial states traversed as a new macro-step successor is determined. We will solve single-step control problems by considering games on this micro-step graph.

Variable dependency relation. A *variable dependency relation* for M is a set $C = \{(\psi^1, \succcurlyeq^1), \ldots, (\psi^m, \succcurlyeq^m)\}$ of pairs, where each ψ^i is a boolean formula over the unprimed and primed module variables $V_M \cup V_M'$, and each $\succcurlyeq^i \subseteq V_M \times 2^{V_M}$ is a binary relation with the intention that if ψ^i holds in an extended state, and $x \succcurlyeq^i Y$, then x can be given a next value, and this value can depend on the next values of the variables in Y. The set C is an *IO-variable dependency relation* for an IO-module (M, π_M) if $\succcurlyeq^i \subseteq V_M^o \times 2^{V_M}$ for all $1 \leq i \leq m$; that is, dependencies are specified only for output variables. A variable dependency relation is a syntactic object; to make the variable dependencies more explicit, we

define the corresponding *dependency function* $\tilde{C}$: $S_M \times R_M \times V_M \to 2^{V_M}$ as the function that, given an extended state $\langle s, t' \rangle$ and a variable x, specifies the set of variables on which x depends, as $\tilde{C}(s, t', x) = \bigcup\{Y \mid (\psi, (x, Y)) \in C$ and $\psi[\![s \cup t']\!]$ contains no free variables and is true$\}$. The variable x is *enabled* for (M, C) at the extended state $\langle s, t' \rangle$ if $\tilde{C}(s, t', x) \subseteq \mathcal{V}ar(t)$. The module M *respects* the variable dependency relation C if for every pair of extended states $\langle s, t' \rangle$ and $\langle s, u' \rangle$ of M, for every variable $x \in V_M$ that is enabled for (M, C) at both extended states, and for every $b \in \mathbb{B}$, if $t[\tilde{C}(s, t', x)] = u[\tilde{C}(s, u', x)]$, then the extended state $\langle s, t' \cup \{(x', b)\}\rangle$ is an (x, τ_M)-successor of $\langle s, t' \rangle$ iff the extended state $\langle s, u' \cup \{(x', b)\}\rangle$ is an (x, τ_M)-successor of $\langle s, u' \rangle$. If the transition predicate of a module is specified by a set Γ of nondeterministic guarded commands, then the variable dependency relation can be deduced from Γ as follows: for each guarded command $[\![g \to x' = e$ in Γ, let $(g, \succcurlyeq) \in C$, where $\succcurlyeq = \{(x, Y) \mid y \in Y$ iff y' occurs in g or in $e\}$.

Micro-step graph. Consider two modules M and N together with variable dependency relations C_M and C_N. Let $V = V_M \cup V_N$. The micro-step graph of M and N encodes the sequential process by which M and N update the variable values from a state to its macro-step successor. Formally, for a state $s \in S$, the *micro-step graph* $MG_s(M, C_M, N, C_N)$ is a directed acyclic graph whose vertices are the tuples $\langle s, t', U, W \rangle$, where $s \in States(V)$, $t' \in PStates(V')$, $U \subseteq V_M$, and $W \subseteq V_N$, together with the additional distinguished vertex $\perp$, which is used to denote an illegal configuration. The edges of $MG_s(M, C_M, N, C_N)$ are partitioned in M-edges and N-edges; they are defined as follows, for all vertices $\alpha = \langle s, t', U, W \rangle$ and all variables $x \in V$:

- If $x \notin \mathcal{V}ar(t)$ and x is enabled for (M, C_M) at the extended state $\langle s, t' \rangle$, then for each (x, τ_M)-successor $\langle s, u' \rangle$ of α, if $\langle s, u' \rangle$ is also an (x, τ_N)-successor of α, then there is an M-edge from α to the vertex $\langle s, u', U \cup \{x\}, W \rangle$; and if $\langle s, u' \rangle$ is not an (x, τ_N)-successor of α, then there is an M-edge from α to $\perp$. The N-edges are defined symmetrically.
- If $x \in (\mathcal{V}ar(t) \cap W)$ and x is enabled for (M, C_M) at the extended state $\langle s, t' \rangle$, then there is an M-edge from α to $\perp$. The N-edges are defined symmetrically.

A vertex of $MG_s(M, C_M, N, C_N)$ is *terminal* if it does not have any outgoing edges. Note that the micro-step graph has the following properties: there are at most $2^{O(|V|)}$ vertices, the size of each vertex is at most $4 \cdot |V|$, and the depth of the graph is at most $|V| + 1$. If a vertex α of $MG_s(M, C_M, N, C_N)$ has both outgoing M- and N-edges, then α is *mixed*. The *M-reduced micro-step graph* $RMG_s^M(M, C_M, N, C_N)$ is the micro-step graph obtained from $MG_s(M, C_M, N, C_N)$ by pruning, for all mixed vertices α, all N-edges outgoing from α. Intuitively, the reduced graph represents the situation in which the module M has precedence over N in updating variable values.

4 Dependent-Type Modules

Consider an IO-module (M, π_M) together with an IO-variable dependency relation C_M for (M, π_M). Let (E, π_E, C_E) consist of the module E with the input

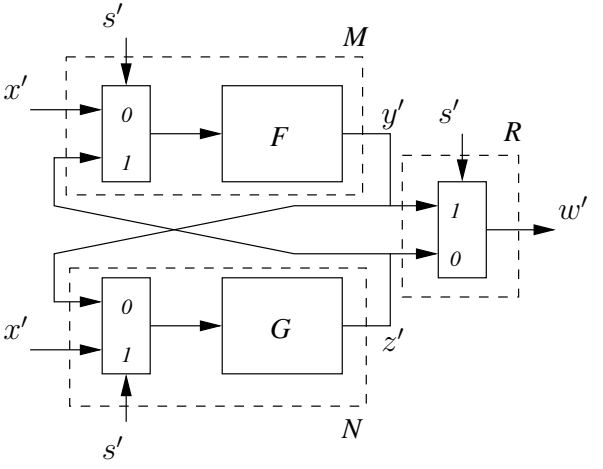


Fig. 1. A cyclic circuit composed of three modules M, N, and R. It performs the following function: if s' then $w' = F(G(x'))$ else $w' = G(F(x'))$, where F and G are two combinational blocks, such as a shifter and adder.

variables $V_E^i = V_M^o$, the output variables $V_E^o = V_M^i$, the transition predicate $\tau_E = \mathrm{T}$, and the IO-variable dependency relation $\{(\mathrm{T}, \{(x, \emptyset)\}) \mid x \in V_E^o\}$. The triple (E, π_E, C_E) is the *most general dependent-type environment* for (M, π_M); it assigns nondeterministically values to the input variables of (M, π_M). The triple (M, π_M, C_M) is a *dependent-type module* if (a) the module M respects the variable dependency relation C_M and (b) for every state $s \in S_M$, if there is a path in the micro-step graph $MG_s^M(M, C_M, E, C_E)$ from the initial vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ to a terminal vertex α, then $\alpha \neq \perp$, and $\alpha = \langle s, t', V_M^o, V_M^i \rangle$ for some state $t \in \mathit{States}(V_M)$. Condition (b) states that for all environment inputs, the module M does not block. We refer to the pair (π_M, C_M) as a *dependent type* for M.

Example 1 [Mal94] Cyclic circuits are often used in hardware systems for minimizing the circuit size. As an example, consider the circuit in Figure 1. The output w is a function of the inputs s and x. The circuit consists of three dependent-type modules M, N and R. In guarded commands, they are

$$M = \begin{array}{l} [\!] \ \neg s' \ \ \to y' = F(x') \\ [\!] \ \ s' \ \ \to y' = F(z') \end{array} \quad N = \begin{array}{l} [\!] \ \neg s' \ \ \to z' = G(y') \\ [\!] \ \ s' \ \ \to z' = G(x') \end{array} \quad R = \begin{array}{l} [\!] \ \neg s' \ \ \to w' = z' \\ [\!] \ \ s' \ \ \to w' = y' \end{array}$$

Module M has the variables $V_M^o = \{y\}$ and $V_M^i = \{s, x, z\}$, and the IO-variable dependency relation $\{(\neg s', \{(y, \{x, s\})\}), (s', \{(y, \{z, s\})\})\}$. Module N has the variables $V_N^o = \{z\}$ and $V_N^i = \{s, x, y\}$, and the IO-variable dependency relation $\{(\neg s', \{(z, \{y, s\})\}), (s', \{(z, \{x, s\})\})\}$. Module R has the variables $V_R^o = \{w\}$ and $V_R^i = \{y, z\}$, and the IO-variable dependency relation $\{(\neg s', \{(w, \{z, s\})\}), (s', \{(w, \{y, s\})\})\}$. ∎

Proposition 2 *Every IO-module with a dependent type is nonblocking. Every nonblocking IO-module has a dependent type.*

Composition. Two dependent-type modules (M, π_M, C_M) and (N, π_N, C_N) are *dependent-type composable* if (1) they are IO-composable and (2) the IO-composition $(M\|N, \pi_{M\|N})$ respects the IO-variable dependency relation $C_{M\|N} = C_M \cup C_N$. Then the pair $(\pi_{M\|N}, C_{M\|N})$ is a dependent type for the composite module $M\|N$. The following theorem shows that checking composability for dependent-type modules has the same worst-case complexity as for dynamically typed modules.

Theorem 3 *It is coNP-complete to check if two dependent-type modules are dependent-type composable.*

Proof. (sketch) Given two dependent-type modules (M, π_M, C_M) and (N, π_N, C_N), to show that they are not composable one can guess a path $\langle s, \emptyset, \emptyset, \emptyset\rangle, \langle s, t'_1, U_1, W_1\rangle, \langle s, t'_2, U_2, W_2\rangle, \ldots, \langle s, t'_n, U_n, W_n\rangle = \alpha$ in the micro-step graph $MG_s(M, C_M, N, C_N)$, and check that $\mathcal{V}ar(t_n) \subsetneq V_{M\|N}$, and that there is no outgoing edge from α. This last condition can be checked by checking that $\mathcal{V}ar(t_n)$ contains all input variables $V^i_{M\|N}$, and that no output variable undefined in t'_n is enabled at the extended state $\langle s, t'_n\rangle$. Note that this check requires only polynomial time, because a variable can be enabled only when all variables that occur in its enabling condition are assigned a value by $s \cup t'_n$. ∎

Dependent types capture a larger class of nonblocking synchronous composition than dynamically typed modules, as shown by the following proposition. For a dependent-type module (M, π_M, C_M), the variable $x \in V_M$ depends on $y \in V_M$ at a state $s \in S_M$, written $x \succ^s y$, if there exists a partial state $t' \in S_M$ and a pair $(\psi, \succcurlyeq) \in C_M$ such that $\psi[\![s \cup t']\!]$ is true and $x \succcurlyeq Y$ with $y \in Y$. Then $D_M = \{(s, \succ^s) \mid s \in S_M\}$ is a dynamic type for M.

Proposition 3 *There are two dependent-type modules that are dependent-type composable but not composable if viewed as dynamically typed.*

Example 2 The dependent-type modules M, N, and R from Example 1 are dependent-type composable. If these modules are viewed as dynamically typed modules, the output of each module will depend on the respective inputs. Hence there will be a cyclic dependency in the union of their dependency relations, namely, $y \succ z$ and $z \succ y$ at all states. Since dynamically typed modules do not permit cyclic dependencies, these modules are not dynamically composable. ∎

Unknown-type control. By relaxing the composability requirement of modules from dynamic to dependent types, we can control a larger class of modules.

Proposition 4 *There is a dependent-type module (M, π_M, C_M), a control objective φ over V_M, and a state $s \in S_M$ such that s is single-step controllable w.r.t. φ by a dependent-type controller but not by a dynamically typed controller.*

Example 3 Let M be the module with $V_M^i = \{u, v\}$ and $V_M^o = \{x\}$, and the following guarded commands:

$$M = \begin{array}{ll} [\!] & u' \rightarrow x' = \neg v' \\ [\!] & \neg u' \rightarrow x' = \mathrm{T} \\ [\!] & \neg u' \rightarrow x' = \mathrm{F} \end{array}$$

The control objective is $x = v$. There is no dynamically typed controller (at any state), because such a controller would have x depend on v, and M can set x to be $\neg v$. But a dependent-type controller can change the dependencies, namely, have x not depend on any variable and have v depend on x. The following is a dependent-type controller:

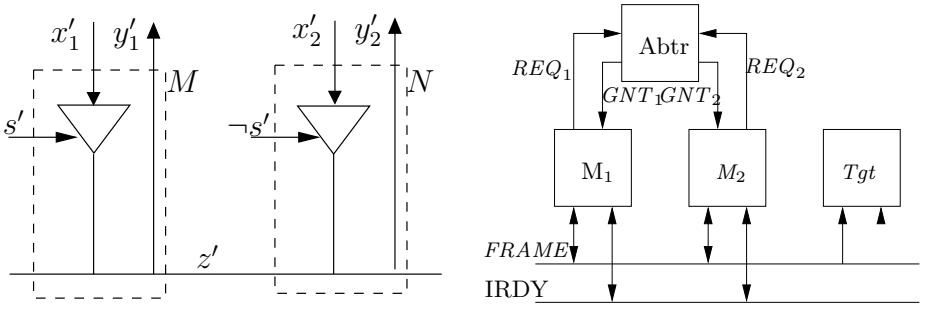
$$N = \begin{array}{ll} [\!] & \mathrm{T} \rightarrow u' = \mathrm{F} \\ [\!] & \mathrm{T} \rightarrow v' = x' \end{array} \blacksquare$$

Consider the single-step unknown-type control problem $((M, \pi_M, C_M), s, \varphi)$. It is convenient to view this control problem as a game between the dependent-type module (M, π_M, C_M) and its controller. The game is played on the M-reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, where (E, C_E) is the most general dependent-type environment for (M, π_M). Note that every nonterminal vertex of the reduced micro-step graph either has only outgoing M-edges or has only outgoing E-edges. We call the vertices with only outgoing M-edges the *module vertices*, and the vertices with only outgoing E-edges the *environment vertices*. Then we solve the game by the following marking algorithm. A terminal vertex $\alpha = \langle s, t', U, W\rangle$ is marked if $U \cup W = V_M$ and $\varphi[\![t]\!]$ is true. A module vertex α is marked if all successors β of α are marked. An environment vertex α is marked if some successor β of α is marked. The answer to the given single-step unknown-type control problem is Yes iff the vertex $\langle s, \emptyset, \emptyset, \emptyset\rangle$ is marked.

If the answer to the control problem is Yes, then the method also suggests a way of synthesizing a dependent-type controller (N, π_N, C_N) as a set of guarded commands Γ. Given a state $s \in S_M$, denote by χ_s the *characteristic formula* of s, defined by $\chi_s = \bigwedge\{x \mid (x, \mathrm{T}) \in s\} \wedge \bigwedge\{\neg x \mid (x, \mathrm{F}) \in s\}$. The controller has the output variables $V_N^o = V_M^i$ and input variables $V_N^i = V_M^o$. For every marked environment vertex $\alpha = \langle s, t', U, W\rangle$ in the reduced micro-step graph, choose one marked successor $\langle s, t' \cup \{(x', b)\}, U, W \cup \{x\}\rangle$ of α, and add to Γ the guarded command $[\![\chi_s \wedge \chi_{t'} \rightarrow x' = b$. Like composability checking, the single-step unknown-type control problem for dependent-type modules is no harder than its counterpart for dynamically typed modules.

Theorem 4 *The single-step unknown-type control problem for dependent-type modules is PSPACE-complete. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

Fixed-type control. The fixed-type control problem is computationally harder than the unknown-type control problem, although it is no harder than its counterpart for dynamically typed modules. The additional complexity is due to the fact that we need to construct explicitly the micro-step graph.



(a) Bidirectional data-bus (b) PCI

Fig. 2. Examples of bidirectional modules. Figure 2(a): A system with two processors communicating through a bidirectional bus. The variable z is output of the module M if $s' = \text{T}$, and it is output of N if $s' = \text{F}$. Figure 2(b): A PCI system with two master devices and one target device. The figure also shows the arbiter.

Theorem 5 *The single-step fixed-type control problem for dependent-type modules is NEXP-complete. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

Proof. (sketch) Given a dependent-type module (M, π_M, C_M), a state $s \in S_M$, a control objective φ over V_M, and a dependent type (π_N, C_N) for the controller, one can guess a subgraph G of the reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, for the most general dependent-type environment (E, π_E, C_E), and check that (1) the vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ is in G; (2) for all module vertices α in G, all successors of α are also in G; (3) for all environment vertices α in G, there is exactly one successor β of α in G such that (α, β) is an edge in G; (4) for all terminal vertices $\langle s, t', \cdot, \cdot \rangle$, the formula $\varphi[\![t]\!]$ is true; (5) the controller respects the IO-variable dependency relation C_N at all environment vertices in G. All of these conditions can be checked in time polynomial in the size of G. NEXP-hardness comes from the fact that dependent-type modules can encode dynamically typed modules. ∎

5 Bidirectional Modules

A *bidirectional module* (M, C_M) consists of a module M and a variable dependency relation C_M for M such that M respects C_M. We refer to C_M as a *bidirectional type* for M. Note that a bidirectional type does not contain an IO-type. The information about which variables can be outputs of a bidirectional module is encoded instead by its variable dependency relation: if a variable $x \notin \mathcal{V}ar(t')$ is enabled at the extended state $\langle s, t' \rangle$, then x can be assigned a value at $\langle s, t' \rangle$, and thus used as an output. Unlike the various typed modules we defined previously, bidirectional modules do not guarantee nonblocking. This is because a

bidirectional module may not be able to produce suitable "outputs" (i.e., values for some module variables) for all possible environment "inputs" (i.e., values for the other module variables). For instance, the environment may try to assign a value to a variable that already has a value assigned by the module. Hence, the environment has to be specified explicitly, and two bidirectional modules are composable only if the result is nonblocking.

Composition. Informally speaking, two bidirectional modules are composable if in all macro-steps, every module variable is assigned a value exactly once (by either of the modules). Then the composition is nonblocking, and each variable is output of exactly one component. Given two bidirectional modules (M, C_M) and (N, C_N), let $V = V_M \cup V_N$. The modules (M, C_M) and (N, C_N) are *bidirectionally composable* if for all $s \in \mathit{States}(V)$, if there is a path in the micro-step graph $MG_s(M, C_M, N, C_N)$, from the initial vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ to a terminal vertex α, then $\alpha \neq \bot$, and $\alpha = \langle s, t', U, W \rangle$ with $U \cup W = V$. Unfortunately, checking bidirectional composability is computationally harder than checking composability for the other previous types.

Theorem 6 *It is Π_2-complete to check if two bidirectional modules are bidirectionally composable.*

The additional hardness comes from the fact that when composing two bidirectional modules, one of the modules may choose a value for the common variables such that the other module blocks. This cannot not happen with statically typed, dynamically typed, or dependent-type modules, because they do not block on any inputs they receive. For a bidirectional module (M, C_M), let $V_M^o = \{x \in V_M \mid$ there exists a pair $(\psi, \succcurlyeq) \in C_M$ such that $x \succcurlyeq Y$ for some $Y \subseteq V_M\}$. Then the pair (π_M, C_M), where $\pi_M = (V_M^o, V_M \setminus V_M^o)$, is a dependent type for M.

Proposition 5 *There are two bidirectional modules that are bidirectionally composable but not composable if viewed as dependent-type modules.*

Example 4 In a multi-processor system, a bidirectional data-bus can be modeled as a bidirectional module. Typically, at most one processor has the right to write to the data-bus, while the others can only read from the data-bus. Figure 2(a) shows a simplified system in which there are two processors M and N communicating via the data-bus z. The variable z may be an output of either M or N, depending on the value of s. In guarded commands, M and N are:

$$M = \begin{array}{l} [\!] \; s' \to z' = x_1' \\ [\!] \; \mathrm{T} \to y_1' = x_1' \end{array} \qquad N = \begin{array}{l} [\!] \; \neg s' \to z' = x_2' \\ [\!] \; \mathrm{T} \to y_2' = x_2' \end{array}$$

We assume that M and N are composed with an environment that nondeterministically chooses values for the variables s, x_1, and x_2. These three modules are bidirectionally composable, but they are not composable if viewed as dependent-type modules because z is output of both M and N. ∎

Example 5 (PCI bus arbitration) Bidirectional modules can be used to model the PCI bus protocol[SIG]. The PCI bus is an industry standard commonly used for interfacing between the core computer system (e.g., CPU, memory, etc.) and the peripheral devices (e.g., audio, video etc.). Typically, the master devices attached to a PCI bus may request to own the bus in order to communicate with the respective target devices. Once a request is granted by the arbiter, the bus will be owned by the selected master device and only this master device or its target device can write onto the bus.

Consider an instance of the bus protocol depicted in Figure 2(b). There are two master devices and one target device. During the arbitration phase, the master devices may request to own the bus. When ownership is granted by the arbiter, the selected master device checks if the bus is idle (by observing that the values of both signals *FRAME* and *IRDY* are high), and then drives these two signals, so that they become outputs of this master device. Note that the two signals can be output of either master device, depending on the decision of the arbiter. The following guarded commands model the arbitration phase of the system. The two master devices M_i, $i \in \{1, 2\}$ can be described as:

$$M_i = \begin{array}{l} [\!] \ \mathrm{T} \rightarrow \ REQ_i = \mathrm{T} \\ [\!] \ \mathrm{T} \rightarrow \ REQ_i = \mathrm{F} \\ [\!] \ FRAME \wedge IRDY \wedge GNT_i \rightarrow \ FRAME' = \mathrm{F}; IRDY' = \mathrm{T} \end{array}$$

The arbiter *Abtr* can be described as:

$$Abtr = \begin{array}{l} [\!] \ \mathrm{T} \rightarrow \ GNT_1' = \mathrm{F}; GNT_1' = \mathrm{F} \\ [\!] \ REQ_1 \rightarrow \ GNT_1' = \mathrm{T}; GNT_1' = \mathrm{F} \\ [\!] \ REQ_2 \rightarrow \ GNT_1' = \mathrm{F}; GNT_2' = \mathrm{T} \ \blacksquare \end{array}$$

Unknown-type control. Relaxing the composability requirement of modules from dependent to bidirectional types, we can control a larger class of modules.

Proposition 6 *There is a bidirectional module (M, C_M), a control objective φ over V_M, and a state of $s \in S_M$ such that s is single-step controllable w.r.t. φ by a bidirectional controller but not by a dependent-type controller.*

Example 6 Let M be the module with the variables $V = \{x, u\}$ and the following guarded commands:

$$M = \begin{array}{l} [\!] \ u' \rightarrow \ x' = \mathrm{F} \\ [\!] \ \neg u' \rightarrow \end{array}$$

The control objective is $x = \mathrm{T}$. There is no dependent-type controller (at any state), because if the controller sets u' to T, then M will set x' to F. On the other hand, if the controller sets u' to F, then M does not have an enabled guarded command to assign a value to x'. But a bidirectional controller can bind the variable x to its output and set its value to T. The following is a possible bidirectional controller:

$$N = \begin{array}{l} [\!] \ \mathrm{T} \ \rightarrow \ u' = \mathrm{F} \\ [\!] \ \mathrm{T} \ \rightarrow \ x' = \mathrm{T} \ \blacksquare \end{array}$$

Consider the single-step unknown-type control problem $((M, C_M), s, \varphi)$. As for dependent-type modules, this control problem can be viewed as a game played between the module and the controller on the M-reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, where (E, C_E) is the most general bidirectional environment for M, defined by $V_E = V_M$, $\tau_E = \mathrm{T}$, and variable dependency relation $C_E = \{(\mathrm{T}, \{(x, \emptyset)\}) \mid x \in V_E\}$. A vertex $\langle s, t', U, W\rangle$ is a *stop node* if $U \cup W = V_M$. To solve the game, we use the following marking algorithm. A stop node $\langle s, t', U, W\rangle$ is marked if $\varphi[\![t]\!]$ is true, or if it does not have any outgoing M-edges. For all other vertices α, if it is a module vertex, then it is marked if all successors of α are marked; and if it is an environment vertex, then it is marked if some successor of α is marked. The answer to the given single-step unknown-type control problem is Yes iff the vertex $\langle s, \emptyset, \emptyset, \emptyset\rangle$ is marked.

If the answer to the control probem is Yes, then we can synthesize a bidirectional controller as follows. We construct a subgraph of the reduced micro-step graph by keeping all successors of each marked environment vertex, but only one marked successor of each marked module vertex. This subgraph, called the *control graph,* may not be unique. The following observation is crucial: in every control graph there are no distinct vertices $\langle s, t_1', U_1, W_1\rangle$ and $\langle s, t_2', U_2, W_2\rangle$ with $t_1 = t_2$ but $U_1 \neq U_2$. We can therefore synthesize a bidirectional controller (N, C_N) as a set of guarded commands Γ from *any* control graph. The controller has module variables $V_N = V_E$. For every environment vertex $\alpha = \langle s, t', U, W\rangle$ in the control graph, if $\beta = \langle s, u', U, W \cup x\rangle$ is the unique successor of α, then we add to Γ the guarded command $[\![\chi_s \wedge \chi_{u'} \to x' = b$. Note that by the observation on control graphs, no two guarded commands have identical guards. In summary, the single-step unknown-type control problem for bidirectional modules is no harder than its counterpart for dependent-type modules.

Theorem 7 *The single-step unknown-type control problem for bidirectional modules is PSPACE-complete. Moreover, if the answer is Yes, then a bidirectional controller can be synthesized.*

Fixed-type control. Also the single-step fixed-type control problem for bidirectional modules is no harder than its counterpart for dependent-type modules.

Theorem 8 *The single-step fixed-type control problem for bidirectional modules is NEXP-complete. Moreover, if the answer is Yes, then a bidirectional controller can be synthesized.*

Proof. (sketch) The proof is very similar to that for dependent-type modules. One adds to the list of things to check that at each node of the guessed subgraph there is no conflict in the value assignments to the variables. ∎

References

[AdAHM99] R. Alur, L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. Automating modular verification. In *Concurrency Theory*, LNCS 1664, pp. 82–97. Springer, 1999.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[dAHM00a] L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. The control of synchronous systems. In *Concurrency Theory*, LNCS 1877, pp. 458–473. Springer, 2000.

[dAHM00b] L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. Detecting errors before reaching them. In *Computer-Aided Verification*, LNCS 1855, pp. 186–201. Springer, 2000.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: design, semantics, implementation. *Science Computer Programming*, 19:87–152, 1992.

[GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. Symp. Theory of Computing*, pp. 60–65. ACM Press, 1982.

[Hen61] L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods*, pp. 167–183. Polish Scientific Publishers, 1961.

[Mal94] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13:950–956, 1994.

[McN93] R. McNaughton. Infinite games played on finite graphs. *Ann. Pure and Applied Logic*, 65:149–184, 1993.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Symp. Principles of Programming Languages*, pp. 179–190. ACM Press, 1989.

[PRSV98] R. Passerone, J.A. Rowson, A.L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proc. Design Automation Conference*, pp. 8–13. ACM Press, 1998.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[SIG] PCI SIG. PCI local bus specification, rev. 2.2.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science*, LNCS 900, pp. 1–13. Springer, 1995.

Author Index