ARCoSS

LNCS 6901

Joost-Pieter Katoen
Barbara König (Eds.)

CONCUR 2011 – Concurrency Theory

22nd International Conference, CONCUR 2011
Aachen, Germany, September 2011
Proceedings

Springer

Lecture Notes in Computer Science 6901

Joost-Pieter Katoen Barbara König (Eds.)

CONCUR 2011 – Concurrency Theory

22nd International Conference, CONCUR 2011
Aachen, Germany, September 6-9, 2011
Proceedings

Springer

Volume Editors

Joost-Pieter Katoen
RWTH Aachen University, Fachgruppe Informatik
Software Modeling and Verification
Ahornstraße 55, 52056 Aachen, Germany
E-mail: katoen@cs.rwth-aachen.de

Barbara König
Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften
Abteilung für Informatik und Angewandte Kognitionswissenschaft
Lotharstraße 65, 47057 Duisburg, Germany
E-mail: barbara_koenig@uni-due.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-23216-9 e-ISBN 978-3-642-23217-6
DOI 10.1007/978-3-642-23217-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011934620

CR Subject Classification (1998): D.2, F.3, C.2, D.3, H.4, D.2.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 22nd Conference on Concurrency Theory (CONCUR 2011) held in Aachen, Germany, September 6-9, 2011. The purpose of the CONCUR conference is to bring together researchers, developers, and students in order to advance the theory of concurrency and promote its applications.

This edition of the conference attracted 94 submissions. We would like to thank all their authors for their interest in CONCUR 2011. After careful reviewing and discussions, the Program Committee selected 32 papers for presentation at the conference. Each submission was reviewed by at least three reviewers, who wrote detailed evaluations and gave insightful comments. The conference Chairs would like to thank the Program Committee members and all their subreviewers for their excellent work, as well as for the constructive discussions. We are grateful to the authors for having revised their papers so as to address the comments and suggestions by the referees.

The conference program was greatly enriched by the invited talks by Parosh Aziz Abdulla (joint invited speaker with QEST 2011), Rachid Guerraoui, Wil van der Aalst, and Ursula Goltz.

This year the conference was jointly organized with the 8th International Conference on Quantitative Evaluation of Systems (QEST 2011) and the 6th International Symposium on Trustworthy Global Computing (TGC 2011). On the one hand, we wanted to acknowledge the increasing application of quantitative methods in concurrency theory and to strengthen the link to work on quantitative system evaluation. On the other hand, we wanted to emphasize the importance of safety and reliability in global computing.

In addition, CONCUR included the following satellite workshops:

- Third Workshop on Computational Models for Cell Processes (CompMod 2011)
- 18th International Workshop on Expressiveness in Concurrency (EXPRESS 2011)
- 10th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2011)
- Third Workshop on Games for Design, Verification and Synthesis (GASICS 2011)
- 4th International “Logics, Agents, and Mobility” Workshop (LAM 2011)
- ERCIM Working Group Meeting on Models and Logics for Quantitative Analysis (MLQA 2011)
- 9th International Workshop on Security Issues in Concurrency (SecCo 2011)
- Workshop on Structural Operational Semantics (SOS 2011)
- Young Researchers Workshop on Concurrency Theory (YR-CONCUR 2011)

We would like to thank everybody who contributed to the organization of CONCUR 2011. Furthermore, we thank the RWTH Aachen University, IVU

Traffic Technologies AG, the European Microsoft Innovation Center (EMIC) GmbH, and the Deutsche Forschungsgemeinschaft (DFG) for their financial support.

We are also grateful to Andrei Voronkov for providing us with his conference software system EasyChair, which was extremely helpful for the Program Committee discussions and the production of the proceedings.

September 2011
Joost-Pieter Katoen
Barbara König

Organization

Steering Committee

Roberto Amadio	Université Paris Diderot, France
Jos Baeten	Eindhoven University of Technology, The Netherlands
Eike Best	Carl-von-Ossietzky-Universität Oldenburg, Germany
Kim Larsen	Aalborg University, Denmark
Ugo Montanari	Università di Pisa, Italy
Scott Smolka	Stony Brook University, USA

Program Committee

Christel Baier	Technical University of Dresden, Germany
Paolo Baldan	Università di Padova, Italy
Ahmed Bouajjani	Université Paris Diderot, France
Franck van Breugel	York University, Toronto, Canada
Roberto Bruni	Università di Pisa, Italy
Rocco De Nicola	University of Florence, Italy
Dino Distefano	Queen Mary, University of London, and Monoidics Ltd., UK
Javier Esparza	Technische Universität München, Germany
Yuxi Fu	Shanghai Jiaotong University, China
Paul Gastin	ENS Cachan, Paris, France
Keijo Heljanko	Aalto University, Espoo, Finland
Anna Ingólfsdóttir	Reykjavik University, Iceland
Joost-Pieter Katoen	RWTH Aachen University, Germany
Maciej Koutny	Newcastle University, UK
Antonín Kučera	Masaryk University, Brno, Czech Republic
Barbara König	University of Duisburg-Essen, Germany
Gerald Lüttgen	University of Bamberg, Germany
Bas Luttik	Eindhoven University of Technology, The Netherlands
Madhavan Mukund	Chennai Mathematical Institute, India
Ernst-Rüdiger Olderog	Carl-von-Ossietzky-Universität Oldenburg, Germany
Joel Ouaknine	Oxford University, UK
Jan Rutten	CWI Amsterdam and Radboud University Nijmegen, The Netherlands
Vladimiro Sassone	University of Southampton, UK
Mariëlle Stoelinga	University of Twente, The Netherlands

Irek Ulidowski	University of Leicester, UK
Björn Victor	Uppsala University, Sweden
Mahesh Viswanathan	University of Illinois, Urbana-Champaign, USA
Andrzej Wasowski	IT University of Copenhagen, Denmark

Organizing Committee

Henrik Bohnenkamp
Sander Bruggink (Workshop Co-chair)
Arnd Gehrmann
Mathias Hülsbusch (Workshop Co-chair)
Christina Jansen
Nils Jansen
Joost-Pieter Katoen (Co-chair)
Barbara König (Co-chair)
Ulrich Loup
Thomas Noll
Elke Ohlenforst
Sabrina von Styp

Additional Reviewers

Acciai, Lucia
Aceto, Luca
Alglave, Jade
Andova, Suzana
Asarin, Eugene
Atig, Mohamed Faouzi
Baeten, Jos
Bauer, Sebastian
Benes, Nikola
Berard, Beatrice
Bernardo, Marco
Biondi, Fabrizio
Bocchi, Laura
Bodei, Chiara
Bonakdarpour, Borzoo
Bonchi, Filippo
Boreale, Michele
Borgström, Johannes
Bouyer, Patricia
Brotherston, James
Buchholz, Peter
Cai, Xiaojuan
Caires, Luis
Calzavara, Stefano
Capecchi, Sara
Chadha, Rohit
Chaloupka, Jakub
Chatterjee, Krishnendu
Chen, Yijia
Ciesinski, Frank
Cimini, Matteo
Crafa, Silvia
Cuijpers, Pieter
Cyriac, Aiswarya
D'Argenio, Pedro R.
Darondeau, Philippe
Degano, Pierpaolo
Delahaye, Benoit
Demri, Stephane
Deng, Yuxin
Denielou, Pierre-Malo
Dodds, Mike
Doyen, Laurent
Dräger, Klaus
Dubrovin, Jori
Dubslaff, Clemens
Emmi, Michael
Enea, Constantin
Finkbeiner, Bernd
Fokkink, Wan
Forejt, Vojtech
Fournet, Cedric
Francalanza, Adrian
Gadducci, Fabio
Galloway, Andy
Galpin, Vashti
Gan, Xiang
Garg, Pranav
Gay, Simon
Geeraerts, Gilles
Ghelli, Giorgio
Giunti, Marco
van Glabbeek, Rob
Godskesen, Jens Chr.
Göller, Stefan
Gorla, Daniele
Gorogiannis, Nikos
Gotsman, Alexey
Grigore, Radu

Hansen, Helle Hvid
Hasuo, Ichiro
Hilscher, Martin
Hoenicke, Jochen
Jansen, David N.
Jobstmann, Barbara
Johansson, Magnus
Junttila, Tommi
Jurdzinski, Marcin
Khomenko, Victor
Kindermann, Roland
Kleijn, Jetty
Klein, Joachim
Klin, Bartek
Klüppelholz, Sascha
Krause, Christian
Kremer, Steve
Kretinsky, Jan
Kufleitner, Manfred
Lanese, Ivan
Lange, Martin
Langerak, Rom
Launiainen, Tuomas
Lazic, Ranko
Linker, Sven
Loreti, Michele
Lozes, Etienne
Lu, Pinyan
Maranget, Luc
Markey, Nicolas
Markovski, Jasen
Mayr, Richard
Mehnert, Hannes
Melgratti, Hernan
Mereacre, Alexandru
Merro, Massimo
Merz, Stephan
Meyer, Roland
Miculan, Marino
Milius, Stefan
Mokhov, Andrey
Montanari, Ugo
Montesi, Fabrizio
Moreaux, Patrice
Morgan, Carroll
Mousavi,
Mohammadreza
Murawski, Andrzej
Narayan Kumar, K.
Nestmann, Uwe
Nielson, Hanne Riis
Novotny, Petr
Nyman, Ulrik
Obdrzalek, Jan
Ochmanski, Edward
Owens, Scott
Parrow, Joachim
Paulevé, Loïc
Petreczky, Mihaly
Philippou, Anna
Phillips, Iain
Picaronny, Claudine
Piterman, Nir
Pommereau, Franck
Prabhakar, Pavithra
Quesel, Jan-David
Rafnsson, Willard
Rehak, Vojtech
Rensink, Arend
Reynier, Pierre-Alain
Ridge, Tom
Rinetzky, Noam
Ruppert, Eric
Sack, Joshua
Salaün, Gwen
Sangnier, Arnaud
Sawa, Zdenek
Schmidt, Klaus
Schnoebelen, Philippe
Schwentick, Thomas
Schwoon, Stefan
Sidorova, Natalia
Sighireanu, Mihaela
Smith, Michael
Sobocinski, Pawel
Sokolova, Ana
Song, Lei
Srba, Jiri
Swaminathan, Mani
Tautschnig, Michael
Thistle, John
Tiga, Siegbert
van Tilburg, Paul
Timmer, Mark
Trancón Y Widemann,
Baltasar
Traonouez, Louis-Marie
Tripakis, Stavros
Villard, Jules
Viswanathan, Ramesh
Vogler, Walter
Wahl, Thomas
Wieringa, Siert
Willemse, Tim
Wimmel, Harro
Wonisch, Daniel
Worrell, James
Wrigstad, Tobias
Xu, Xian
Yang, Mu
Zappa Nardelli,
Francesco
Zavattaro, Gianluigi

Table of Contents

Invited Talks

Real-Time Systems

Probabilistic Systems

Automata

Separation Logic

π-Calculus

Petri Nets

Process Algebra and Modeling

Verification

Games

Bisimulation

Carrying Probabilities to the Infinite World[⋆]

Parosh Aziz Abdulla

Uppsala University, Sweden

Abstract. We give en example-guided introduction to a framework that we have developed in recent years in order to extend the applicability of program verification to the context of systems modeled as infinite-state Markov chains. In particular, we describe the class of *decisive Markov chains*, and show how to perform qualitative and quantitative analysis of Markov chains that arise from probabilistic extensions of classical models such as Petri nets and communicating finite-state processes.

1 Introduction

In recent years, several approaches have been proposed for automatic verification of infinite-state systems (see e.g., [2, 1]). In a parallel development, there has been an extensive research effort for the design and analysis of models with stochastic behaviors (e.g., [12, 7, 6, 11]). Recently, several works have considered verification of infinite-state Markov chains that are generated by push-down systems (e.g., [9, 10]). We consider verification of Markov chains with infinite state spaces. We describe a general framework that can handle probabilistic versions of several classical models such as Petri nets and communicating finite-state processes. We do that by defining abstract conditions on infinite Markov chains that give rise to the class of *decisive Markov chains*. For this class, we perform qualitative and quantitative analysis wrt. standard properties such as reachability and repeated reachability of a given set of configurations. This presentation is informal and example-based. For the technical details, we refer to our works in [3–5].

2 Transition Systems

A transition system $\mathcal{T}$ is a pair $(C, \longrightarrow)$ where C is a (potentially infinite) set of *configurations*, and $\longrightarrow \subseteq C \times C$ is the *transition relation*. As usual, we write $c \longrightarrow c'$ to denote that $(c, c') \in \longrightarrow$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For a configuration c, a *c-run* is a sequence $c_0 \longrightarrow c_1 \longrightarrow c_2 \longrightarrow \cdots$ where $c_0 = c$. For a natural number k, we write $c \xrightarrow{k} c'$ if there is a sequence $c_0 \longrightarrow c_1 \longrightarrow \cdots \longrightarrow c_\ell$ with $\ell \leq k$, $c_0 = c$ and $c_\ell = c'$, i.e., we can reach c' from c in k or fewer steps. Notice that $c \xrightarrow{*} c'$ iff $c \xrightarrow{k} c'$ for some k. We lift the above notation to sets of configurations. For sets $C_1, C_2 \subseteq C$ of configurations,

[⋆] This tutorial is based on common work with Noomene Ben Henda, Richard Mayr, and Sven Sandberg.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 1–16, 2011.

we write $C_1 \longrightarrow C_2$ if $c \longrightarrow c'$ for some $c \in C_1$ and $c' \in C_2$. We use $C_1 \xrightarrow{k} C_2$ and $C_1 \xrightarrow{*} C_2$ in a similar way. We also mix the notations, so we we write for instance $c \xrightarrow{*} C_2$ instead of $\{c\} \xrightarrow{*} C_2$ We say that C_2 is *reachable* from C_1 if $C_1 \xrightarrow{*} C_2$. A transition system $\mathcal{T}$ is said to be k*-spanning* wrt. a give set F of configurations if for any configuration c, we have that $c \xrightarrow{*} F$ implies $c \xrightarrow{k} F$. In other words, for any configuration c, either c cannot reach F or it can reach F within k steps. We say that $\mathcal{T}$ is *finitely spanning* wrt. F if there is a k such that $\mathcal{T}$ is k-spanning wrt. F. In other words, if $\mathcal{T}$ is *finitely spanning* wrt. F then $\exists k.\ \forall c \in C.\ c \xrightarrow{*} F \supset c \xrightarrow{k} F$. We define $\widetilde{F} := \left\{c |\ c \not\xrightarrow{*} F\right\}$, i.e., $\widetilde{F}$ is the set of configurations from which F is not reachable. For a set $U \subseteq C$, we define $\mathit{Pre}(U) := \{c |\ \exists c' \in U.\ c \longrightarrow c'\}$, i.e., $\mathit{Pre}(U)$ is the set of configurations that can reach U through the execution of a single transition. We assume familiarity with the temporal logic CTL^*. Given a CTL^* path-formula ϕ, we use $(c \models \phi)$ to denote the set of c-runs that satisfy ϕ.

3 Petri Nets

We illustrate some ideas of our methodology, using the model of *Petri Nets*. After recalling the standard definitions of Petri nets, we describe the transition system induced by a Petri net. We describe how checking safety properties can be translated to the reachability of sets of configurations which are upward closed wrt. a natural ordering on the set of configurations[1]. We give a sketch of an algorithm to solve the reachability problem, and show that Petri nets are finitely spanning with respect to upward closed sets of configurations. Finally, we briefly mention a model closely related to Petri nets, namely that of Vector Addition Systems with States (VASS).

3.1 Model

A Petri net $\mathcal{N}$ is a tuple (P, T, F), where P is a finite set of *places*, T is a finite set of *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. If $(p, t) \in F$ then p is said to be an *input* place of t; and if $(t, p) \in F$ then p is said to be an *output* place of t. We use $\mathit{In}(t) := \{p |\ (p, t) \in F\}$ and $\mathit{Out}(t) := \{p |\ (t, p) \in F\}$ to denote the sets of input places and output places of t respectively.

Figure 1 shows an example of a Petri net with three places (drawn as circles), namely $\mathtt{L}$, $\mathtt{W}$, and $\mathtt{C}$; and two transitions (drawn as rectangles), namely t_1 and t_2. The flow relation is represented by edges from places to transitions, and from transitions to places. For instance, the flow relation in the example includes the pairs $(\mathtt{L}, t_1)$ and $(t_2, \mathtt{W})$, i.e., $\mathtt{L}$ is an input place of t_1, and $\mathtt{W}$ is an output place of t_2.

The transition system induced by a Petri net is defined by the set *configurations* together with the *transition relation* defined on them. A *configuration* c

[1] Reachability of upward closed sets of configurations is referred to as the *coverability problem* in the Petri net literature.

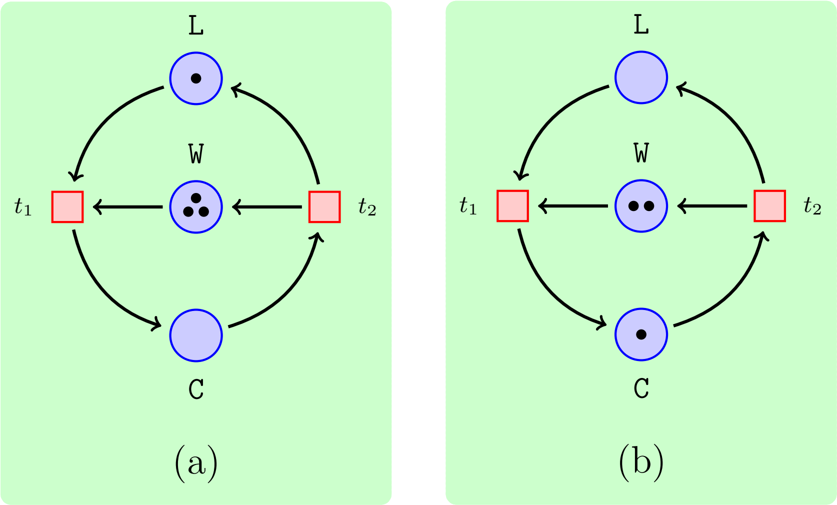


Fig. 1. (a) A simple Petri net (b) The result of firing t_1

of a Petri net[2] is a multiset over P. The configuration c defines the number of *tokens* in each place. Figure 1 (a) shows a configuration where there is one token in place L, three tokens in place W, and no token in place C. The configuration corresponds to the multiset $\left[\mathtt{L}, \mathtt{W}^3\right]$.

The operational semantics of a Petri net is defined through the notion of *firing* transitions. This gives a transition relation on the set of configurations. More precisely, when a transition t is fired, then a token is removed from each input place, and a token is added to each output place of t. The transition is fired only if each input place has at least one token. Formally, we write $c_1 \longrightarrow c_2$ to denote that there is a transition $t \in T$ such that $c_1 \geq \mathit{In}(t)$ and $c_2 = c_1 - \mathit{In}(t) + \mathit{Out}(t)$.

A set $U \subseteq C$ of configurations is said to be *upward closed* if $c \in U$ and $c \leq c'$ implies that $c' \in U$. For a configuration $c \in C$, define the *upward closure* of c by $\widehat{c} := \{c' |\ c \leq c'\}$. We extend the definition to a set $C_1 \subseteq C$ of configurations by $\widehat{C_1} := \cup_{c \in C_1} \widehat{c}$.

The Petri net of Figure 1 can be seen as a model of a simple mutual exclusion protocol, where access to the critical section is controlled by a global lock. A process is either *waiting* or is in its *critical section*. Initially, all the processes are in their waiting states. When a process wants to access the critical section, it must first acquire the lock. This can be done only if no other process has already acquired the lock. From the critical section, the process eventually releases the lock and moves back to the waiting state. The numbers of tokens in places W and C represent the number of processes in their waiting states and critical sections respectively. Absence of tokens in L means that the lock is currently taken by some process.

[2] A configuration in a Petri net is often called a *marking* in the literature.

The set C_{init} of *initial configurations* are those of the form $[\mathtt{L}, \mathtt{W}^n]$ where $n \geq 0$. In other words, all the processes are initially in their waiting states, and the lock is free. The transition t_1 models a process moving to its critical section, while the transition t_2 models a process going back to its waiting state. For instance, if we start from the configuration $\left[\mathtt{L}, \mathtt{W}^4\right]$, we can fire the transition t_1 to obtain the configuration $\left[\mathtt{C}, \mathtt{W}^3\right]$ from which we can fire the transition t_2 to obtain the configuration $\left[\mathtt{L}, \mathtt{W}^4\right]$, and so on.

3.2 Safety Properties

We are interested in checking a safety property for the Petri net in Figure 1. In a safety property, we want to show that "nothing bad happens" during the execution of the system. Typically, we define a set *Bad* of configurations, i.e., configurations which we do not want to occur during the execution of the system. In this particular example, we are interested in proving mutual exclusion. The set *Bad* contains those configurations that violate mutual exclusion, i.e., configurations in which at least two processes are in their critical sections. These configurations are of the form $\left[\mathtt{L}^k, \mathtt{W}^m, C^n\right]$ where $n \geq 2$. Checking the safety property can be carried out by checking whether we can fire a sequence of transitions taking us from an initial configuration to a bad configuration, i.e., we check whether the set *Bad* is reachable.

To analyze safety properties, we study some aspects of the behavior of Petri nets. First, we observe that the behavior of a Petri net is *monotonic*: if $c_1 \longrightarrow c_2$ and $c_1 \leq c_3$ then there is a c_4 such that $c_3 \longrightarrow c_4$ and $c_4 \geq c_2$.

We will work with sets of configurations that are upward closed with respect to $\leq$. Such sets are interesting in our setting for two reasons. First, all sets of bad configurations that occur in our examples are upward closed. For instance, in our example, whenever a configuration contains two processes in their critical sections then any larger configuration will also contain (at least) two processes in their critical sections, so the set *Bad* is upward closed. In this manner, checking the safety property amounts to deciding reachability of an upward closed set. Second, each upward closed set U can be uniquely represented by its set of minimal elements. This set, which we refer to as the set of *generator* of U, is finite due to Dickson's lemma [8]. In fact, since the ordering $\leq$ is anti-symmetric, it follows that each upward closed set has a unique generator. Finally, we observe that, due to monotonicity, if U is upward closed then $Pre(U)$ is upward closed. In other words, upward closedness is preserved by going backwards in the transition relation.

Below, we give a sketch of backward reachability algorithm for checking safety properties.

3.3 Algorithm

As mentioned above, we are interested in checking whether it is the case that *Bad* is reachable. The safety property is violated iff the question has a positive

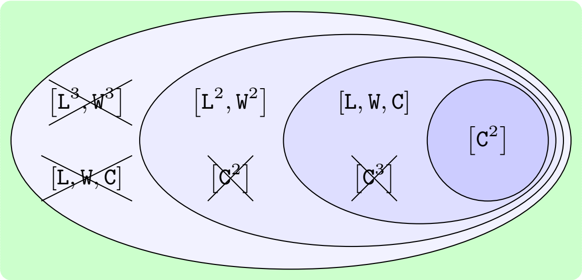


Fig. 2. Running the backward reachability algorithm on the example Petri net. Each ellipse contains the configurations generated during one iteration. The subsumed configurations are crossed over.

answer. The algorithm, illustrated in Figure 2, starts from the set of bad configurations, and tries to find a path backwards through the transition relation to the set of initial configurations. The algorithm operates on upward closed sets of configurations. An upward closed set is symbolically represented by a finite set of configurations, namely the members of its generator. In the above example, the set $gen\,(Bad)$ is the singleton $\{[\mathtt{C}^2]\}$. Therefore, the algorithm starts from the configuration $c_0 = [\mathtt{C}^2]$. From the configuration c_0, we go backwards and derive the generator of the set of configurations from which we can fire a transition and reach a configuration in $Bad = \widehat{c_0}$. Transition t_1 gives the configuration $c_1 = [\mathtt{L}, \mathtt{W}, \mathtt{C}]$, since $\widehat{c_1}$ contains exactly those configurations from which we can fire t_1 and reach a configuration in $\widehat{c_0}$. Analogously, transition t_2 gives the configuration $c_2 = [\mathtt{C}^3]$, since $\widehat{c_2}$ contains exactly those configurations from which we can fire t_2 and reach a configuration in $\widehat{c_0}$. Since $c_0 \leq c_2$, it follows that $\widehat{c_2} \subseteq \widehat{c_0}$. In such a case, we say that c_2 is *subsumed* by c_0. Since $\widehat{c_2} \subseteq \widehat{c_0}$, we can discard c_2 safely from the analysis without the loss of any information. Now, we repeat the procedure on c_1, and obtain the configurations $c_3 = [\mathtt{L}^2, \mathtt{W}^2]$ (via t_1), and $c_4 = [\mathtt{C}^2]$ (via t_2), where c_4 is subsumed by c_0. Finally, from c_3 we obtain the configurations $c_5 = [\mathtt{L}^3, \mathtt{W}^3]$ (via t_1), and $c_6 = [\mathtt{L}, \mathtt{W}, \mathtt{C}]$ (via t_2). The configurations c_5 and c_6 are subsumed by c_3 and c_1 respectively. The iteration terminates at this point since all the newly generated configurations were subsumed by existing ones, and hence there are no more new configurations to consider. In fact, the set $B = \{[\mathtt{C}^2], [\mathtt{L}, \mathtt{W}, \mathtt{C}], [\mathtt{L}^2, \mathtt{W}^2]\}$ is the generator of the set of configurations from which we can reach a bad configurations. The three members in B are those configurations which are not discarded in the analysis (they were not subsumed by other configurations). To check whether Bad is reachable, we check the intersection $\widehat{B} \cap C_{init}$. Since the intersection is empty, we conclude that Bad is not reachable, and hence the safety property is satisfied by the system.

3.4 Finite Span

An interesting consequence of the above algorithm is that Petri nets are finitely spanning with respect to an upward closed set of configurations. First, the reachability algorithm is guaranteed to terminate by Dickson's lemma [8]. Suppose that the algorithm starts by an upward closed set U (represented by its generator) and suppose that it terminates in k steps. Then, we claim that the Petri net is k-spanning wrt. U. To see this, consider a configuration c such that $c \xrightarrow{*} U$. Then, the algorithm will generate some c' such that $c' \leq c$. Suppose that c' is generated in step $\ell \leq k$. Then $c' \xrightarrow{\ell} U$. By monotonicity, we have that $c \xrightarrow{\ell} U$. For instance, the span of the Petri net of Figure 1 wrt. $\left[\mathtt{C}^2\right]$ is equal to 2.

3.5 VASS

A VASS is simply a Petri net equipped with a finite set of control states. Each transition has exactly one input control state and one output control state. Thus a transitions changes the control state of the VASS (in addition to changing the numbers of the tokens in the places). We can also think of a VASS as a counter machine where the counters are allowed to be tested for equality with zero.

4 Markov Chains

A *Markov chain* $\mathcal{M}$ is a tuple (C, P) where C is a (potentially infinite) set of *configuration*, and $P : C \times C \rightarrow [0, 1]$, such that $\sum_{c' \in C} P(c, c') = 1$, for each $c \in C$. A Markov chain induces a transition system, where the transition relation consists of pairs of configurations related by positive probabilities. In this manner, concepts defined for transition systems can be lifted to Markov chains. For instance, for a Markov chain $\mathcal{M}$, a run of $\mathcal{M}$ is a run in the underlying transition system, and $\mathcal{M}$ is *finitely spanning* w.r.t. given set F if the underlying transition system is finitely spanning w.r.t. F, etc.

We use $Prob_c(\phi)$ to denote the measure of the set of c-runs $(c \models \phi)$ (which is measurable by [12]). Sometimes, we refer to $Prob_c(\phi)$ as the probability by which ϕ holds at c. For instance, given a set $F \subseteq C$, $Prob_c(\Diamond F)$ is the measure of c-runs which eventually reach F. We say that *almost all* runs of a Markov chain satisfy a given property ϕ if $Prob_c(\phi) = 1$. In this case one says that $(c \models \phi)$ holds *almost certainly*. For formulas ϕ_1, ϕ_2, we use $Prob_c(\phi_1 \mid \phi_2)$ to denote the probability that ϕ_2 holds under the assumption that ϕ_1 holds.

5 Decisive Markov Chains

In this section, we introduce *decisive Markov chains*, present two sufficient conditions for decisiveness, and show examples of models that induce decisive Markov chains.

Consider a Markov chain $\mathcal{M} = (C, P)$ and a set F of configurations. We say that $\mathcal{M}$ is *decisive* wrt. F if each run of the system almost certainly will eventually either reach F or reach $\widetilde{F}$. Formally, for each configuration c, it is the case that $Prob_c\left(\Diamond F \vee \Diamond \widetilde{F}\right) = 1$. Put differently, if F is always reachable along a run ρ then ρ will almost certainly eventually reach F, i.e., $Prob_c(\Diamond F \mid \Box\exists\Diamond F) = 1$. Figure 3 shows an illustration of a run in a decisive Markov chain.

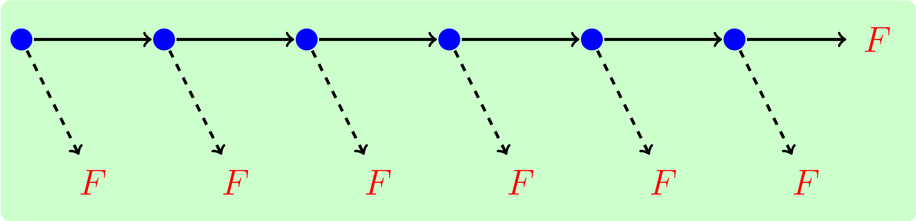


Fig. 3. Illustration of a run in a decisive Markov chain

Notice that all finite Markov chains are decisive (wrt. any given set of configurations). On the other hand, Figures 4–5 show examples of infinite Markov chains that are not decisive. Let us consider the Markov chain of Figure 4. The configuration F is reachable from each configuration in the Markov chain. Therefore $\widetilde{F} = \emptyset$, and hence $Prob_{Init}\left(\Diamond F \vee \Diamond \widetilde{F}\right) = Prob_{Init}(\Diamond F) = \frac{2}{3} < 1$.

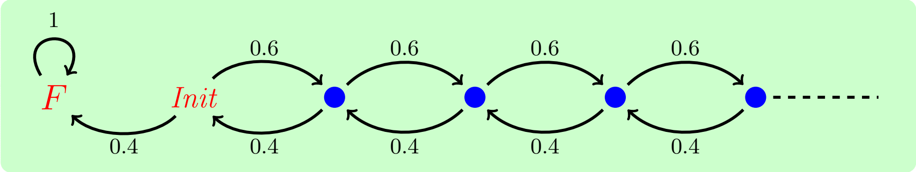


Fig. 4. A Markov chain that is not decisive

Next, let us consider the Markov chain of Figure 5. Again, the configuration F is reachable from each configuration in the Markov chain. Therefore, $Prob_{Init}\left(\Diamond F \vee \Diamond \widetilde{F}\right) = Prob_{Init}(\Diamond F) < 0.2$.

5.1 Sufficient Condition I

A configuration c is said to be of *coarseness* β if for each $c' \in C$, $P(c, c') > 0$ implies $P(c, c') \geq \beta$. A Markov chain $\mathcal{M} = (C, P)$ is said to be of *coarseness* β if each $c \in C$ is of coarseness β. We say that $\mathcal{M}$ is *coarse* if $\mathcal{M}$ is of coarseness β, for some $\beta > 0$. Notice that if $\mathcal{M}$ is coarse then the underlying transition system is finitely branching; however, the converse is not necessarily true. For instance, the Markov chain of Figure 4 is coarse (it is of coarseness 0.4), while the Markov chain of Figure 5 is not coarse.

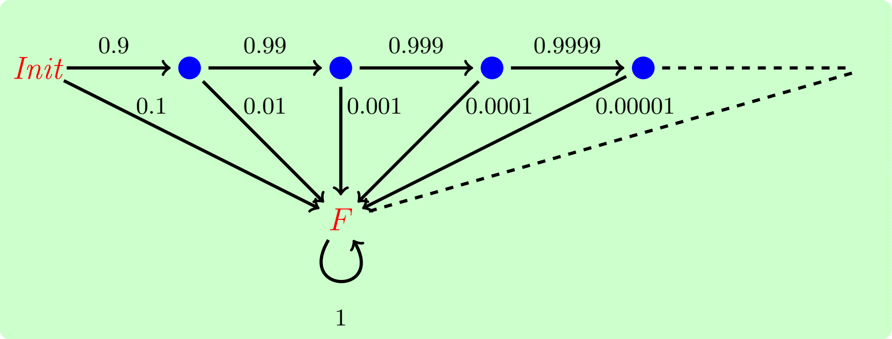


Fig. 5. Another Markov chain that is not decisive

A sufficient condition for decisiveness is the combination of coarseness and finite spanning. If a Markov chain $\mathcal{M}$ is both coarse and finitely spanning wrt. to set F of configurations then $\mathcal{M}$ is decisive wrt. F. The situation is illustrated in Figure 6 that shows a run in a Markov chain with coarseness 0.1 and span 3 (wrt. some F). The idea is that if we have a run ρ from which F is always reachable, then from each configuration along the run, the probability of hitting F within the next 3 steps is at least 0.001. Therefore, the probability that ρ will avoid F forever is equal to 0. In other words, ρ will almost certainly eventually reach F.

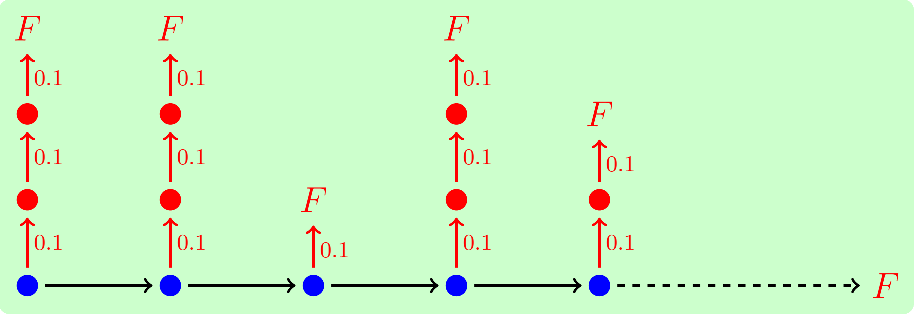


Fig. 6. A run in a Markov chain that is both coarse and finitely spanning

5.2 Sufficient Condition II

An *attractor* $A \subseteq C$ is a set of configurations, such that each run of $\mathcal{M}$ will almost certainly eventually reach A. Figure 7 illustrates an attractor. Formally, for each $c \in C$, we have $Prob_c(\Diamond A) = 1$, i.e., the set A is reached from c with probability one.

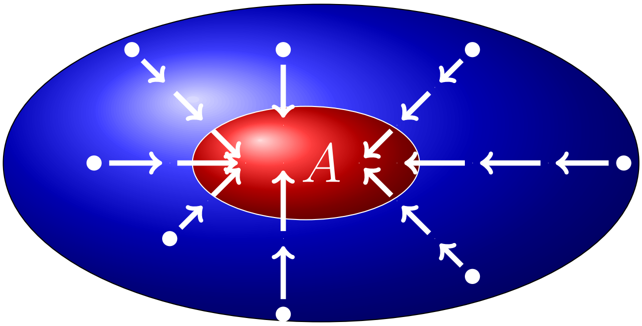


Fig. 7. An attractor

In fact, any run of the system will almost certainly visit A infinitely often. The reason (illustrated in Figure 8) is the following. Consider a run ρ. By definition of an attractor, ρ will almost certainly eventually reach a configuration $c_1 \in A$. We apply the definition of an attractor to the continuation of ρ from c_1. This continuation will almost certainly eventually reach a configuration $c_2 \in A$. The reasoning can be repeated infinitely thus obtaining an infinite sequence $c_1, c_2, \ldots$ of configurations inside A that will be visited. This means that A will be visited infinitely often with probability 1.

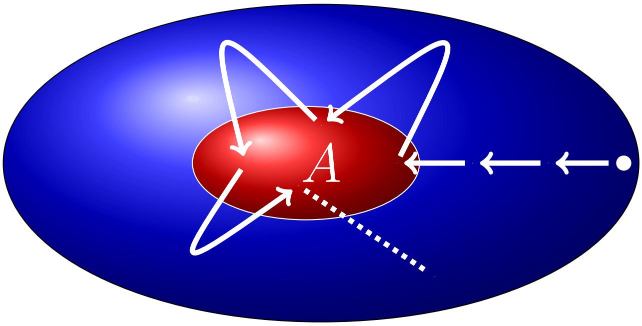


Fig. 8. Repeated reachability of an attractor

The existence of a finite attractor is a sufficient condition for decisiveness wrt. any set F of configurations. Assume a finite attractor A (see Figure 9). We partition A into two sets: $A_0 := A \cap \widetilde{F}$ and $A_1 := A \cap \neg\widetilde{F}$. In other words, the configurations in A_0 cannot reach F (in the underlying transition system), while from each configuration in A_1 there is a path to F. Consider a run ρ. We show that ρ will almost certainly eventually either reach $\widetilde{F}$ or reach F. We know that ρ will almost certainly visit A infinitely often. If ρ reaches F at some point then we are done. Otherwise, ρ will visit A_1 infinitely often with probability 1. Since

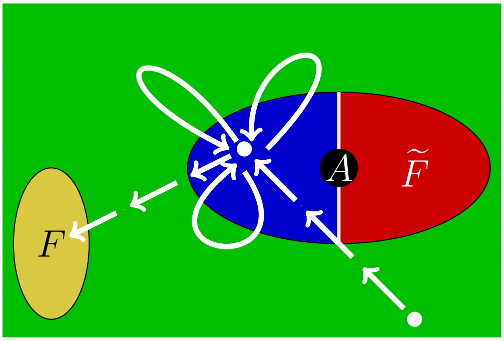


Fig. 9. Decisiveness due to a finite attractor

A_1 is a finite set, with probability 1, there is a configuration $c \in A_1$ that will be visited by ρ. By definition, we know that F is reachable from c, i.e., there is path (say of length k) from c to F. Let β be the probability that this path is taken during the next k steps of the run. This means that each time ρ visits c, it will reach F during the next k steps with probability at least β, which implies that ρ cannot avoid F forever. Thus ρ will almost certainly eventually reach F.

5.3 Probabilistic Petri Nets

To induce Markov chains from Petri nets, we associate weights (natural numbers) with the transitions of the net. If several transitions are enabled from a given configuration then a particular transition will be fired with a probability that reflects its weight relative to the weights of the rest of the enabled transitions.

For instance, Figure 10 shows a weighted version of the Petri net of Figure 1, where the transitions t_1 and t_2 have weights that are 3 and 2 respectively. Consider the configuration $c_1 = \left[\mathtt{L}, \mathtt{W}^3\right]$. From c_1 only transition t_1 is enabled. Therefore the probability of moving from c_1 to the configuration $c_2 = \left[\mathtt{C}, \mathtt{W}^2\right]$ is given by $P(c_1, c_2) = 1$, while $P(c_1, c_1') = 0$ for all other configurations c_1'. On the other hand, both t_1 and t_2 are enabled from the configuration $c_3 = \left[\mathtt{L}, \mathtt{C}, \mathtt{W}^3\right]$. Therefore, the probability of moving from c_3 to the configuration $c_4 = \left[\mathtt{C}^2, \mathtt{W}^2\right]$ is given by $P(c_3, c_4) = \frac{3}{2+3} = \frac{3}{5}$; while, for $c_5 = \left[\mathtt{L}, \mathtt{C}, \mathtt{W}^3\right]$, we have $P(c_3, c_5) = \frac{2}{2+3} = \frac{2}{5}$. In such a manner, we obtain an infinite-state Markov chain, where the configurations are those of the Petri net, and the probability distribution is defined by the weights of the transitions as described above. On the one hand, this Markov chain is finitely spanning wrt. an upward closed set F of configurations, since the underlying transition system is finitely spanning wrt. to F (as explained in Section 3). On the other hand, the Markov chains is coarse. In fact, the Markov chain is at least of coarseness $\frac{1}{w}$ where w is the sum of weights of all the transitions in the Petri net. It follows that the Markov chain induced by a Petri net is decisive wrt. any upward closed set F.

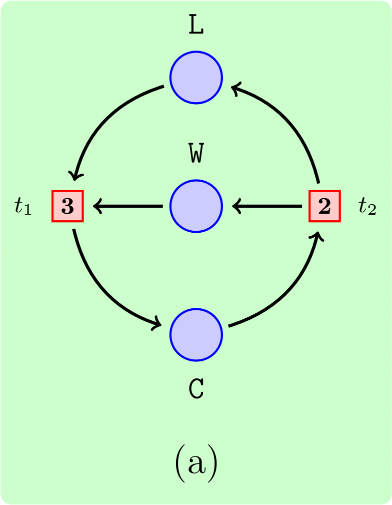


Fig. 10. A weighted Petri net

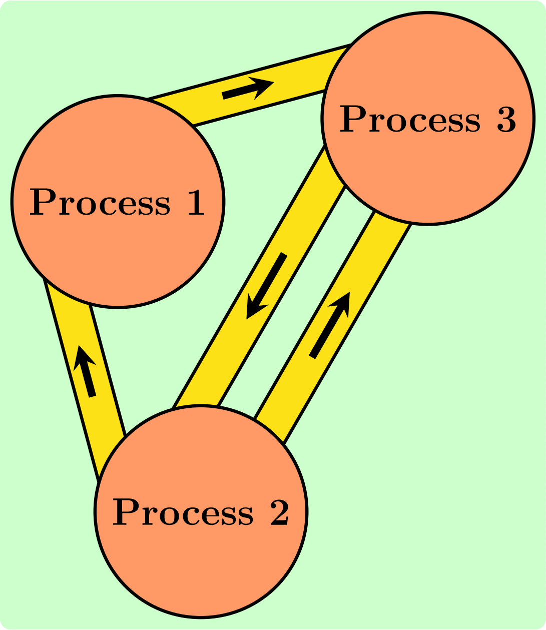


Fig. 11. Communicating finite-state processes

5.4 Communicating Processes

We consider systems that consist of finite sets of finite-state processes, communicating through unbounded channels that behave as FIFO queues (see Figure 11).

During each step in a run of the system, a process may either send a message to a channel (in which case the message is appended to the tail of the channel), or receive a message from a channel (in which case the message is fetched from the head of the channel). Choices between different enabled transitions are resolved probabilistically by associating weights to the transition in a similar manner to the case of Petri nets. Furthermore, after each step in a run of the system, a given message may be *lost* (removed from the buffer) by a predefined probability λ. The probability of loss is identical (equal to λ) for all messages that reside in the channels. The induced Markov chain is infinite-state, since the sizes of the buffers are not bounded. However, such a Markov chains always contains a finite attractor. In fact, the finite attractor is given by the set of configurations in which all the channels are empty. This is due to the fact that the more messages inside a buffer, the higher the probability that "many of" these message will be lost in the next step. Thus, each run of the system will almost certainly reach a configuration where all the channels are empty. Consequently, the induced Markov chain is decisive wrt. any set F of configurations.

6 Qualitative Analysis

In this section, we consider *qualitative analysis* of (infinite-state) Markov chains. We are given an *initial configuration Init* and a set of final (target) configurations F. In *qualitative reachability analysis*, we want to check whether $Prob_{Init}(\Diamond F) = 1$, i.e., whether the probability of reaching F from $Init$ is equal to 1. In *qualitative repeated reachability analysis*, we want to check whether $Prob_{Init}(\Box\Diamond F) = 1$, i.e., whether the probability of repeatedly reaching F from $Init$ is equal to 1. In the case of decisive Markov chains, we reduce the problems to corresponding problems defined on the underlying transition systems.

6.1 Reachability

For sets of configurations C_1, C_2 and a run $\rho = c_0 \longrightarrow c_1 \longrightarrow c_1 \longrightarrow \cdots$, we say that ρ satisfies C_1 $\underline{Before}$ C_2 if there is an $i \geq 0$ such that $c_i \in C_1$ and for all $j : 0 \leq j \leq i$ it is the case that $c_j \notin C_2$. Then, for a configuration c, we have $c \models \exists.\ C_1\ \underline{Before}\ C_2$ iff there is a c-run that reaches C_1 before reaching C_2. We will show that $Prob_{Init}(\Diamond F) = 1$ iff $Init \not\models \exists.\ \widetilde{F}\ \underline{Before}\ F$. One direction of the proof is illustrated in Figure 12. In fact, this direction holds for any Markov chain and is not dependent on the Markov chain being decisive.

Suppose that $Init \models \exists.\ \widetilde{F}\ \underline{Before}\ F$, i.e., there is an $Init$-run that reaches $\widetilde{F}$ before reaching F. This means that there is a prefix ρ' of ρ that will reach $\widetilde{F}$ before reaching F. The prefix ρ' has a certain probability β. Notice that the measure of all runs that are continuations of ρ' (that have ρ' as a prefix) is equal to β. Furthermore, all these continuations will not reach F (since ρ' visits $\widetilde{F}$ from which F is not reachable). This means that the measure of computations that will never reach F is larger than β, and hence the measure of computations that will reach F is smaller than $1 - \beta < 1$.

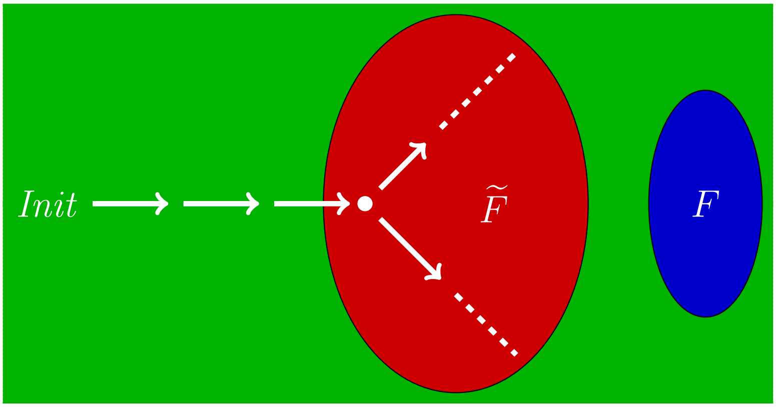


Fig. 12. Reaching $\widetilde{F}$ before F

The other direction of the proof, namely that $\mathit{Init} \not\models \exists.\ \widetilde{F}\ \underline{\mathit{Before}}\ F$ implies $\mathit{Prob}_{\mathit{Init}}(\Diamond F) = 1$ does not hold in general. As a counter-example, consider the Markov chain of Figure 4. In this example $\widetilde{F} = \emptyset$ since F is reachable from every configuration in the Markov chain. Therefore, $\mathit{Init} \not\models \exists.\ \widetilde{F}\ \underline{\mathit{Before}}\ F$ holds trivially. However, as mentioned in Section 5, we have $\mathit{Prob}_{\mathit{Init}}(\Diamond F) = \frac{2}{3} < 1$.

This direction of the proof holds for Markov chain that is decisive wrt. F (Figure 13). Consider any *Init*-run ρ. By decisiveness, ρ will almost certainly reach either F or $\widetilde{F}$. In the first case, the claim holds trivially. In the second case, since ρ visits $\widetilde{F}$ and since all runs must visit F before visiting $\widetilde{F}$, we know that ρ must have visited F (before visiting $\widetilde{F}$).

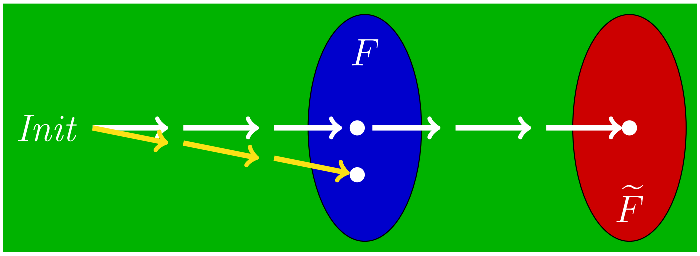


Fig. 13. Reaching F before $\widetilde{F}$

Thus, we have reduced the problem of checking whether $\mathit{Prob}_{\mathit{Init}}(\Diamond F) = 1$ in a Markov chain that is decisive wrt. F to that of checking whether $\mathit{Init} \models \exists.\ \widetilde{F}\ \underline{\mathit{Before}}\ F$ in the underlying transition system. In fact, as clear from the structure of the proof, the two problems are equivalent.

The problem of checking $\mathit{Init} \models \exists.\ \widetilde{F}\ \underline{\mathit{Before}}\ F$ is decidable for Petri nets in case F is given by a set of control states (we are asking about the reachability of a set of control states in a VASS). However, the problem is undecidable in case F is given by an arbitrary upward closed set of configurations. This is quite surprising since the control state reachability problem and upward closed set reachability problem are equivalent for all other models (whether probabilistic or not). The problem is also decidable for lossy channel systems which is the underlying transition system model for communicating processes (as described in Section 5).

6.2 Repeated Reachability

We will show that, for a configuration Init and a set F of configurations in a decisive Markov chain, we have that $\mathit{Prob}_{\mathit{Init}}(\Box\Diamond F) = 1$ iff $\mathit{Init} \models \forall\Box\ \exists\Diamond\ F$. The formula states that the set F remains reachable along all Init-runs. Notice that this is equivalent to $\mathit{Init} \not\models \exists\Diamond\ \widetilde{F}$. (it is not the case that there is an Init-run that leads to $\widetilde{F}$). Also, in this case, one direction of the proof (illustrated in Figure 12) holds for any Markov chain (it is not dependent on the Markov chain being decisive). Suppose that $\mathit{Init} \not\models \forall\Box\ \exists\Diamond\ F$ (i.e. $\mathit{Init} \models \exists\Diamond\ \widetilde{F}$). This means that there is a Init-run ρ that reaches $\widetilde{F}$. The run ρ is similar to the one shown in Figure 12 (with the difference that ρ is now allowed to visit F before visiting $\widetilde{F}$). In a similar manner to the case of reachability, there is a prefix ρ' that will reach $\widetilde{F}$ and that has a certain probability β. Furthermore, none of the continuations of ρ' will reach F. This means that the measure of computations that will not repeatedly visit F is smaller than $1 - \beta < 1$.

Also here, the other direction of the proof, namely that $\mathit{Init} \models \forall\Box\ \exists\Diamond\ F$. implies $\mathit{Prob}_{\mathit{Init}}(\Box\Diamond F) = 1$ does not hold in general. For instance in the Markov chain of Figure 4, we have $\widetilde{F} = \emptyset$. Therefore, $\mathit{Init} \models \forall\Box\ \exists\Diamond\ F$ holds. However, we have $\mathit{Prob}_{\mathit{Init}}(\Box\Diamond F) \leq \mathit{Prob}_{\mathit{Init}}(\Diamond F) = \frac{2}{3} < 1$. This direction of the proof holds for Markov chain that is decisive wrt. F (Figure 14). Consider any Init-run ρ. Since $\widetilde{F}$ is not reachable from Init, it follows by decisiveness that ρ will almost certainly reach some configuration $c_1 \in F$. We apply the definition of an attractor to the continuation of ρ from c_1. This continuation will almost certainly eventually reach a configuration $c_2 \in F$. The reasoning can be repeated infinitely thus obtaining an infinite sequence $c_1, c_2, \ldots$ of configurations inside the F that will be visited. This means that F will be visited infinitely often with probability 1.

We have then reduced the problem of checking whether $\mathit{Prob}_{\mathit{Init}}(\Box\Diamond F) = 1$ in a Markov chain that is decisive wrt. F to that of checking whether $\mathit{Init} \models \forall\Box\ \exists\Diamond\ F$ in the underlying transition system.

The problem of checking $\mathit{Init} \models \forall\Box\ \exists\Diamond\ F$ is decidable for Petri nets in case F is an arbitrary upward closed set of configurations. This is again surprising since it means that repeated reachability is a simpler problem than simple reachability (as we have seen, the former is decidable for upward closed sets while the latter is undecidable). The problem is also decidable for lossy channel systems.

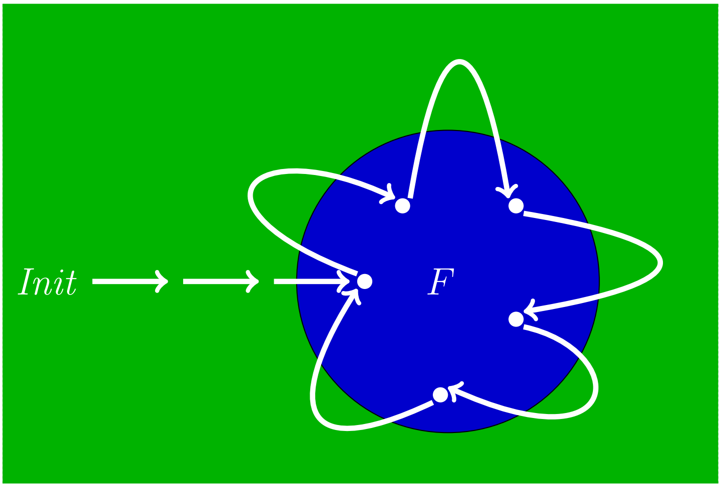


Fig. 14. Repeatedly reaching F

7 Quantitative Analysis

We give a sketch of a algorithm that computes the probability $Prob_{Init}(\Diamond F)$ up to an arbitrary given precision ϵ. The algorithm builds the reachability tree starting from *Init* as the root of the tree. It maintains two variables, namely the variable *yes* that under-approximates $Prob_{Init}(\Diamond F)$, and *no* that under-approximates $Prob_{Init}(\neg\Diamond F)$. Each time the algorithm picks a leaf from the tree (corresponding to a configuration c), it computes the successors of the node. For each successor c' such that $c' \notin F \cup \widetilde{F}$, it creates a child labeled with c' and labels the edge between the nodes by $P(c, c')$. If $c' \in F$ then it will close the node and increases the value *yes* by the weight the path from the root to the current node (equal to the product of the probabilities on the edges along the path). If $c' \in \widetilde{F}$ it will close the node and increases the value *no* analogously. The algorithm terminates when $yes + no \geq \epsilon$. The algorithm is guaranteed to terminate in case the Markov chain is decisive wrt. F since, as more and more steps of the algorithm are executed, the sum $yes + no$ will get arbitrarily close to one.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bulletin of Symbolic Logic 16(4), 457–515 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proc. LICS 1996, 11th IEEE Int. Symp. on Logic in Computer Science, pp. 313–321 (1996)
3. Abdulla, P.A., Henda, N.B., Mayr, R.: Decisive markov chains. Logical Methods in Computer Science (2007); A Preliminary Version appeared in Proc. LICS 2005

4. Abdulla, P.A., Ben Henda, N., Mayr, R., Sandberg, S.: Eager markov chains. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 24–38. Springer, Heidelberg (2006)
5. Abdulla, P.A., Henda, N.B., Mayr, R., Sandberg, S.: Limiting behavior of Markov chains with eager attractors. In: D'Argentio, P.R., Milner, A., Rubino, G. (eds.) 3rd International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 253–262 (2006)
6. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time markov chains. IEEE Trans. Software Eng. 29(6), 524–541 (2003)
7. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
8. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Amer. J. Math. 35, 413–422 (1913)
9. Esparza, J., Kučera, A., Mayr, R.: Model checking probabilistic pushdown automata. In: Proc. LICS 2004, 20th IEEE Int. Symp. on Logic in Computer Science, pp. 12–21 (2004)
10. Etessami, K., Yannakakis, M.: Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 340–352. Springer, Heidelberg (2005)
11. Huth, M., Kwiatkowska, M.: Quantitative analysis and model checking. In: Proc. LICS 1997, 12th IEEE Int. Symp. on Logic in Computer Science, pp. 111–122 (1997)
12. Vardi, M.: Automatic verification of probabilistic concurrent finite-state programs. In: Proc. FOCS 1985, 26th Annual Symp. Foundations of Computer Science, pp. 327–338 (1985)

Generalized Universality

Eli Gafni[1] and Rachid Guerraoui[2]

[1] UCLA
[2] EPFL

Abstract. *State machine replication* reduces distributed to centralized computing. Any sequential service, modeled by a state machine, can be replicated over any number of processes and made highly available to all of them. At the heart of this fundamental reduction lies the so called *universal consensus* abstraction, key to providing the illusion of single shared service, despite replication.

Yet, as universal as it may be, consensus is just one specific instance of a more general abstraction, *k-set consensus* where, instead of agreeing on a unique decision, the processes may diverge but at most k different decisions are reached. It is legitimate to ask whether the celebrated state machine replication construct has its analogue with $k > 1$. If it did not, one could question the aura of distributed computing deserving an underpinning Theory for 1, the unit of multiplication, would be special in a field, distributed computing, that does not arithmetically multiply.

This paper presents, two decades after k-set consensus was introduced, the generalization with $k > 1$ of state machine replication. We show that with k-set consensus, any number of processes can emulate k state machines of which at least one remains highly available. While doing so, we also generalize the very notion of consensus universality.

Keywords: State machine replication, k-set consensus, universality.

1 Introduction

One of the most fundamental constructs of distributed computing is the *replicated state machine* protocol [11]. It essentially makes a distributed system emulate a, highly available, centralized one using a *consensus* abstraction [6]. Making the approach efficient by allowing a system to run with little overhead while the system's components are well behaved, and nevertheless not let it commit to a mistake while experiencing faults, has been a trust of distributed computing [4].

How does state machine replication work? A computing service is modeled as a state machine that executes commands deterministically. Processes hold each a copy of this state machine, to which they issue commands. To provide the illusion of sharing a single state machine, the processes use consensus. Each instance of consensus is used to decide which command to execute next and hence make sure all commands are executed on the state machine copies in the same order: this,

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 17–27, 2011.

together with the very fact that the state machine is deterministic, implies that all its copies keep the same state. (It was later shown how the choice of which proposed commands to execute next can be made fair [8].) Consensus is said to be *universal* [8] in the sense that its availability implies the fair availability of any shared service.

Yet, as universal as it may be, consensus is just the specific case of a more general abstraction: *k-set consensus* [5], where the number of decisions that can be output by processes is more than 1 but at most k. That realization of Chaudhuri in 1990, the challenge she posed, whether k-set consensus is solvable in a system where the number of processes is larger than k, and her quote of Saks about "smelling like Sperner", has resulted, three years later, in the discovery of the connection between distributed computing and algebraic-topology [9,2,10].[1]

Given the importance of the state machine replication construct and the cornerstone role of consensus, it is natural to ask what form of construct we get if we generalize consensus to k-set consensus. Not being able to generalize state machine replication, and the actual universality of (1-set) consensus, to the case where $k > 1$, would be frustrating and would somehow reveal a hole in the theory of distributed computing.

We show in this paper that k-set consensus is, in a precise sense, *k-universal*: with k-set consensus, we can implement k state machines with the guarantee that at least one machine remains highly available to all processes. In other words, whereas consensus reduces distributed computing to centralized computing, i.e., "1-computing", k-set consensus is the generalization that reduces distributed computing to "k-computing".

"Practical" applications might be foreseen. Multiple state machines, implementing different services, one of which is guaranteed to progress, is better than one state machine that does not progress. This could be the case if consensus cannot be reached but 2-set consensus can: a shared memory system of 3 processes and 1 failure, or a system that provides some weak compare-swap primitive that allows for two winners. In fact, multiple state machines, even implementing the same service, may provide for an interesting alternative behavior to a classical state machine replication scheme at the time when the system is not stable. Instead of blocking like a single machine will do, in our case at least one machine will progress. There is of course the danger that the state machines diverge from each other but many applications can tolerate divergence of view for a while. When the system stabilizes, these divergent views may be reconciled to continue with what is effectively a single view of the system. k read-write processes proceeding wait-free.

The rest of the paper is organized as follows. We first recall below the basic state machine replication construct, then we present the properties of generalized state machine replication and finally our protocol.

[1] The connection has been symmetric so far: whatever one has proved using pure algorithmic implementation arguments had the analogue in algebraic-topology.

2 State Machine Replication: The Classics

2.1 Model

We assume here that processes can exchange information by reading and writing from shared memory cells, as well as agree on common decisions. More precisely, we assume a basic *read-write* shared memory model augmented with *consensus* objects [1].

Processes can be *correct*, in which case they execute an infinite number of steps of the algorithm assigned to them, or they *crash* and stop any activity. We consider an *asynchronous* system, meaning we make no assumption neither on process relative speeds nor on the time needed to access shared read-write memory cells or consensus objects.

The way consensus is used is simple: processes propose inputs and get back outputs such that the following properties are satisfied.

1. *Validity:* any output is the input of some process.
2. *Agreement:* all outputs are the same;
3. *Progress:* any correct process that proposes an input gets back an output;

2.2 Protocol

The algorithm underlying state machine replication is depicted in Figure 1. It is *round-based* as we will explain below.

Every process maintains locally a copy of the state machine as well as an ordered list of commands, denoted respectively *sm* and *comList* in Figure 1. The state machine is deterministic: the same command executed on different copies of the state machine in the same state, leads to copies that are also in the same state .

The processes typically have different lists of commands, say requests coming from different users of a web service modeled by the state machine. For the sake of presentation simplicity, we assume here that every process has an infinite such list of commands. A process picks one command at a time from its list; we also say that the process *issues* the command. As we will explain, the process does not issue the next command until it managed to execute the previous command on its state machine. Of course, the challenge for the protocol is to execute commands on the various copies of the state machine in the same order. This is where the consensus abstraction comes to play.

Consensus objects form a list, denoted by *ConsList* in Figure 1, and exactly one object of the list is used in each *round* of the protocol. Processes go round-by-round, incrementally, in each round proposing a command to the consensus object of the round and executing the command returned by that consensus object. Crucial to the correctness of the protocol lies the very fact that, in any given round, the consensus instance used by all the processes is the same shared object.

Basically, every process *p* proposes the next command it wants executed to the next consensus object. This, in turn, returns a command, not necessarily

```
local data structures:
sm                                                  (* a copy of the state machine *)
comList                                                    (* a list of commands *)
passed := true     (* determines if the process executed its previous command *)

shared data structure:
ConsList                                   (* a list of shared consensus objects *)

forever do:
if passed then com1 := comList.next()              (* pick the next command *)
cons := ConsList.next()                      (* pick the next consensus object *)
com2 := cons.propose(com1)                        (* agree on the next command *)
sm.execute(com2)                          (* execute the agreed upon command *)
if com2 = com1 then passed := true
else passed := false                             (* test if own command passed *)
```

Fig. 1. State machine replication

that proposed by p, but one proposed by at least some process. The command returned to a process p is then executed by p on its state machine: we simply say that p *executes* the command. To ensure that every process executes the commands in their original order, no process issues its next command unless it has executed its previous one.

2.3 Correctness

The correctness of the protocol of Figure 1 lies on four observations.

1. *Validity:* If a process q executes command c, then c was issued by some process p and q has executed every command issued by p before c. This follows from the facts that (a) a consensus object returns one of the inputs proposed (*validity* property of consensus), i.e., one of the commands issued by a process and (b) a process does not issue a new command unless it has executed the previous one it issued.
2. *Ordering:* If a process executes command c without having executed command c', then no process executes c' without having executed c. This follows from the facts that (a) the processes execute the commands output by the consensus objects, (b) the consensus objects are invoked by the processes in the same order and (c) each such object returns the same command to all processes (*agreement* property of consensus).
3. *Progress:* Every correct process executes an infinite number of commands on the state machine. This follows from the facts that (a) there is no wait statement in the algorithm and (b) every invocation to consensus by a correct process returns a command to that process (*progress* property of consensus).

It is important to notice at this point that this simple protocol does not guarantee fairness. Consensus objects could always return the commands proposed by the same process, i.e., there is no obligation for a consensus object to be fair with respect to the processes of which it selects the input. Fairness could however easily be ensured by having the processes help each other. Namely, when a process issues a command, it first writes it in shared memory before proposing it to consensus. Processes would now propose sets of commands (their own and those of others) to consensus; accordingly, a consensus would return a set; the set would be the same at all processes which would execute the commands in the same deterministic order. For presentation simplicity, we omit fairness and helping.

3 Generalized State Machine Replication

Basically, with consensus, a state machine can be replicated over any number of processes and made highly available to all those processes. This makes of consensus a *universal* abstraction for any sequential service can be modeled as a state machine accessed by any number of processes. In a sense, with consensus, computing on several distributed computers is reduced to computing on a single, highly available, one.

In the following, we generalize this idea to show that, with k-set consensus, we can replicate k state machines one of which at least one is highly available. (The one that remains highly available is unknown in advance, for otherwise this would boil down to classical state machine replication.) In some sense, we show that k-set consensus is *k-universal.* We first give below the model underlying general state machine replication, then we define the properties we seek it to ensure before diving into the details of our protocol.

3.1 Properties

Here, we assume k state machines replicated over all processes of the system. The processes have each at their disposal a list of k-vectors of commands that they *issue* and seek to *execute* on their local copies of the k state machines: a command issued at entry j of a vector is to be executed on machine $\text{sm}[j]$. Again, for presentation simplicity, we assume the lists of commands are infinite.

A generalized state machine replication protocol satisfies the following properties:

1. *Validity:* If a process q executes command c on state machine $\text{sm}[i]$, then c was issued by some process p at entry i (of p's command vector), and q has executed every command issued by p before c at entry i.
2. *Ordering:* If a process executes command c on state machine $\text{sm}[i]$ without having executed command c' on $\text{sm}[i]$, then no process executes c' without having executed c on $\text{sm}[i]$.
3. *Progress:* There is at least one state machine $\text{sm}[i]$ on which every correct process executes an infinite number of commands.

It is easy to see that for the case where $k = 1$, these properties correspond exactly to those of state machine replication.

3.2 K-set Consensus

We assume here a standard read-write memory but now augmented with the k-set consensus abstraction [5]. We assume k-set consensus in its *vector* form [3]: It takes as input a k-vector of non-nil values, and returns, to each process, a k-vector composed of nil values and exactly one non-nil value among those proposed. We simply call this abstraction the *consensus vector*. It ensures the following properties:

1. *Validity:* any non-nil value returned at some entry i of an output vector is the input of some process at entry i of an input vector.
2. *Agreement:* Any two non-nil values returned to any two processes at the same entry of the output vectors are the same.
3. *Progress:* Every correct process that proposes an input (vector) gets back an output (vector) and every output contains exactly one non-nil value.

It is important to notice that the agreement property above does not prevent one process from getting a non-nil value returned at entry i and nil at entry j, whereas another process is getting some non-nil value at entry j and nil at entry i.

3.3 From 1 to *k*

To get a sense of the technical difficulty behind our generalization, consider first a naive protocol resulting from (a) replacing, in Figure 1, the consensus abstraction with the consensus vector one, and (b) having, in every round r, a process p executes on state machine sm[i] the command obtained at position i from the consensus vector, if any, i.e., if p obtains nil at position i in r, then p does simply not execute anything on state machine sm[i] in round r.

Clearly, such a protocol would guarantee liveness (progress): at least one state machine will remain highly available since the consensus vector will return at least one non-nil value and at least one command will be executed in every round. Yet, safety (ordering) will be violated as we illustrate now through a simple two-round execution of this naive protocol.

- Round 1. Assume p obtains a command c at position 1 (after proposing its initial command vector): p will then accordingly execute c on sm[1]. In the meantime, assume process q obtains a command $c' \neq c$ at position 2 and accordingly executes c' on sm[2].
- Round 2. Assume p obtains a command at position 2 and accordingly executes that command on its state machine sm[2]. This would violate ordering for p ignores that q already executed c' on sm[2] in round 1.

Intuitively, the issue should be sorted out by having every process announce what command it has executed before proceeding to the next round: say q would need to notify p that q has executed c' on sm[2] in round 1. This notification is not trivial for it needs to be synchronized with the action where p needs itself to execute a command on sm[1].

To sort out this issue, *adopt-commit* objects [7] come in handy. These can be implemented in a standard asynchronous read-write memory. We recall below the specification of such objects before explaining how they are used in our context.

3.4 Vectors of Adopt-Commit Objects

The specification of an adopt-commit object is as follows. Every process proposes an input value to such an object and obtains an output value, either in a *committed* or *adopted* status. (One could model such an output as a pair, combining a value and a bit depicting the status committed or adopted of that value). The following properties are satisfied:

1. *Validity:* The output value of any process is an input value of some process.
2. *Agreement:* If a committed value is returned to a process, then no different output value (committed or adopted) can be returned to any other process.
3. *Progress:* Every correct process that proposes an input value obtains an output value.
4. *Commitment:* If no two input values are different, then no output value can be adopted. (It is necessarily committed).

We use a vector of adopt-commit objects at each round, and this vector acts as a synchronization *filter* through which processes go, after passing the consensus vector and before actually executing commands on their state machines. Each process, after obtaining an output from the consensus vector, goes through the vector of adopt-commit objects. (In a specific order we explain below). In short, a process only executes commands that are committed. Those adopted are kept for next round.

3.5 Protocol

Our generalized state machine replication protocol is depicted in Figure 2. We denote the list of consensus vectors by *ConsVectList*, the list of adopt-commit vectors by *AConsVectList* and the list of vectors of commands available to a process by *comVectList*. A process can pick the next element in a list using function *next()* and also recall the last element picked in a list using function *current()*. Processes do not add items in those lists during the execution of the protocol.

The protocol proceeds in rounds. In every round, the initial vector of commands is denoted by *comVect*, the one resulting from the vector of consensus objects is denoted by *comVect1* (this one might contain nil values) and the one resulting from the vector of adopt-commit objects is denoted by *comVect2* (this one contains values in an adopted or committed status). If the latter vector returns a command that is committed (resp. adopted) to a process p, we say that p *commits* (resp. *adopts*) the command.

```
local data structures:
1  smVect[]                                          (* a vector of state machines *)
2  comVectList                                        (* a list of command vectors *)
3  for j := 1 to k do: comVect[j] :=              (* pick the first command vector *)
   comVectList[j].next()

shared data structures:
4  ConsVectList                                     (* a list of consensus vectors *)
5  AConsVectList                                (* a list of adopt-commit vectors *)

forever do:
6  consVect := ConsVectList.next()               (* pick the next consensus vector *)
7  comVect1 := consVect.propose(comVect); (* decide a new vector of commands *)
8  aconsVect := AConsVectList.next()       (* pick the next adopt-commit vector *)

9  for i := 1 to k do:
10   if comVect1[i] ≠ nil then:
11     comVect2[i] := aconsVect[i].propose(comVect1[i])   (* exploit success first *)

12 for i := 1 to k do:
13   if comVect1[i] = nil then:                    (* try to commit old commands *)
14     comVect2[i] := aconsVect[i].propose(comVect[i])

15 for i := 1 to k do:
16   if older(comVect2[i],comVect[i]) then sm[i].execute(comVect[i])  (* catch-up *)
                                            (* keep the command for next round *)
17   if adopted(comVect2[i]) then comVect[i] := comVect2[i]
18   else
19     sm[i].execute(comVect2[i])
20     if comVect2[i] := comVectList[i].current()
21       then comVect[i] := comVectList[i].next()
22       else comVect[i] := comVectList[i].current()
23     add(comVect[i],comVect2[i])          (* remember the committed command *)
```

Fig. 2. Generalized state machine replication

For presentation simplicity, we assume that a process can test if a command was adopted simply using a function *adopted(c)*, a process can encode in a command c' the fact it has committed c, simply by writing $add(c', c)$, and the process can check that fact by simply testing if $older(c, c')$.

Two main ideas underly our generalized state machine replication protocol (Figure 2):

1. *Exploit success first.* To ensure *liveness*, a process p, at round r, accesses first the adopt-commit object corresponding to the non-nil value (i.e., the command) returned by the consensus vector at r (lines 10–11 in Figure 2). Subsequently, p proceeds to the rest of the entries at which is was returned nil and proposes the original commands to the consensus vector (lines 13–14 in Figure 2).

This ensures that at least one process will commit a command in every round. Indeed, for an adopt-commit object not to commit a command, it has to be concurrently invoked with at least two distinct values. The first process p to return from any of the adopt-commit object, by virtue of being first, must commit the command. No process q can prevent p from committing by proposing a distinct command concurrent with p, as then, q's command was not returned there, and q already went through the adopt-commit object of its returned command, contradicting the fact that p was the first to return from any adopt-commit object.

2. *Remember commitments.* To ensure *safety*, a process p might need to execute two commands on the same machine in the same round. A process p might indeed adopt a command c in round r for entry i, then commit another command c' in round $r + 1$ for that same entry i. This might happen if another process q committed c at r and then moved to propose and commit c' at $r + 1$. In this case, p should execute c and then c', both in round $r + 1$. Should p execute c' without having executed c, p would violate safety.
   In our protocol, when q commits a command c in round r, then moves to round $r+1$ with a command c', q encodes in c' the fact that c was committed before c' (line 23 in Figure 2): hence, in round $r + 1$, p will decode that information from c', then execute c before c' (line 16 in in Figure 2). In fact, p executes c even if it only adopts c' in round $r + 1$.
   It is important to notice here that c cannot "get lost" as every process that did not commit c in round r must have adopted c at round r. Hence, all proposed values to the adopt-commit object at entry i at round $r + 1$, which are not c, are commands which encode the very fact that c has been committed at round r.

4 Correctness

Theorem 1. *If a process q executes command c on state machine $sm[i]$, then c was issued by some process p at entry i (of its command vector), and q has executed every command issued by p before c at entry i.*

Proof. There are exactly two places of the algorithm of Figure 2 where a process p can execute a command c on its state machine $\mathrm{sm}[i]$: at line 19 and line 16. For p to execute a command c on $\mathrm{sm}[i]$ at line 19, p must have obtained c from an adopt-commit object at entry i (*comVect2*$[i]$). For p to execute a command c on $\mathrm{sm}[i]$ at line 16, some process p' must have obtained c from an adopt-commit object at entry i (*comVect2*$[i]$) and added it to the command vector at entry i (line 23). In both cases, some process p'' must have proposed c to an adopt-commit object at entry i, and hence must have obtained c from a consensus vector at entry i. In turn, some process p''' must have proposed c to that vector at entry i and hence must have issued the command at entry c. It remains to show now that p has executed on $\mathrm{sm}[i]$ every command c' issued by p before c at entry i. The only place where a process p issues a new command at entry i in the algorithm of Figure 2 is at line 20. By the preliminary test performed in that line, this happens only if the command executed by p at entry i was issued by p for rank i. Hence, p cannot execute a new command at entry i which is issued by p, without having executed the previous command issued by p.

Theorem 2. *If a process executes command c on state machine $sm[i]$ without having executed command c' on $sm[i]$, then no process executes c' without having executed c on $sm[i]$.*

Proof. Assume a process p executes, on its state machine $\text{sm}[i]$, command c at some round r. If this happens at line 16, then some process q has committed c through an adopt-commit object $aconsVect[i]$ in round $r-1$ (line 18). Assume furthermore that process p did not execute c' before c. This means that no adopt-commit object $aconsVect[i]$ has returned c' in a committed status at round $r' < r-1$. Assume now that p executes on $\text{sm}[i]$, command c at round r at line 19. This means that p has committed c through an adopt-commit object $aconsVect[i]$ in round r (line 18). Assume furthermore that process p did not execute c' before c. This means that no adopt-commit object $aconsVect[i]$ has returned c' in a committed status at round $r' < r$. Hence, no process q can execute c' without having executed c on state machine $\text{sm}[i]$.

Lemma 1. *If a process p commits command c in round r on state machine $sm[i]$, then every process which finishes round $r+1$ executes c on state machine $sm[i]$.*

Proof. Assume process p commits command c in round r on state machine $\text{sm}[i]$. By the specification of adopt-commit, $\text{aconsVect}[i]$ returns command c (either in a committed or adopted status) to all processes that invoked it in round r. Hence, all processes which start round $r+1$ either (a) executed c on $\text{sm}[i]$ in round r and start round $r+1$ with a command c' such that c' encoded the commitment of c (line 23), or (b) start round $r+1$ with command c itself (line 17). In both cases, any process that did not execute c in round r will, in round $r+1$, either commit c and execute it (line 19) or learn about c having been committed and execute it (line 16).

Theorem 3. *An infinite number of commands are executed on at least one state machine at all correct processes.*

Proof. Assume at least one process is correct. Assume by contradiction that there is a round at which no process executes a command on a state machine. This means that no adopt-commit object returns a committed command. Given that the protocol of Figure 2 has no wait statement, every adopt-commit object must have had two different concurrently proposed values. This means that all processes obtained different values from the consensus vector. Consequently, all processes started at different adopt-commit objects. This is in contradiction with the fact that every adopt-commit object has two different concurrent proposals. Hence, at least one process commits a command on at least one machine in every round. This follows from the order according to which processes access adopt-commit objects. By Lemma 1, all correct processes execute a command on at least one machine every two rounds. Hence, there is at least one state machine on which all correct processes execute an infinite sequence of commands.

5 Concluding Remarks

When k-set consensus was introduced [5], as creative feat as it was, it was formulated in the "wrong" way. The question "what the analogue of (consensus) state machine replication is?" could not be imagined, as k-set referred to multiple values. When [3] equated k-set consensus with vector consensus, the question of generalizing state machine replication started to make sense. In retrospect, generalized state machines replication as presented here is so simple, that it begs the question "why so long?"

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
2. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: STOC (1993)
3. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: The k-simultaneous consensus problem. Distributed Computing 22(3), 185–195 (2010)
4. Lamport, L.: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
5. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
6. Fischer, M., Lynch, N., Paterson, M.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
7. Gafni, E.: Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. In: PODC (1998)
8. Herlihy, M.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 11(1), 124–149 (1991)
9. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
10. Saks, M., Zaharoglou, F.: Wait-Free k-set consensus is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1987)

Causal Nets: A Modeling Language Tailored towards Process Discovery

Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
{W.M.P.v.d.Aalst,A.Adriansyah,B.F.v.Dongen}@tue.nl

Abstract. Process discovery—discovering a process model from example behavior recorded in an event log—is one of the most challenging tasks in process mining. The primary reason is that conventional modeling languages (e.g., Petri nets, BPMN, EPCs, and ULM ADs) have difficulties representing the observed behavior properly and/or succinctly. Moreover, discovered process models tend to have deadlocks and livelocks. Therefore, we advocate a new representation more suitable for process discovery: *causal nets*. Causal nets are related to the representations used by several process discovery techniques (e.g., heuristic mining, fuzzy mining, and genetic mining). However, unlike existing approaches, we provide declarative semantics more suitable for process mining. To clarify these semantics and to illustrate the non-local nature of this new representation, we relate causal nets to Petri nets.

1 Motivation

In this paper, we advocate the use of *Causal-nets* (*C-nets*) in process mining. C-nets were introduced in [2] and, in our view, provide a better representational bias for process discovery than conventional design-oriented languages such as Petri nets, BPMN, BPEL, EPCs, YAWL, and UML activity diagrams.

Figure 1 shows a C-net modeling the booking of a trip. After activity a (*start booking*) there are three possible activities: b (*book flight*), c (*book car*), and d (*book hotel*). The process ends with activity e (*complete booking*). Each activity has sets of potential input and output bindings (indicated by the black dots). Every connected set of dots on the output arcs of an activity is an output binding. For example, a has four output bindings modeling that a may be followed by (1) just b, (2) just c, (3) b and d, or (4) b, c, and d. Hence, it is not possible to book just a hotel or a hotel and a car. Activity c has two input bindings modeling that it is preceded by (1) just a or (2) a and b. This construct is used to model that when both a flight and a car are booked, the flight is booked first. Output bindings create *obligations* whereas input bindings remove obligations. For example, the occurrence of a with output binding $\{b, d\}$ creates two obligations: both b and d need to be executed while referring to the obligations created by a.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 28–42, 2011.

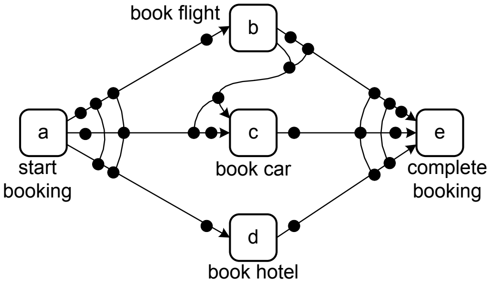


Fig. 1. Causal net C_{travel}

In a C-net there is one start activity (a in Fig. 1) and one end activity (e in Fig. 1). A *valid binding sequence* models an execution path starting with a and ending with e while removing all obligations created during execution. The behavior of a C-net is *restricted* to valid binding sequences. Hence, unlike conventional modeling languages, the semantics are non-local. Section 2 explains the semantics of C-nets in more detail and provides additional examples.

C-nets address important limitations of conventional languages in the context of *process mining* [2]. Process mining is an emerging research discipline focusing on the interplay between *event logs* (observed behavior) and process models. *Process discovery* is the process mining task that aims to learn process models based on example behavior recorded in events logs, e.g., based on a multi-set of activity sequences (process instances) a Petri net that models the observed behavior is discovered. *Conformance checking* is the process mining task that compares the example behavior in a events log with the modeled behavior. Based on such a comparison it is possible to highlight and quantify commonalities and differences.

In the last decade dozens of new process discovery techniques have been proposed, typically aiming at the creation of a conventional process model (e.g., a Petri net or EPC). This means that the search space that is implied by such a design-oriented language—often referred to as the "representational bias"—is not tailored towards process mining. This creates various problems. In this paper, we focus on two of them:

- The discovered process model is *unable to represent the underlying process well*, e.g., a significant proportion of the behavior seen in the log is not possible in the model (non-fitting model), the model allows for behavior not related to the event log (underfitting), the model is overfitting (no generalization), or the model is overly complex because all kinds of model elements need to be introduced without a direct relation to the event log (e.g., places, gateways, and events).
- Most of the process models in the search space determined by conventional languages are *internally inconsistent* (deadlocks, livelocks, etc.), i.e., there are more inconsistent models than consistent ones. Process discovery

techniques need to "guess" the underlying model based on example behavior. If almost all of these guesses result in models that are obviously incorrect (even without considering the event log), then the results are of little value.

Consider for example an algorithm producing a Petri net (e.g., the various region-based approaches [11] and variants of the α-algorithm [2]). The behavior in a Petri net can be restricted by adding places. However, places have no direct meaning in terms of the behavior seen in the event log. Moreover, the addition or removal of places may introduce deadlocks, livelocks, etc.

This is the reason why the more useful process discovery techniques use alternative representations: *fuzzy models* [7], *heuristic nets* [9], *flexible heuristic nets* [10], *causal matrices* [8], etc. Also for conformance checking one can find similar representations, e.g., *flexible models* [4]. On the one hand, these representations are similar to C-nets (i.e., activities can model XOR/OR/AND-splits and joins without introducing separate model elements). On the other hand, the semantics of such models are very different from the semantics we use for C-nets. The distinguishing feature is that we *limit the possible behavior to valid binding sequences*, thus excluding a variety of anomalies.

This paper introduces C-nets while focusing on their semantics (Sect. 2). We believe that our formalization sheds new light on the representations used in [4,7,8,9,10]. We also provide two mappings: one from C-nets to Petri nets and one from Petri nets to C-nets (Sect. 3). These mappings help to clarify the semantics and highlight the distinguishing features of C-nets. Moreover, to illustrate the practical relevance of C-nets, we describe how the ProM framework is supporting/using C-nets (Sect. 4).

2 Causal Nets

This section introduces *causal nets* – a representation tailored towards process mining – and their semantics.

2.1 Definition

A Causal-net (C-net) is a graph where nodes represent activities and arcs represent causal dependencies. Each activity has a set of possible *input bindings* and a set of possible *output bindings*. Consider, for example, the C-net shown in Fig. 2. Activity a has only an empty input binding as this is the start activity. There are two possible output bindings: $\{b, d\}$ and $\{c, d\}$. This means that a is followed by either b and d, or c and d. Activity e has two possible input bindings ($\{b, d\}$ and $\{c, d\}$) and three possible output bindings ($\{g\}$, $\{h\}$, and $\{f\}$). Hence, e is preceded by either b and d, or c and d, and is succeeded by just g, h or f. Activity z is the end activity having two input bindings and one output binding (the empty binding). This activity has been added to create a unique end point. All executions commence with start activity a and finish with end activity z. Note that unlike, Petri nets, there are no places in the C-net; the routing logic is solely represented by the possible input and output bindings.

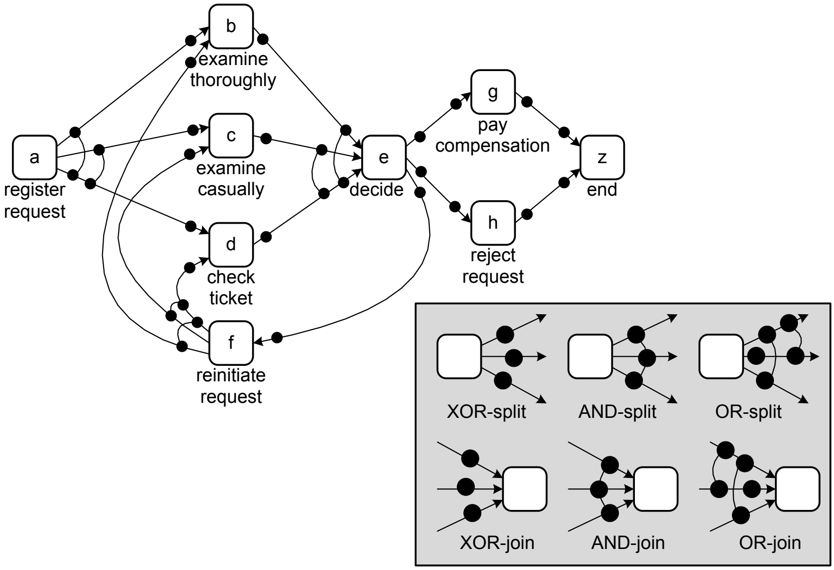


Fig. 2. C-net C_{rfc} modeling a "Request For Compensation" (RFC) process

Definition 1 (Causal net [2]). *A* Causal net *(C-net) is a tuple* $C = (A, a_i, a_o, D, I, O)$ *where:*

- A *is a finite set of* activities*;*
- $a_i \in A$ *is the* start activity*;*
- $a_o \in A$ *is the* end activity*;*
- $D \subseteq A \times A$ *is the* dependency relation*,*
- $AS = \{X \subseteq \mathcal{P}(A) \mid X = \{\emptyset\} \ \vee \ \emptyset \notin X\}$*,*[1]
- $I \in A \rightarrow AS$ *defines the set of possible* input bindings *per activity; and*
- $O \in A \rightarrow AS$ *defines the set of possible* output bindings *per activity,*

such that

- $D = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup_{as \in I(a_2)} as\}$*;*
- $D = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup_{as \in O(a_1)} as\}$*;*
- $\{a_i\} = \{a \in A \mid I(a) = \{\emptyset\}\}$*;*
- $\{a_o\} = \{a \in A \mid O(a) = \{\emptyset\}\}$*; and*
- *all activities in the graph* (A, D) *are on a path from* a_i *to* a_o*.*

The C-net of Fig. 2 can be described as follows. $A = \{a, b, c, d, e, f, g, h, z\}$ is the set of activities, $a = a_i$ is the unique start activity, and $z = a_o$ is the unique end activity. The arcs shown in Fig. 2 visualize the dependency relation

[1] $\mathcal{P}(A) = \{A' \mid A' \subseteq A\}$ is the powerset of A. Hence, elements of AS are *sets of sets* of activities.

$D = \{(a,b),(a,c),(a,d),(b,e),\ldots,(g,z),(h,z)\}$. Functions I and O describe the sets of possible input and output bindings. $I(a) = \{\emptyset\}$ is the set of possible input bindings of a, i.e., the only input binding is the empty set of activities. $O(a) = \{\{b,d\},\{c,d\}\}$ is the set of possible output bindings of a, i.e., activity a is followed by d and either b or c. $I(b) = \{\{a\},\{f\}\}$, $O(b) = \{\{e\}\}$, $\ldots$, $I(z) = \{\{g\},\{h\}\}$, $O(z) = \{\emptyset\}$. Note that any element of AS is a set of sets of activities, e.g., $\{\{b,d\},\{c,d\}\} \in AS$. If one of the elements is the empty set, then there cannot be any other elements, i.e., for any any $X \in AS$: $X = \{\emptyset\}$ or $\emptyset \notin X$. This implies that only the unique start activity a_i has the empty binding as (only) possible input binding. Similarly, only the unique end activity a_o has the empty binding as (only) possible output binding.

An *activity binding* is a tuple (a, as^I, as^O) denoting the occurrence of activity a with input binding as^I and output binding as^O. For example, $(e,\{b,d\},\{f\})$ denotes the occurrence of activity e in Fig. 2 while being preceded by b and d, and succeeded by f.

Definition 2 (Binding). *Let $C = (A, a_i, a_o, D, I, O)$ be a C-net. $B = \{(a, as^I, as^O) \in A \times \mathcal{P}(A) \times \mathcal{P}(A) \mid as^I \in I(a) \;\wedge\; as^O \in O(a)\}$ is the set of* activity bindings. *A* binding sequence *σ is a sequence of activity bindings, i.e., $\sigma \in B^*$.*

Note that sequences are denoted using angle brackets, e.g., $\langle\,\rangle$ denotes the empty sequence. B^* is the set of all sequences over B (including $\langle\,\rangle$). A possible binding sequence for the C-net of Fig. 2 is $\sigma_{ex} = \langle(a,\emptyset,\{b,d\}), (b,\{a\},\{e\}), (d,\{a\},\{e\}), (e,\{b,d\},\{g\}), (g,\{e\},\{z\}), (z,\{g\},\emptyset)\rangle$.

Function $\alpha \in B^* \rightarrow A^*$ projects binding sequences onto *activity sequences*, i.e., the input and output bindings are abstracted from and only the activity names are retained. For instance, $\alpha(\sigma_{ex}) = \langle a,b,d,e,g,z\rangle$.

Consider C-net C_{travel} shown in Figure 1. The possible input and output bindings of C_{travel} are defined as follows: $O(a) = I(e) = \{\{b\},\{c\},\{b,d\}, \{b,c,d\}\}$, $I(a) = O(e) = \{\emptyset\}$, $I(b) = I(d) = \{\{a\}\}$, $O(c) = O(d) = \{\{e\}\}$, $I(c) = \{\{a\},\{a,b\}\}$, and $O(b) = \{\{e\},\{c,e\}\}$. A possible binding sequence for the C-net shown in Fig. 1 is $\sigma = \langle(a,\emptyset,\{b,c,d\}), (d,\{a\},\{e\}), (b,\{a\},\{c,e\}), (c,\{a,b\},\{e\}), (e,\{b,c,d\},\emptyset)\rangle$, i.e., the scenario in which a hotel, a flight, and a car are booked. $\alpha(\sigma) = \langle a,d,b,c,e\rangle$ is the corresponding activity sequence. Note that in Fig. 1 a hotel can only be booked if a flight is booked. Moreover, when both a car and a flight are booked, then first the flight needs to be booked.

2.2 Valid Sequences

A binding sequence is *valid* if a predecessor activity and successor activity always "agree" on their bindings. For a predecessor activity x and successor activity y we need to see the following "pattern": $\langle\ldots,(x,\{\ldots\},\{y,\ldots\}),\ldots, (y,\{x,\ldots\},\{\ldots\}),\;\ldots\rangle$, i.e., an occurrence of activity x with y in its output binding needs to be followed by an occurrence of activity y, and an occurrence of activity y with x in its input binding needs to be preceded by an occurrence of activity x. To formalize the notion of a valid sequence, we first define the notion of *state*. States are represented by multi-sets of *obligations*, e.g., state $[(a,b)^2,(a,c)]$ denotes the state where there are two pending activations of b by a and there is one pending activation of c by a. This means that b needs to happen twice while having a in its input binding and c needs to happen once while having a in its input binding.

Definition 3 (State). *Let $C = (A, a_i, a_o, D, I, O)$ be a C-net. $S = \mathbb{B}(A \times A)$ is the* state space *of C. $s \in S$ is a* state*, i.e., a multi-set of pending* obligations*. Function $\psi \in B^* \rightarrow S$ is defined inductively: $\psi(\langle\,\rangle) = [\,]$ and $\psi(\sigma \oplus (a, as^I, as^O)) = (\psi(\sigma) \setminus (as^I \times \{a\})) \uplus (\{a\} \times as^O)$ for any binding sequence $\sigma \oplus (a, as^I, as^O) \in B^*$.*[2] *$\psi(\sigma)$ is the state after executing binding sequence σ.*

Consider C-net C_{rfc} shown in Fig. 2. Initially there are no pending "obligations", i.e., no output bindings have been enacted without having corresponding input bindings. If activity binding $(a, \emptyset, \{b,d\})$ occurs, then $\psi(\langle(a,\emptyset,\{b,d\})\rangle) = \psi(\langle\,\rangle) \setminus (\emptyset \times \{a\}) \uplus (\{a\} \times \{b,d\}) = ([\,] \setminus [\,]) \uplus [(a,b),(a,d)] = [(a,b),(a,d)]$. State $[(a,b),(a,d)]$ denotes the obligation to execute both b and d using input bindings involving a. Input bindings remove pending obligations whereas output bindings create new obligations.

A *valid sequence* is a binding sequence that (1) starts with start activity a_i, (2) ends with end activity a_o, (3) only removes obligations that are pending, and (4) ends without any pending obligations. Consider, for example, the valid sequence $\sigma = \langle(a,\emptyset,\{b,d\}),(d,\{a\},\{e\}),(b,\{a\},\{e\}),(e,\{b,d\},\emptyset)\rangle$ for C-net C_{travel} in Fig. 1:

$$\begin{aligned}
\psi(\langle\,\rangle) &= [\,]\\
\psi(\langle(a,\emptyset,\{b,d\})\rangle) &= [(a,b),(a,d)]\\
\psi(\langle(a,\emptyset,\{b,d\}),(d,\{a\},\{e\})\rangle) &= [(a,b),(d,e)]\\
\psi(\langle(a,\emptyset,\{b,d\}),(d,\{a\},\{e\}),(b,\{a\},\{e\})\rangle) &= [(b,e),(d,e)]\\
\psi(\langle(a,\emptyset,\{b,d\}),(d,\{a\},\{e\}),(b,\{a\},\{e\}),(e,\{b,d\},\emptyset)\rangle) &= [\,]
\end{aligned}$$

Sequence σ indeed starts with start activity a, ends with end activity e, only removes obligations that are pending (i.e., for every input binding there was an earlier output binding), and ends without any pending obligations: $\psi(\sigma) = [\,]$.

Definition 4 (Valid). *Let $C = (A, a_i, a_o, D, I, O)$ be a C-net and $\sigma = \langle(a_1, as_1^I, as_1^O), (a_2, as_2^I, as_2^O), \ldots, (a_n, as_n^I, as_n^O)\rangle \in B^*$ be a binding sequence. σ is a* valid sequence *of C if and only if:*

- *$a_1 = a_i$, $a_n = a_o$, and $a_k \in A \setminus \{a_i, a_o\}$ for $1 < k < n$;*
- *$\psi(\sigma) = [\,]$; and*
- *for any non-empty prefix $\sigma' = \langle(a_1, as_1^I, as_1^O), \ldots, (a_k, as_k^I, as_k^O)\rangle$ $(1 \leq k \leq n)$: $(as_k^I \times \{a_k\}) \leq \psi(\sigma'')$ with $\sigma'' = \langle(a_1, as_1^I, as_1^O), \ldots, (a_{k-1}, as_{k-1}^I, as_{k-1}^O)\rangle$*

$V_{CN}(C)$ is the set of all valid sequences of C.

[2] $\oplus$ is used to concatenate an element to the end of a sequence, e.g., $\langle a,b,c\rangle \oplus d = \langle a,b,c,d\rangle$. $X \uplus Y$ is the union of two multi-sets. $X \setminus Y$ removes Y from X (difference of two multi-sets). Ordinary sets will be used as multi-sets throughout this paper.

The first requirement states that valid sequences start with a_i and end with a_o (a_i and a_o cannot appear in the middle of valid sequence). The second requirement states that at the end there should not be any pending obligations. (One can think of this as the constraint that no tokens left in the net.) The last requirement considers all non-empty prefixes of σ: $\langle (a_1, as^I_1, as^O_1), \ldots, (a_k, as^I_k, as^O_k)\rangle$. The last activity binding of the prefix (i.e., (a_k, as^I_k, as^O_k)) should only remove pending obligations, i.e., $(as^I_k \times \{a_k\}) \leq \psi(\sigma'')$ where $as^I_k \times \{a_k\}$ are the obligations to be removed and $\psi(\sigma'')$ are the pending obligations just before the occurrence of the k-th binding. (One can think of this as the constraint that one cannot consume tokens that have not been produced.)

The C-net in Fig. 1 has seven valid sequences: only b is executed ($\langle (a, \emptyset, \{b\}), (b, \{a\}, \{e\}), (e, \{b\}, \emptyset)\rangle$), only c is executed (besides a and e), b and d are executed (two possibilities), and b, c and d are executed (3 possibilities because b needs to occur before c). The C-net in Fig. 2 has infinitely many valid sequences because of the loop construct involving f.

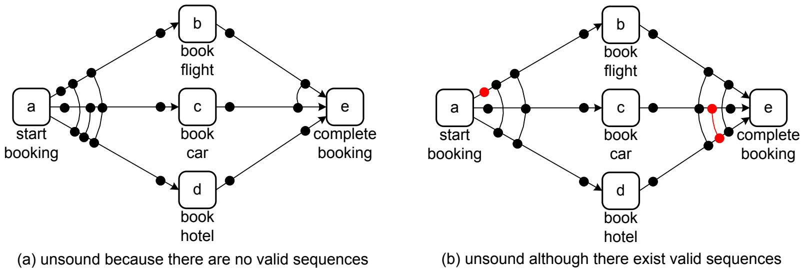
For the semantics of a C-net we only consider valid sequences, i.e., *invalid sequences are not part of the behavior* described by the C-net. This means that C-nets do not use plain "token-game semantics" as employed in conventional languages like BPMN, Petri nets, EPCs, and YAWL. The semantics of C-nets are more declarative as they are defined over complete sequences rather than a local firing rule. Note that the semantics abstract from the moment of choice; pending obligations are not exposed to the environment and are not fixed during execution (i.e., all *valid* interpretations remain open).

2.3 Soundness

The notion of *soundness* has been defined for a variety of workflow and business process modeling notations (e.g., workflow nets as shown in Sect. 3.1). A process model is *sound* if it is free of deadlocks, livelocks, and other obvious anomalies. A similar notion can be defined for C-nets.

Definition 5 (Soundness of C-nets [2]). *A C-net* $C = (A, a_i, a_o, D, I, O)$ *is* sound *if (1) for all* $a \in A$ *and* $as^I \in I(a)$ *there exists a* $\sigma \in V_{CN}(C)$ *and* $as^O \subseteq A$ *such that* $(a, as^I, as^O) \in \sigma$*, and (2) for all* $a \in A$ *and* $as^O \in O(a)$ *there exists a* $\sigma \in V_{CN}(C)$ *and* $as^I \subseteq A$ *such that* $(a, as^I, as^O) \in \sigma$*.*

Since the semantics of C-nets already enforce "proper completion" and the "option to complete", we only need to make sure that there are valid sequences and that all parts of the C-net can potentially be activated by such a valid sequence. The C-nets C_{travel} and C_{rfc} in Figs. 1 and 2 are sound. Figure 3 shows two C-nets that are not sound. In Fig. 3(a), there are no valid sequences because none of output bindings of a matches any of the input bindings of e. For example, consider the binding sequence $\sigma = \langle (a, \emptyset, \{b\}), (b, \{a\}, \{e\})\rangle$. Sequence σ cannot be extended into a valid sequence because $\psi(\sigma) = [(b, e)]$ and $\{b\} \not\in I(e)$, i.e., the input bindings of e do not allow for just booking a flight whereas the output bindings of a do. In Fig. 3(b), there are valid sequences, e.g.,



(a) unsound because there are no valid sequences

(b) unsound although there exist valid sequences

Fig. 3. Two C-nets that are not sound. The first net (a) does not allow for any valid sequence, i.e., $V_{CN}(C) = \emptyset$. The second net (b) has valid sequences but also shows input/output bindings that are not realizable (indicated in red).

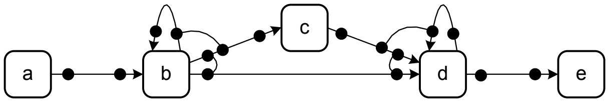


Fig. 4. A sound C-net for which there does not exist a WF-net having the same set of activity sequences

$\langle(a, \emptyset, \{c\}), (c, \{a\}, \{e\}), (e, \{c\}, \emptyset)\rangle$. However, not all bindings appear in one or more valid sequences. For example, the output binding $\{b\} \in O(a)$ does not appear in any valid sequence, i.e., after selecting just a flight the sequence cannot be completed properly. The input binding $\{c, d\} \in I(e)$ also does not appear in any valid sequence, i.e., the C-net suggests that only a car and hotel can be booked but there is no corresponding valid sequence.

Figure 4 shows another C-net. One of the valid binding sequences for this C-net is $\langle(a, \emptyset, \{b\}), (b, \{a\}, \{b, c\})$, $(b, \{b\}, \{c, d\})$, $(c, \{b\}, \{d\})$, $(c, \{b\}, \{d\})$, $(d$, $\{b, c\}, \{d\})$, $(d, \{c, d\}, \{e\}), (e, \{d\}, \emptyset)\rangle$, i.e., the sequence $\langle a, b, b, c, c, d, d, e\rangle$. This sequence covers all the bindings. Therefore, the C-net is sound. Examples of other valid sequences are $\langle a, b, c, d, e\rangle$, $\langle a, b, c, b, c, d, d, e\rangle$, and $\langle a, b, b, b, c, c, c, d, d, d, e\rangle$.

C-nets are particularly suitable for process mining given their declarative nature and expressiveness without introducing all kinds of additional model elements (places, conditions, events, gateways, etc.). Several process discovery use similar representations [7,8,9,10]. However, these models tend to use rather informal semantics; the model serves more like a "picture" showing dependencies rather than an end-to-end process model.

3 Relating C-nets and Petri Nets

To better understand the semantics of C-nets, we relate C-nets to Petri nets. We provide a mapping from WF-nets to C-nets and show that the resulting C-net is behaviorally equivalent to the original WF-net. We also provide a mapping from C-nets to WF-nets that over-approximates the behavior.

3.1 Petri Nets and WF-nets

We assume that the reader is familiar with Petri nets. Therefore, we just summarize the basic concepts and notations relevant for the two mappings.

Definition 6 (Petri net). *A* Petri net *is a triplet $N = (P, T, F)$ where P is a finite set of* places*, T is a finite set of* transitions *such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the* flow relation*. A* marked *Petri net is a pair (N, M), where $N = (P, T, F)$ is a Petri net and where $M \in \mathbb{B}(P)$ is a* multi-set *over P denoting the* marking *of the net.*

Petri nets are defined in the standard way. Markings, i.e., states of the net, are denoted as multi-sets. For any $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ (input nodes) and $x\bullet = \{y \mid (x, y) \in F\}$ (output nodes). A transition t is *enabled* if each of its input places $\bullet t$ contains at least one token. An enabled transition t may *fire*, i.e., one token is removed from each of the input places $\bullet t$ and one token is produced for each of the input places $t\bullet$. Formally: $(M \setminus \bullet t) \uplus t\bullet$ is the marking resulting from firing enabled transition t in marking M.

A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, M_0) if and only if, for some $n \in \{0, 1, \ldots\}$, there exist markings $M_1, \ldots, M_n$ and transitions $t_1, \ldots, t_n \in T$ such that $\sigma = \langle t_1 \ldots t_n \rangle$ and, for all i with $0 \leq i < n$, t_{i+1} is enabled in M_i and firing t_{i+1} results in marking M_{i+1}.

For business process modeling and process mining, often a restricted class of Petri nets is used: *Workflow nets (WF-nets)* [1,3]. The reason is that *process instances* have a clear starting and ending point. For example, a customer order, a patient treatment, a request for a mortgage, etc. all have a life-cycle with a well-defined start and end. Process instances are often referred to as *cases*. A WF-net describes the life-cycle of such cases.

Definition 7 (Workflow net [1]). *Petri net $N = (P, T, F)$ is a* workflow net *(WF-net) if and only if (1) P contains an input place p_i (also called source place) such that $\bullet p_i = \emptyset$, (2) P contains an output place p_o (also called sink place) such that $p_o\bullet = \emptyset$, and (3) every node is on a path from p_i to p_o.*

Cases start in the marking $[p_i]$ (one token in the unique source place) and ideally end in the marking $[p_o]$ (one token in the unique sink place). The WF-net should ensure that it is always possible to reach the final marking $[p_o]$. Moreover, a WF-net should not contain dead parts, i.e., parts that can never be activated. These requirements result in the classical definition of soundness for WF-nets.

Definition 8 (Soundness [1,3]). *Let $N = (P, T, F)$ be a WF-net with input place p_i and output place p_o. N is* sound *if and only if (1) for any marking reachable from $[p_i]$ it is possible to reach the marking $[p_o]$ (*option to complete*), and (2) $(N, [p_i])$ contains no dead transitions (*absence of dead parts*, i.e., for any $t \in T$, there is a firing sequence enabling t).*

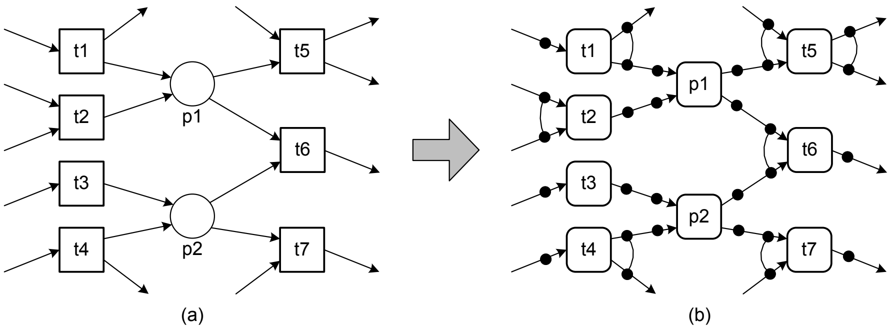


Fig. 5. Mapping a fragment of a WF-net (a) onto a C-net (b) using Definition 10

We are interested in the set $V_{PN}(N)$ of all firing sequences that start in marking $[p_i]$ and end in marking $[p_o]$. Note that in a sound WF-net, all full firing sequences (i.e., firing sequences ending in a dead marking) are valid.

Definition 9 (Valid firing sequences). *Let $N = (P, T, F)$ be a WF-net. $V_{PN}(N) \subseteq T^*$ is the set of all* valid firing sequences*, i.e., firing sequences starting in marking $[p_i]$ and ending in marking $[p_o]$.*

At first sight, C-nets seem to be related to *zero-safe nets* [5]. The places in a zero-safe net are partitioned into *stable-places* and *zero-places*. Observable markings only mark stable-places, i.e., zero-places need to be empty. In-between observable markings zero-places may be temporarily marked. However, zero-places cannot be seen as bindings because the obligations between two activities may be non-local, i.e., an output binding may create the obligation to execute an activity occurring much later in the process.

3.2 Mapping WF-nets onto C-nets

Any sound WF-net can be transformed into an equivalent C-net by converting places into activities with XOR-join and XOR-split bindings. The idea is sketched in Fig. 5 and can be formalized as follows.

Definition 10 (Mapping I). *Let $N = (P, T, F)$ be a WF-net with input place p_i and output place p_o. $C_N = (A, a_i, a_o, D, I, O)$ is the corresponding C-net with $A = T \cup P$, $a_i = p_i$, $a_o = p_o$, $D = F$, $I(t) = \{\bullet t\}$ and $O(t) = \{t\bullet\}$ for $t \in T$, and $I(p) = \{\{t\} \mid t \in \bullet p\}$ and $O(p) = \{\{t\} \mid t \in p\bullet\}$ for $p \in P$.*

To relate valid firing sequences in WF-nets to valid binding sequences in C-nets, we define a generic projection operation. $\sigma \uparrow Y$ is the projection of some sequence $\sigma \in X^*$ onto some subset $Y \subseteq X$, i.e., elements of σ not in Y are removed. This operation can be generalized to sets of sequences, e.g., $\{\langle a, b, c, a, b, c, d\rangle, \langle b, b, d, e\rangle\} \uparrow \{a, b\} = \{\langle a, b, a, b\rangle, \langle b, b\rangle\}$.

Theorem 1. *Let $N = (P, T, F)$ be a sound WF-net having C_N as its corresponding C-net.*

- *For any valid firing sequence $\sigma_N \in V_{PN}(N)$, there exists a valid binding sequence $\sigma_C \in V_{CN}(C_N)$ such that $\alpha(\sigma_C) \upharpoonright T = \sigma_N$.*
- *For any valid binding sequence $\sigma_C \in V_{CN}(C_N)$, there exists a valid firing sequence $\sigma_N \in V_{PN}(N)$ such that $\alpha(\sigma_C) \upharpoonright T = \sigma_N$.*

Proof. Let σ_N be a valid firing sequence of N. Replay σ_N on N while labeling each token with the name of the transition that produced it. Suppose that $t6$ in Fig. 5 fires while consuming a token from $p1$ produced by $t2$ and a token from $p2$ produced by $t3$. This occurrence of $t6$ corresponds to the subsequence $\langle \ldots, (p1, \{t2\}, \{t6\}), (p2, \{t3\}, \{t6\}), (t6, \{p1, p2\}, \{\ldots\})\rangle$. This way it is possible to construct a valid binding sequence σ_C. Note that there may be multiple valid binding sequences corresponding to σ_N.

Let σ_C be a valid binding sequence. It is easy to see that σ_C can be replayed on the WF-net. In fact, one can simply abstract from "place activities" as these correspond to routing decisions not relevant for WF-nets (only the presence of a token matters not where the token came from). Therefore, each σ_C corresponds to a single σ_N. □

C-nets are at least as expressive as sound WF-nets because all valid firing sequences in N have a corresponding valid binding sequence in C_N and vice-versa. The reverse does not hold as is illustrated by Fig. 4. This model is unbounded and has infinitely many binding sequences. Since sound WF-nets are bounded [1,3], they can never mimic the behavior of the C-net in Fig. 4.

3.3 Mapping C-nets onto WF-nets

Figure 4 illustrates that WF-nets are not as expressive as C-net. Nevertheless, it is interesting to construct WF-nets that over-approximate the behavior of C-nets.

Definition 11 (Mapping II). *Let $C = (A, a_i, a_o, D, I, O)$ be a C-net. $N_C = (P, T, F)$ is the corresponding WF-net with $P = \{p^I_a \mid a \in A\} \cup \{p^O_a \mid a \in A\} \cup \{p^D_{(a_1,a_2)} \mid (a_1, a_2) \in D\}$, $T^I = \{a^I_X \mid a \in A \ \wedge \ X \in I(a) \ \wedge \ X \neq \emptyset\}$, $T^O = \{a^O_X \mid a \in A \ \wedge \ X \in O(a) \ \wedge \ X \neq \emptyset\}$, $T = A \cup T^I \cup T^O$, $F = \{(p^I_a, a) \mid a \in A\} \cup \{(a, p^O_a) \mid a \in A\} \cup \{(a^I_X, p^I_a) \mid a^I_X \in T^I\} \cup \{(p^O_a, a^O_X) \mid a^O_X \in T^O\} \cup \{(p^D_{(a_1,a)}, a^I_X) \mid a^I_X \in T^I \ \wedge \ a_1 \in X\} \cup \{(a^O_X, p^D_{(a,a_2)}) \mid a^O_X \in T^O \ \wedge \ a_2 \in X\}$.*

Figure 6 illustrates this construction. The black transitions correspond to silent transitions (often referred to as τ transitions). Since there is a unique start activity a_i, there is one source place $p_i = p^I_{a_i}$. Moreover, there is one sink place $p_o = p^O_{a_o}$ and all nodes are on a path from p_i to p_o. Therefore, N_C is indeed a WF-net.

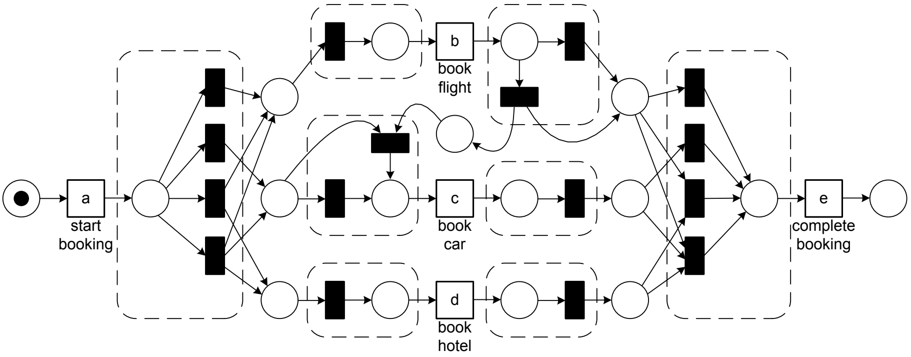


Fig. 6. A C-net transformed into a WF-net: every valid firing sequence of the WF-net corresponds to a valid sequence of the C-net C_{travel} shown in in Fig. 1 and vice versa

It is easy to see that Definition 11 is such that the WF-net can mimic any valid binding sequence. However, the corresponding WF-net does not need to be sound and may have a firing sequence that cannot be extended into a valid firing sequence.

Theorem 2. *Let $C = (A, a_i, a_o, D, I, O)$ be a C-net having N_C as its corresponding WF-net.*

- *For any valid binding sequence $\sigma_C \in V_{CN}(C)$, there exists a valid firing sequence $\sigma_N \in V_{PN}(N_C)$ such that $\alpha(\sigma_C) = \sigma_N \uparrow A$.*
- *For any valid firing sequence $\sigma_N \in V_{PN}(N_C)$, there exists a valid binding sequence $\sigma_C \in V_{CN}(C)$ such that $\alpha(\sigma_C) = \sigma_N \uparrow A$.*

Proof. It is easy to construct a valid firing sequence σ_N for any valid binding sequence σ_C. An activity binding (a, X, Y) in σ_C corresponds to the firing subsequence $\langle a^I_X, a, a^O_Y \rangle$ in σ_N. (For the start and end activity, a^I_X respectively a^O_Y are omitted.) The constructed sequence meets all requirements.

Let σ_N be a valid firing sequence. Consider the occurrence of a transition $a \in A$ in σ_N. Based on the structure of the WF-net it can be seen that a was preceded by a *corresponding* transition in T^I (unless $a = a_i$) and will be followed by a *corresponding* transition in T^O (unless $a = a_o$). The reason is that a has a dedicated input place (no other transition can consume from it) and a dedicated output place (no other transition can add tokens) and that after executing σ_N only $p^O_{a_o}$ is marked. Hence, for every occurrence of some transition $a \in A$ there is a corresponding occurrence of a transition $a^I_X \in T^I$ and a corresponding occurrence of a transition $a^O_Y \in T^O$. This information can be used to construct $\sigma_C \in V_{CN}(C)$ such that $\alpha(\sigma_C) = \sigma_N \uparrow A$. □

The theorem shows that the expressiveness of C-nets is due its declarative semantics which considers only valid binding sequences (and not the notation itself). If one restricts WF-nets to valid firing sequences (and allows for silent transitions!), the same expressiveness is achieved.[3] Note that this is related to the notion of *relaxed soundness* [6]. In fact, a C-net C is sound if and only if the corresponding WF-net N_C is relaxed sound. In [6] it is shown that for some relaxed sound WF-nets a corresponding sound WF-net can be constructed.

4 Application of C-nets in ProM

In the previous sections we introduced C-nets and related them to Petri nets. After these theoretical considerations, we briefly describe the way in which the *ProM framework* supports C-nets. ProM is an open-source process analysis tool with a pluggable architecture. Originally, the focus of ProM was exclusively on process mining. However, over time the scope of the system broadened to also include other types of analysis (e.g., verification). In the remainder, we provide a brief overview of ProM's functionality. Note that we show just a fraction of the hundreds of plug-ins available (cf. `www.processmining.org`).

4.1 Model Management and Conversion

ProM is able to load and save C-nets in a dedicated file format. Petri nets can be converted to C-nets using the construction of Definition 10. Similarly, it is possible to convert a C-net into a Petri net using the construction of Definition 11. Conversions to and from other formats (EPCs, BPMN, etc.) are being developed. These formats can already be converted to Petri nets thus enabling an indirect conversion from these formats to C-nets.

4.2 Model-Based Verification

ProM has extensive support for transition systems and Petri nets. Moreover, also Petri nets with reset and inhibitor arcs and specific subclasses such as WF-nets are supported. Typical Petri nets properties such as liveness, boundedness, etc. can be analyzed using various plug-ins. ProM also embeds the well-known LoLA (a Low Level Petri Net Analyzer) tool for more advanced forms of model-based analysis. There are also plug-ins analyzing structural properties of the net (invariants, traps, siphons, components, etc.). These plug-ins can be applied to WF-nets. Moreover, plug-ins like Woflan are able to verify soundness and diagnose errors.

The plug-in "Check Soundness of Causal Net" checks the property defined in Definition 5. Internally, the plug-in converts the model into a WF-net and then checks relaxed soundness.

4.3 Process Discovery

One of the most challenging topics in process mining is the automated derivation of a model based on example traces [2]. The starting point for process discovery

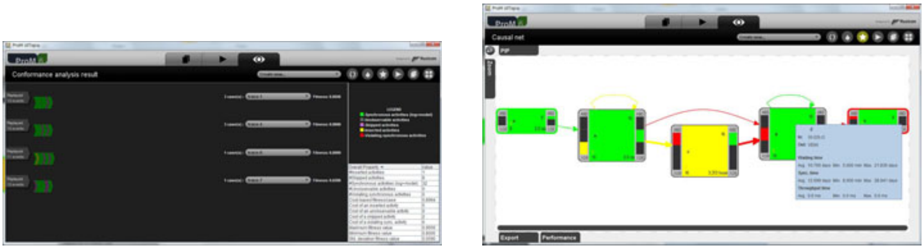
[3] Expressiveness in terms matching sequences.

is an event log in MXML or XES format. ProM provides a wide variety of process discovery techniques, e.g., techniques based on state-based region theory, language-based region theory, genetic mining, fuzzy mining, folding of partial orders, or heuristic mining. The process discovery plug-ins in ProM typically produce a Petri net or a model close to C-nets [2,7,8,9,10]. Using the various conversion plug-ins such results can be mapped onto C-nets.

What is missing are dedicated process discovery techniques producing C-nets while exploiting the representational bias. This is a topic for further research.

4.4 Conformance Checking and Performance Analysis

Given an event log and a process model, it is possible to replay the log on the model. ProM provides several plug-ins that replay logs on Petri nets. An example, is the "Conformance Checker" plug-in [2].



(a) Conformance analysis

(b) Performance analysis

Fig. 7. Two ProM plug-ins showing the results obtained through replaying the event log on a C-net

Recently, ProM started to support several plug-ins that replay logs on C-nets [4]. Figure 7(a) shows that ProM is able to discover deviations between a C-net and an event log. The plug-in indicates where deviations occur and what the overall fitness of the log is (using configurable cost functions). Most event logs contain timestamps. Therefore, replay can also be used to identify bottlenecks and to measure waiting and service times. Figure 7(b) shows the result of such analysis; the colors and numbers indicate different performance measurements.

5 Conclusion

This paper makes the case for Causal-nets (C-nets) in process mining. C-nets provide a better representational bias than conventional languages that are either too restrictive (e.g., OR-joins, unstructured loops, and skipping cannot be expressed) or too liberal (in the sense that most models are incorrect). Key ingredients are (1) the notion of bindings allowing for any split and join behavior and (2) the semantic restriction to valid binding sequences.

We explored the basic properties of C-nets and analyzed their relation to Petri nets. Moreover, we described the degree of support provided by ProM. Model management, conversion, verification, process discovery, conformance checking, and performance analysis of C-nets are supported by ProM 6 which can be downloaded from www.processmining.org.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
3. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. Formal Aspects of Computing 23(3), 333–363 (2011)
4. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Towards Robust Conformance Checking. In: Muehlen, M.z., Su, J. (eds.) BPM 2010. LNBIP, vol. 66, pp. 122–133. Springer, Heidelberg (2011)
5. Bruni, R., Montanari, U.: Zero-Safe Nets: Comparing the Collective and Individual Token Approaches. Information and Computation 156(1-2), 46–89 (2000)
6. Dehnert, J., van der Aalst, W.M.P.: Bridging the Gap Between Business Models and Workflow Specifications. International Journal of Cooperative Information Systems 13(3), 289–332 (2004)
7. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 328–343. Springer, Heidelberg (2007)
8. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
9. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
10. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible Heuristics Miner (FHM). BETA Working Paper Series, WP 334. Eindhoven University of Technology, Eindhoven (2010)
11. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94, 387–412 (2010)

On Causal Semantics of Petri Nets*
(Extended Abstract)

Rob J. van Glabbeek[1,2], Ursula Goltz[3], and Jens-Wolfhard Schicke[3]

[1] NICTA, Sydney, Australia
[2] School of Comp. Sc. and Engineering, Univ. of New South Wales, Sydney, Australia
[3] Institute for Programming and Reactive Systems, TU Braunschweig, Germany

rvg@cs.stanford.edu, goltz@ips.cs.tu-bs.de, drahflow@gmx.de

Abstract. We consider approaches for causal semantics of Petri nets, explicitly representing dependencies between transition occurrences. For one-safe nets or condition/event-systems, the notion of process as defined by Carl Adam Petri provides a notion of a run of a system where causal dependencies are reflected in terms of a partial order. A well-known problem is how to generalise this notion for nets where places may carry several tokens. Goltz and Reisig have defined such a generalisation by distinguishing tokens according to their causal history. However, this so-called *individual token interpretation* is often considered too detailed. A number of approaches have tackled the problem of defining a more abstract notion of process, thereby obtaining a so-called *collective token interpretation*. Here we give a short overview on these attempts and then identify a subclass of Petri nets, called *structural conflict nets*, where the interplay between conflict and concurrency due to token multiplicity does not occur. For this subclass, we define abstract processes as equivalence classes of Goltz-Reisig processes. We justify this approach by showing that we obtain exactly one maximal abstract process if and only if the underlying net is conflict-free with respect to a canonical notion of conflict.

1 Introduction

In this paper we address a well-known problem in Petri net theory, namely how to generalise Petri's concept of non-sequential processes to nets where places may carry multiple tokens.

One of the most interesting features of Petri nets is that they allow the explicit representation of causal dependencies between action occurrences when modelling reactive systems. This is a key difference with models of reactive systems (like standard transition systems) with an inherent so-called interleaving semantics, modelling concurrency by non-deterministic choice between sequential executions. In [GG01] it has been shown, using the model of event structures or configuration structures, that causal semantics are superior to interleaving semantics when giving up the assumption that actions are atomic entities.

* This work was partially supported by the DFG (German Research Foundation).

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 43–59, 2011.

In the following, we give a concise overview on existing approaches on semantics of Petri nets that give an account of their runs, without claiming completeness, and following closely a similar presentation in [GGS11a].

Initially, Petri introduced the concept of a net together with a definition of its dynamic behaviour in terms of the firing rule for single transitions or for finite sets (*steps*) of transitions firing in parallel. Sequences of transition firings or of steps are the usual way to define the behaviour of a Petri net. When considering only single transition firings, the set of all firing sequences yields a linear time interleaving semantics (no choices between alternative behaviours are represented). Otherwise we obtain a linear time step semantics, with information on possible parallelism, but without explicit representation of causal dependencies between transition occurrences.

Petri then defined *condition/event systems*, where — amongst other restrictions — places (there called conditions) may carry at most one token. For this class of nets, he proposed what is now the classical notion of a *process*, given as a mapping from an *occurrence net* (acyclic net with unbranched places) to the original net [Pet77,GSW80]. A process models a run of the represented system, obtained by choosing one of the alternatives in case of conflict. It records all occurrences of the transitions and places visited during such a run, together with the causal dependencies between them, which are given by the flow relation of the net. A linear-time causal semantics of a condition/event system is thus obtained by associating with a net the set of its processes. Depending on the desired level of abstraction, it may suffice to extract from each process just the partial order of transition occurrences in it. The firing sequences of transitions or steps can in turn be extracted from these partial orders. Nielsen, Plotkin and Winskel extended this to a branching-time semantics by using occurrence nets with forward branched places [NPW81]. These capture all runs of the represented system, together with the branching structure of choices between them.

However, the most frequently used class of Petri nets are nets where places may carry arbitrary many tokens, or a certain maximal number of tokens when adding place capacities. This type of nets is often called *place/transition systems* (P/T systems). Here tokens are usually assumed to be indistinguishable entities, for example representing a number of available resources in a system. Unfortunately, it is not straightforward to generalise the notion of process, as defined by Petri for condition/event systems, to P/T systems. In fact, it has now for more than 20 years been a well-known problem in Petri net theory how to formalise an appropriate causality-based concept of process or run for general P/T systems. In the following we give an introduction to the problem and a short overview on existing approaches.

As a first approach, Goltz and Reisig generalised Petri's notion of process to general P/T systems [GR83]. We call this notion of a process *GR-process*. It is based on a canonical unfolding of a P/T systems into a condition/event system, representing places that may carry several tokens by a corresponding number of conditions (see [Gol87]). Fig. 1 shows a P/T system with two of its GR-processes.

Engelfriet adapted GR-processes by additionally representing choices between alternative behaviours [Eng91], thereby adopting the approach of [NPW81]

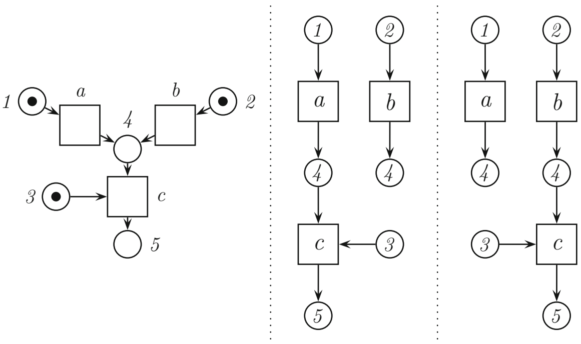


Fig. 1. A net N with its two maximal GR-processes. The correspondence between elements of the net and their occurrences in the processes is indicated by labels.

to P/T systems, although without arc weights. Meseguer, Sassone and Montanari extended this to cover also arc weights [MMS97].

However, if one wishes to interpret P/T systems with a causal semantics, there are alternative interpretations of what "causal semantics" should actually mean. Goltz already argued that when abstracting from the identity of multiple tokens residing in the same place, GR-processes do not accurately reflect runs of nets, because if a Petri net is conflict-free it should intuitively have only one complete run (for there are no choices to resolve), yet it may have multiple maximal GR-processes [Gol86]. This phenomenon already occurs in Fig. 1, since the choice between alternative behaviours is here only due to the possibility to choose between two tokens which can or even should be seen as indistinguishable entities. A similar argument is made, e.g., in [HKT95].

At the heart of this issue is the question whether multiple tokens residing in the same place should be seen as individual entities, so that a transition consuming just one of them constitutes a conflict, as in the interpretation underlying GR-processes and the approach of [Eng91,MMS97], or whether such tokens are indistinguishable, so that taking one is equivalent to taking the other. Van Glabbeek and Plotkin call the former viewpoint the *individual token interpretation* of P/T systems. For an alternative interpretation, they use the term *collective token interpretation* [GP95]. A possible formalisation of these interpretations occurs in [Gla05]. In the following we call process notions for P/T systems which are adherent to a collective token philosophy *abstract processes*. Another option, proposed by Vogler, regards tokens only as notation for a natural number stored in each place; these numbers are incremented or decremented when firing transitions, thereby introducing explicit causality between any transitions removing tokens from the same place [Vog91].

Mazurkiewicz applies again a different approach in [Maz89]. He proposes *multitrees*, which record possible multisets of fired transitions, and then takes confluent subsets of multitrees as abstract processes of P/T systems. This approach does not explicitly represent dependencies between transition occurrences

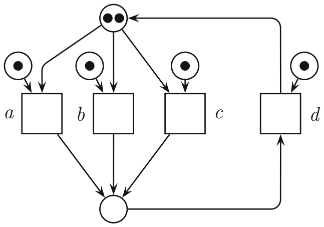


Fig. 2. A net with only a single process up to swapping equivalence

and hence does not apply to nets with self-loops, where such information may not always be retrieved.

Yet another approach has been proposed by Best and Devillers in [BD87]. Here an equivalence relation is generated by a transformation for changing causalities in GR-processes, called *swapping*, that identifies GR-processes which differ only in the choice which token was removed from a place. In this paper, we adopt this approach and we show that it yields a fully satisfying solution for a subclass of P/T systems. We call the resulting notion of a more abstract process *BD-process*. In the special case of one-safe P/T systems (where places carry at most one token), or for condition/event systems, no swapping is possible, and a BD-process is just an isomorphism class of GR-processes.

Meseguer and Montanari formalise runs in a net N as morphisms in a category $\mathcal{T}(N)$ [MM88]. In [DMM89] it has been established that these morphisms "coincide with the commutative processes defined by Best and Devillers" (their terminology for BD-processes). Likewise, Hoogers, Kleijn and Thiagarajan represent an abstract run of a net by a *trace*, thereby generalising the trace theory of Mazurkiewicz [Maz95], and remark that "it is straightforward but laborious to set up a 1-1 correspondence between our traces and the equivalence classes of finite processes generated by the swap operation in [Best and Devillers, 1987]".

To explain why it can be argued that BD-processes are not fully satisfying as abstract processes for general P/T systems, we recall in Fig. 2 an example due to Ochmański [Och89,BMO09], see also [DMM89,GGS11a]. In the initial situation only two of the three enabled transitions can fire, which constitutes a conflict. However, the equivalence obtained from the swapping transformation (formally defined in Section 3) identifies all possible maximal GR-processes and hence yields only one complete abstract run of the system. We are not aware of a solution, i.e. any formalisation of the concept of a run of a net that correctly represents both causality and parallelism of nets, and meets the requirement that for this net there is more than one possible complete run.

In [GGS11a] and in the present paper, we continue the line of research of [MM88,DMM89,Maz89,HKT95] to formalise a causality-based notion of an abstract process of a P/T system that fits a collective token interpretation. As remarked already in [Gol86], 'what we need is some notion of an "abstract process"' and 'a notion of maximality for abstract processes', such that 'a P/T-system is conflict-free iff it has exactly one maximal abstract process starting at the initial marking'. The example from Fig. 2 shows that BD-processes are in general not suited. We defined in [GGS11a] a subclass of P/T systems where conflict and concurrency are clearly separated. We called these nets *structural conflict nets*. Using the formalisation of conflict for P/T systems from [Gol86], we have shown that, for this subclass of P/T systems, we obtain more than one maximal BD-process whenever the net contains a conflict.[1] The proof of this result is quite involved; it was achieved by using an alternative characterisation of BD-processes via firing sequences from [BD87].

In this paper, we will show the reverse direction of this result, namely that we obtain exactly one maximal BD-process of a structural conflict net if the net is conflict-free. Depending on the precise formalisation of a suitable notion of maximality of BD-processes, this holds even for arbitrary nets. Summarising, we then have established that we obtain exactly one maximal abstract process in terms of BD-processes for structural conflict nets *if and only if* the net is conflict-free with respect to a canonical notion of conflict.

We proceed by defining basic notions for P/T systems in Section 2. In Section 3, we define GR-processes and introduce the swapping equivalence. Section 4 recalls the concept of conflict in P/T systems and defines structural conflict nets.[2] In Section 5, we recapitulate the alternative characterisation of BD-processes from [BD87] in terms of an equivalence notion on firing sequences [BD87] and prove in this setting that a conflict-free net has exactly one maximal run. Finally, in Section 6, we investigate notions of maximality for BD-processes and then transfer the result from Section 5 to BD-processes. Due to lack of space, the proofs of Lemma's 2, 3, 4, 6 and 7, some quite involved, are omitted; these can be found in [GGS11b].

2 Place Transition Systems

We will employ the following notations for multisets.

Definition 1. Let X be a set.

- A *multiset* over X is a function $A\colon X \rightarrow \mathbb{N}$, i.e. $A \in \mathbb{N}^X$.
- $x \in X$ is an *element of* A, notation $x \in A$, iff $A(x) > 0$.
- For multisets A and B over X we write $A \subseteq B$ iff $A(x) \leq B(x)$ for all $x \in X$; $A \cup B$ denotes the multiset over X with $(A \cup B)(x) := \max(A(x), B(x))$, $A \cap B$ denotes the multiset over X with $(A \cap B)(x) := \min(A(x), B(x))$, $A + B$ denotes the multiset over X with $(A + B)(x) := A(x) + B(x)$, $A - B$ is given by $(A - B)(x) := A(x) \dot{-} B(x) = \max(A(x) - B(x), 0)$, and for $k \in \mathbb{N}$ the multiset $k \cdot A$ is given by $(k \cdot A)(x) := k \cdot A(x)$.

[1] The notion of maximality for BD-processes is not trivial. However, with the results from Section 6, Corollary 1 from [GGS11a] may be rephrased in this way.

[2] The material in Sections 2 to 4 follows closely the presentation in [GGS11a], but needs to be included to make the paper self-contained.

- The function $\emptyset : X \rightarrow \mathbb{N}$, given by $\emptyset(x) := 0$ for all $x \in X$, is the *empty* multiset over X.
- If A is a multiset over X and $Y \subseteq X$ then $A \upharpoonright Y$ denotes the multiset over Y defined by $(A \upharpoonright Y)(x) := A(x)$ for all $x \in Y$.
- The cardinality $|A|$ of a multiset A over X is given by $|A| := \sum_{x \in X} A(x)$.
- A multiset A over X is *finite* iff $|A| < \infty$, i.e., iff the set $\{x \mid x \in A\}$ is finite.

Two multisets $A : X \rightarrow \mathbb{N}$ and $B : Y \rightarrow \mathbb{N}$ are *extensionally equivalent* iff $A \upharpoonright (X \cap Y) = B \upharpoonright (X \cap Y)$, $A \upharpoonright (X \setminus Y) = \emptyset$, and $B \upharpoonright (Y \setminus X) = \emptyset$. In this paper we often do not distinguish extensionally equivalent multisets. This enables us, for instance, to use $A \cup B$ even when A and B have different underlying domains. With $\{x, x, y\}$ we will denote the multiset over $\{x, y\}$ with $A(x)=2$ and $A(y)=1$, rather than the set $\{x, y\}$ itself. A multiset A with $A(x) \leq 1$ for all x is identified with the set $\{x \mid A(x) = 1\}$.

Below we define place/transition systems as net structures with an initial marking. In the literature we find slight variations in the definition of P/T systems concerning the requirements for pre- and postsets of places and transitions. In our case, we do allow isolated places. For transitions we allow empty postsets, but require at least one preplace, thus avoiding problems with infinite self-concurrency. Moreover, following [BD87], we restrict attention to nets of *finite synchronisation*, meaning that each transition has only finitely many pre- and postplaces. Arc weights are included by defining the flow relation as a function to the natural numbers. For succinctness, we will refer to our version of a P/T system as a *net*.

Definition 2.

A *net* is a tuple $N = (S, T, F, M_0)$ where

- S and T are disjoint sets (of *places* and *transitions*),
- $F : (S \times T \cup T \times S) \rightarrow \mathbb{N}$ (the *flow relation* including *arc weights*), and
- $M_0 : S \rightarrow \mathbb{N}$ (the *initial marking*)

such that for all $t \in T$ the set $\{s \mid F(s, t) > 0\}$ is finite and non-empty, and the set $\{s \mid F(t, s) > 0\}$ is finite.

Graphically, nets are depicted by drawing the places as circles and the transitions as boxes. For $x, y \in S \cup T$ there are $F(x, y)$ arrows (*arcs*) from x to y. When a net represents a concurrent system, a global state of this system is given as a *marking*, a multiset of places, depicted by placing $M(s)$ dots (*tokens*) in each place s. The initial state is M_0. The system behaviour is defined by the possible moves between markings M and M', which take place when a finite multiset G of transitions *fires*. When firing a transition, tokens on preplaces are consumed and tokens on postplaces are created, one for every incoming or outgoing arc of t, respectively. Obviously, a transition can only fire if all necessary tokens are available in M in the first place. Definition 4 formalises this notion of behaviour.

Definition 3. Let $N = (S, T, F, M_0)$ be a net and $x \in S \cup T$.

The multisets ${}^\bullet x,\ x^\bullet : S \cup T \to \mathbb{N}$ are given by ${}^\bullet x(y) = F(y, x)$ and $x^\bullet(y) = F(x, y)$ for all $y \in S \cup T$. If $x \in T$, the elements of ${}^\bullet x$ and $x^\bullet$ are called *pre-* and *postplaces* of x, respectively. These functions extend to multisets $X : S \cup T \to \mathbb{N}$ as usual, by ${}^\bullet X := \Sigma_{x \in S \cup T} X(x) \cdot {}^\bullet x$ and $X^\bullet := \Sigma_{x \in S \cup T} X(x) \cdot x^\bullet$.

Definition 4. Let $N = (S, T, F, M_0)$ be a net, $G \in \mathbb{N}^T$, G non-empty and finite, and $M, M' \in \mathbb{N}^S$.

G is a *step* from M to M', written $M \xrightarrow{G}_N M'$, iff

- ${}^\bullet G \subseteq M$ (G is *enabled*) and
- $M' = (M - {}^\bullet G) + G^\bullet$.

We may leave out the subscript N if clear from context. Extending the notion to words $\sigma = t_1 t_2 \dots t_n \in T^*$ we write $M \xrightarrow{\sigma} M'$ for

$$\exists M_1, M_2, \dots, M_{n-1}.\ M \xrightarrow{\{t_1\}} M_1 \xrightarrow{\{t_2\}} M_2 \cdots M_{n-1} \xrightarrow{\{t_n\}} M'.$$

When omitting σ or M' we always mean it to be existentially quantified. When $M_0 \xrightarrow{\sigma}_N$, the word σ is called a *firing sequence* of N. The set of all firing sequences of N is denoted by $\mathrm{FS}(N)$.

Note that steps are (finite) multisets, thus allowing self-concurrency. Also note that $M \xrightarrow{\{t,u\}}$ implies $M \xrightarrow{tu}$ and $M \xrightarrow{ut}$. We use the notation $t \in \sigma$ to indicate that the transition t occurs in the sequence σ, and $\sigma \leq \rho$ to indicate that σ is a prefix of the sequence ρ, i.e. $\exists \mu.\ \rho = \sigma\mu$.

3 Processes of Place/Transition Systems

We now define processes of nets. A (GR-)process is essentially a conflict-free, acyclic net together with a mapping function to the original net. It can be obtained by unwinding the original net, choosing one of the alternatives in case of conflict. The acyclic nature of the process gives rise to a notion of causality for transition firings in the original net via the mapping function. Conflicts present in the original net are represented by one net yielding multiple processes, each representing one possible way to decide the conflicts.

Definition 5.

A pair $P = (\mathcal{N}, \pi)$ is a *(GR-)process* of a net $N = (S, T, F, M_0)$ iff

- $\mathcal{N} = (\mathcal{S}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ is a net, satisfying
  - $\forall s \in \mathcal{S}\,.\ |{}^\bullet s| \leq 1 \geq |s^\bullet| \wedge \mathcal{M}_0(s) = \begin{cases} 1 & \text{if } {}^\bullet s = \emptyset \\ 0 & \text{otherwise,} \end{cases}$
  - $\mathcal{F}$ is acyclic, i.e. $\forall x \in \mathcal{S} \cup \mathcal{T}.\ (x, x) \notin \mathcal{F}^+$, where $\mathcal{F}^+$ is the transitive closure of $\{(t, u) \mid F(t, u) > 0\}$,
  - and $\{t \mid (t, u) \in \mathcal{F}^+\}$ is finite for all $u \in \mathcal{T}$.

- $\pi : \mathcal{S} \cup \mathcal{T} \rightarrow S \cup T$ is a function with $\pi(\mathcal{S}) \subseteq S$ and $\pi(\mathcal{T}) \subseteq T$, satisfying
  - $\pi(\mathcal{M}_0) = M_0$, i.e. $M_0(s) = |\pi^{-1}(s) \cap \mathcal{M}_0|$ for all $s \in S$, and
  - $\forall t \in \mathcal{T}, s \in S.\ F(s, \pi(t)) = |\pi^{-1}(s) \cap {}^\bullet t| \wedge F(\pi(t), s) = |\pi^{-1}(s) \cap t^\bullet|$.

P is called *finite* if $\mathcal{T}$ is finite.

The conditions for $\mathcal{N}$ ensure that a process is indeed a mapping from an occurrence net as defined in [Pet77,GSW80] to the net N; hence we define processes here in the classical way as in [GR83,BD87] (even though not introducing occurrence nets explicitly).

A process is not required to represent a completed run of the original net. It might just as well stop early. In those cases, some set of transitions can be added to the process such that another (larger) process is obtained. This corresponds to the system taking some more steps and gives rise to a natural order between processes.

Definition 6. Let $P = ((\mathcal{S}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0), \pi)$ and $P' = ((\mathcal{S}', \mathcal{T}', \mathcal{F}', \mathcal{M}_0'), \pi')$ be two processes of the same net.

- P' is a *prefix* of P, notation $P' \leq P$, and P an *extension* of P', iff $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{T}' \subseteq \mathcal{T}$, $\mathcal{M}_0' = \mathcal{M}_0$, $\mathcal{F}' = \mathcal{F} \restriction (\mathcal{S}' \times \mathcal{T}' \cup \mathcal{T}' \times \mathcal{S}')$ and $\pi' = \pi \restriction (\mathcal{S}' \times \mathcal{T}')$.
- A process of a net is said to be *maximal* if it has no proper extension.

The requirements above imply that if $P' \leq P$, $(x, y) \in \mathcal{F}^+$ and $y \in \mathcal{S}' \cup \mathcal{T}'$ then $x \in \mathcal{S}' \cup \mathcal{T}'$. Conversely, any subset $\mathcal{T}' \subseteq \mathcal{T}$ satisfying $(t, u) \in \mathcal{F}^+ \wedge u \in \mathcal{T}' \Rightarrow t \in \mathcal{T}'$ uniquely determines a prefix of P.

Two processes $(\mathcal{N}, \pi)$ and $(\mathcal{N}', \pi')$ are *isomorphic* iff there exists an isomorphism ϕ from $\mathcal{N}$ to $\mathcal{N}'$ which respects the process mapping, i.e. $\pi = \pi' \circ \phi$. Here an isomorphism ϕ between two nets $\mathcal{N} = (\mathcal{S}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0)$ and $\mathcal{N}' = (\mathcal{S}', \mathcal{T}', \mathcal{F}', \mathcal{M}_0')$ is a bijection between their places and transitions such that $\mathcal{M}_0'(\phi(s)) = \mathcal{M}_0(s)$ for all $s \in \mathcal{S}$ and $\mathcal{F}'(\phi(x), \phi(y)) = \mathcal{F}(x, y)$ for all $x, y \in \mathcal{S} \cup \mathcal{T}$.

Next we formally introduce the swapping transformation and the resulting equivalence notion on GR-processes from [BD87].

Definition 7. Let $P = ((\mathcal{S}, \mathcal{T}, \mathcal{F}, \mathcal{M}_0), \pi)$ be a process and let $p, q \in \mathcal{S}$ with $(p, q) \notin \mathcal{F}^+ \cup (\mathcal{F}^+)^{-1}$ and $\pi(p) = \pi(q)$.

Then $\text{swap}(P, p, q)$ is defined as $((\mathcal{S}, \mathcal{T}, \mathcal{F}', \mathcal{M}_0), \pi)$ with

$$\mathcal{F}'(x, y) = \begin{cases} \mathcal{F}(q, y) & \text{iff } x = p,\ y \in \mathcal{T} \\ \mathcal{F}(p, y) & \text{iff } x = q,\ y \in \mathcal{T} \\ \mathcal{F}(x, y) & \text{otherwise.} \end{cases}$$

Definition 8.

- Two processes P and Q of the same net are *one step swapping equivalent* ($P \approx_{\text{s}} Q$) iff $\text{swap}(P, p, q)$ is isomorphic to Q for some places p and q.

- We write $\approx_{\mathrm{s}}^*$ for the reflexive and transitive closure of $\approx_{\mathrm{s}}$, and $[\![P]\!]$ for the $\approx_{\mathrm{s}}^*$-equivalence class of a finite process P. The prefix relation $\leq$ between processes is lifted to such equivalence classes by $[\![P']\!] \leq [\![P]\!]$ iff $P' \approx_{\mathrm{s}}^* Q' \leq Q \approx_{\mathrm{s}}^* P$ for some Q', Q.
- Two processes P and Q are *swapping equivalent* ($P \approx_{\mathrm{s}}^\infty Q$) iff

$$\begin{aligned}&\downarrow(\{[\![P']\!] \mid P' \leq P,\ P' \text{ finite}\}) =\\ &\downarrow(\{[\![Q']\!] \mid Q' \leq Q,\ Q' \text{ finite}\})\end{aligned}$$

  where $\downarrow$ denotes prefix-closure under $\leq$.
- We call a $\approx_{\mathrm{s}}^\infty$-equivalence class of processes a *BD-process*, and write $[\![P]\!]_\infty$.

It is not hard to verify that if $P \approx_{\mathrm{s}}^* Q \leq Q'$ then $P \leq P' \approx_{\mathrm{s}}^* Q'$ for some process P'. This implies that $\leq$ is a partial order on $\approx_{\mathrm{s}}^*$-equivalence classes of finite processes. Alternatively, this conclusion follows from Theorem 4 in [GGS11a].

Our definition of $\approx_{\mathrm{s}}^\infty$ deviates from the definition of $\equiv_1^\infty$ from [BD87] to make proofs easier later on. We conjecture however that the two notions coincide.

Note that if $P \approx_{\mathrm{s}}^\infty Q$ and P is finite, then also Q is finite. Moreover, for finite GR-processes P and Q we have $P \approx_{\mathrm{s}}^\infty Q$ iff $P \approx_{\mathrm{s}}^* Q$. Thus, for a finite GR-process P, we have $[\![P]\!]_\infty = [\![P]\!]$. In that case we call $[\![P]\!]$ a *finite* BD-process.

We define a *BD-run* as a more abstract and more general form of BD-process. Like a BD-process, a BD-run is completely determined by its finite approximations, which are finite BD-processes; however, a BD-run does not require that these finite approximations are generated by a given GR-process.

Definition 9. Let N be a net.

A *BD-run* $\mathcal{R}$ of N is a non-empty set of finite BD-processes of N such that

- $[\![P]\!] \leq [\![Q]\!] \in \mathcal{R} \Rightarrow [\![P]\!] \in \mathcal{R}$ ($\mathcal{R}$ is prefix-closed), and
- $[\![P]\!], [\![Q]\!] \in \mathcal{R} \Rightarrow \exists\, [\![U]\!] \in \mathcal{R}.\ [\![P]\!] \leq [\![U]\!] \wedge [\![Q]\!] \leq [\![U]\!]$ ($\mathcal{R}$ is directed).

The class of finite BD-processes and the finite elements (in the set theoretical sense) in the class of BD-runs are in bijective correspondence. Every finite BD-run $\mathcal{R}$ must have a largest element, say $[\![P]\!]$, and the set of all prefixes of $[\![P]\!]$ is $\mathcal{R}$. Conversely, the set of prefixes of a finite BD-process $[\![P]\!]$ is a finite BD-run of which the largest element is again $[\![P]\!]$.

We now define a canonical mapping from GR-processes to BD-runs.

Definition 10. Let N be a net and P a process thereof.

Then $BD(P) := \downarrow\{[\![P']\!] \mid P' \leq P,\ P' \text{ finite}\}$.

Lemma 1. Let N be a net and P a process thereof.

Then $BD(P)$ is a BD-run.

Proof. See [GGS11a, Lemma 1]. □

This immediately yields an injective function from BD-processes to BD-runs, since by Definition 8, $P \approx_s^\infty Q$ iff $BD(P) = BD(Q)$. For countable nets (i.e. nets with countably many places and transitions), this function is even a bijection.

Lemma 2. Let N be a countable net and $\mathcal{R}$ a BD-run of N.
Then $\mathcal{R}$ is countable and there exists a process P of N such that $\mathcal{R} = BD(P)$.

Lemma 2 does not hold for uncountable nets, as witnessed by the counterexample in Fig. 3. This net N has a transition t for each real number $t \in \mathbb{R}$. Each such transition has a private preplace s_t with $M_0(s_t) = 1$ and $F(s_t, t) = 1$, which ensures that t can fire only once. Furthermore there is one shared place s with $M_0(s) = 2$ and a loop $F(s, t) = F(t, s) = 1$ for each transition t. There are no other places, transitions or arcs besides the ones mentioned above.

Each GR-process of N, and hence also each BD-process, has only countably many transitions. Yet, any two GR-processes firing the same finite set of transitions of N are swapping equivalent, and the set of all finite BD-processes of N constitutes a single BD-run involving all transitions.

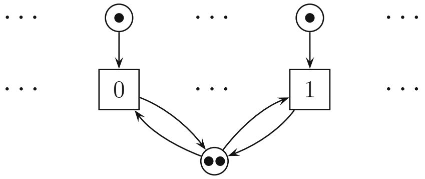


Fig. 3. A net with no maximal GR-process, but with a maximal BD-run.

We now show that the mapping BD respects the ordering of processes.

Lemma 3. Let N be a net, and P and P' two GR-processes of N.
If $P \leq P'$ then $BD(P) \subseteq BD(P')$.

4 Conflicts in Place/Transition Systems

We recall the canonical notion of conflict introduced in [Gol86].

Definition 11. Let $N = (S, T, F, M_0)$ be a net and $M \in \mathbb{N}^S$.

- A finite, non-empty multiset $G \in \mathbb{N}^T$ is in *(semantic) conflict* in M iff $(\forall t \in G.\ M \xrightarrow{G \upharpoonright \{t\}}) \wedge \neg M \xrightarrow{G}$.
- N is *(semantic) conflict-free* iff no finite, non-empty multiset $G \in \mathbb{N}^T$ is in semantic conflict in any M with $M_0 \longrightarrow M$.
- N is *binary-conflict–free* iff no multiset $G \in \mathbb{N}^T$ with $|G| = 2$ is in semantic conflict in any M with $M_0 \longrightarrow M$.

Remark: In a net (S, T, F, M_0) with $S = \{s\}$, $T = \{t, u\}$, $M_0(s) = 1$ and $F(s, t) = F(s, u) = 1$, the multiset $\{t, t\}$ is not enabled in M_0. For this reason the multiset $\{t, t, u\}$ does not count as being in conflict in M_0, even though it is not enabled. However, its subset $\{t, u\}$ is in conflict.

We proposed in [GGS11a] a class of P/T systems where the structural definition of conflict in terms of shared preplaces, as often used in Petri net theory, matches the semantic definition of conflict as given above. We called this class of nets *structural conflict nets*. For a net to be a structural conflict net, we require that two transitions sharing a preplace will never occur both in one step.

Definition 12. Let $N = (S, T, F, M_0)$ be a net.
N is a *structural conflict net* iff $\forall t, u.\ (M_0 \longrightarrow \xrightarrow{\{t,u\}}) \Rightarrow {}^\bullet t \cap {}^\bullet u = \emptyset$.

Note that this excludes self-concurrency from the possible behaviours in a structural conflict net: as in our setting every transition has at least one preplace, $t = u$ implies ${}^\bullet t \cap {}^\bullet u \neq \emptyset$. Also note that in a structural conflict net a non-empty, finite multiset G is in conflict in a marking M iff G is a set and two distinct transitions in G are in conflict in M. Hence a structural conflict net is conflict-free if and only if it is binary-conflict–free. Moreover, two transitions enabled in M are in (semantic) conflict iff they share a preplace.

5 A Conflict-Free Net Has Exactly One Maximal Run

In this section, we recapitulate results from [BD87], giving an alternative characterisation of runs of a net in terms of firing sequences. We use an adapted notation and terminology and a different treatment of infinite runs, as in [GGS11a]. As a main result of the present paper, we then prove in this setting that a conflict-free net has exactly one maximal run. In the following section, this result will be transferred to BD-processes.

The behaviour of a net can be described not only by its processes, but also by its firing sequences. The imposed total order on transition firings abstracts from information on causal dependence, or concurrency, between transition firings. To retrieve this information we introduce an *adjacency* relation on firing sequences, recording which interchanges of transition occurrences are due to semantic independence of transitions. Hence adjacent firing sequences represent the same run of the net. We then define *FS-runs* in terms of the resulting equivalence classes of firing sequences.

Definition 13. Let $N = (S, T, F, M_0)$ be a net, and $\sigma, \rho \in \mathrm{FS}(N)$.

- σ and ρ are *adjacent*, $\sigma \leftrightarrow \rho$, iff $\sigma = \sigma_1 t u \sigma_2$, $\rho = \sigma_1 u t \sigma_2$ and $M_0 \xrightarrow{\sigma_1} \xrightarrow{\{t,u\}}$.
- We write $\leftrightarrow^*$ for the reflexive and transitive closure of $\leftrightarrow$, and $[\sigma]$ for the $\leftrightarrow^*$-equivalence class of a firing sequence σ.

Note that $\leftrightarrow^*$-related firing sequences contain the same (finite) multiset of transition occurrences. When writing $\sigma \leftrightarrow^* \rho$ we implicitly claim that $\sigma, \rho \in \mathrm{FS}(N)$. Furthermore $\sigma \leftrightarrow^* \rho \wedge \sigma\mu \in \mathrm{FS}(N)$ implies $\sigma\mu \leftrightarrow^* \rho\mu$ for all $\mu \in T^*$.

The following definition introduces the notion of *partial* FS-run which is a formalisation of the intuitive concept of a finite, partial run of a net.

Definition 14. Let N be a net and $\sigma, \rho \in \mathrm{FS}(N)$.

- A *partial FS-run* of N is an $\leftrightarrow^*$-equivalence class of firing sequences.
- A partial FS-run $[\sigma]$ is a *prefix* of another partial FS-run $[\rho]$, notation $[\sigma] \leq [\rho]$, iff $\exists \mu.\ \sigma \leq \mu \leftrightarrow^* \rho$.

Note that $\sigma' \leftrightarrow^* \sigma \leq \mu$ implies $\exists \mu'.\ \sigma' \leq \mu' \leftrightarrow^* \mu$; thus the notion of prefix is well-defined, and a partial order.

Similar to the construction of BD-runs out of finite BD-processes, the following concept of an FS-run extends the notion of a partial FS-run to possibly infinite runs, in such a way that an FS-run is completely determined by its finite approximations.

Definition 15. Let N be a net.

An *FS-run* of N is a non-empty, prefix-closed and directed set of partial FS-runs of N.

There is a bijective correspondence between partial FS-runs and the finite elements in the class of FS-runs, just as in the case of BD-runs in Section 3. Much more interesting however is the following bijective correspondence between BD-runs and FS-runs.

Theorem 1. *There exists a bijective function Π from FS-runs to BD-runs such that $\Pi(\mathcal{R}) \subseteq \Pi(\mathcal{R}')$ iff $\mathcal{R} \subseteq \mathcal{R}'$.*

Proof. See [GGS11a], in particular the remarks at the end of Section 5. □

We now show that a conflict-free net has exactly one maximal run. As we have a bijective correspondence, it does not matter which notion of run we use here (FS-run or BD-run). We prove an even stronger result, using binary-conflict–free instead of conflict-free. In preparation we need the following lemma.

Lemma 4. *Let N be a binary-conflict–free net.*

If $\sigma, \sigma' \in \mathrm{FS}(N)$ then $\exists \mu, \mu'.\ \sigma\mu \in \mathrm{FS}(N) \wedge \sigma'\mu' \in \mathrm{FS}(N) \wedge \sigma\mu \leftrightarrow^* \sigma'\mu'$.

Theorem 2. *Let N be a binary-conflict–free net.*

There is exactly one maximal FS-run of N.

Proof. Let $\mathcal{R} = \{[\sigma] \mid \sigma$ is a finite firing sequence of $N\}$. We claim that $\mathcal{R}$ is said maximal FS-run of N.

First we show that $\mathcal{R}$ is prefix closed and directed, and thus indeed an FS-run.

Take any $[\rho] \leq [\sigma] \in \mathcal{R}$. Then by definition of $\leq$, $\exists \nu.\ \rho \leq \nu \wedge \nu \leftrightarrow^* \sigma$. We need to show that $[\rho] \in \mathcal{R}$, i.e. that ρ is a firing sequence of N. Since σ is a firing sequence of N and $\nu \leftrightarrow^* \sigma$, ν is also a firing sequence of N. Together with $\rho \leq \nu$ follows that ρ, too, is a firing sequence of N. Thus $\mathcal{R}$ is prefix closed.

To show directedness, let $[\sigma], [\rho] \in \mathcal{R}$. We need to show that $\exists [\nu] {\in} \mathcal{R}.\ [\sigma] \leq [\nu] \land [\rho] \leq [\nu]$, or with the definitions of $\leq$ and $[\ ]$ expanded, $\exists \nu.\ (\exists \alpha.\ \sigma \leq \alpha \leftrightarrow^* \nu \land \exists \beta.\ \rho \leq \beta \leftrightarrow^* \nu)$. We now apply Lemma 4 to $\sigma, \rho \in \mathrm{FS}(N)$, obtaining μ and μ' as mentioned in that lemma, and take $\alpha = \sigma\mu$ and $\beta = \rho\mu'$. Then Lemma 4 gives us $\alpha \leftrightarrow^* \beta$ and we take $\nu = \alpha$. Thus $\mathcal{R}$ is directed.

Finally we show that $\mathcal{R}$ is maximal. Take any run $\mathcal{R}'$ of N. Then $\mathcal{R}' \subseteq \mathcal{R}$ by definition of $\mathcal{R}$, hence $\mathcal{R}$ is maximal. □

6 BD-Processes Fit Structural Conflict Nets

In this section we show that BD-processes are adequate as abstract processes for the subclass of structural conflict nets.

In [GGS11a] we have shown that a semantic conflict in a structural conflict net always gives rise to multiple maximal GR-processes even up to swapping equivalence.

Theorem 3. Let N be a structural conflict net.

If N has only one maximal GR-process up to $\approx_{\mathrm{s}}^{\infty}$ then N is conflict-free.

Proof. Corollary 1 from [GGS11a]. □

We conjectured in [GGS11a] that, for countable nets, also the reverse direction holds, namely that a countable conflict-free structural conflict net has exactly one maximal GR-process up to $\approx_{\mathrm{s}}^{\infty}$.

In Section 5 we have already shown that a corresponding result holds for runs instead of processes. We will now transfer this result to BD-processes, and hence prove the conjecture.

We proceed by investigating three notions of maximality for BD-processes; they will turn out to coincide for structural conflict nets.

Definition 16.

- A BD-process $[\![P]\!]_\infty$ is *weakly maximal* (or a maximal GR-process up to $\approx_{\mathrm{s}}^{\infty}$), iff some $P' \in [\![P]\!]_\infty$ is maximal (in the GR-process sense).
- A BD-process $[\![P]\!]_\infty$ is *maximal* iff $\forall P' \in [\![P]\!]_\infty\ \forall Q.\ (P' \leq Q \Rightarrow P' \approx_{\mathrm{s}}^{\infty} Q)$.
- A BD-process $[\![P]\!]_\infty$ is *run-maximal* iff the BD-run $BD(P)$ is maximal.

The first notion is the simplest way of inheriting the notion of maximality of GR-process by BD-processes, whereas the last one inherits the notion of maximality from BD-runs. The middle notion is the canonical notion of maximality with respect to a natural order on BD-process, defined below.

Definition 17. Let N be a net.

We define a relation $\preceq$ between BD-processes, via

$$[\![P]\!]_\infty \preceq [\![Q]\!]_\infty :\Leftrightarrow \exists P' \approx_{\mathrm{s}}^{\infty} P\ \exists Q' \approx_{\mathrm{s}}^{\infty} Q.\ P' \leq Q'\ ,$$

and construct an order between BD-processes via

$$[\![P]\!]_\infty \leq [\![Q]\!]_\infty :\Leftrightarrow [\![P]\!]_\infty \preceq^{+} [\![Q]\!]_\infty\ .$$

By construction, the relation $\leq$ is reflexive and transitive (even though $\preceq$ in general is not transitive). Lemma 3 yields that it also is antisymmetric, and hence a partial order. Namely, if $[\![P]\!]_\infty \leq [\![Q]\!]_\infty$ and $[\![Q]\!]_\infty \leq [\![P]\!]_\infty$, then $BD(P) = BD(Q)$, so $P \approx_s^\infty Q$, implying $[\![P]\!]_\infty = [\![Q]\!]_\infty$.

Now maximality according to Definition 16 is simply maximality w.r.t. $\leq$:

$$[\![P]\!]_\infty \text{ is maximal iff } \nexists [\![P']\!]_\infty.\ [\![P]\!]_\infty \leq [\![P']\!]_\infty \wedge [\![P]\!]_\infty \neq [\![P']\!]_\infty.$$

The following lemma tells how the above notions of maximality form a hierarchy.

Lemma 5. Let N be a net and P a process thereof.

1. If $[\![P]\!]_\infty$ is run-maximal, it is maximal.
2. If $[\![P]\!]_\infty$ is maximal, it is weakly maximal.

Proof. "1": This follows since $[\![P]\!]_\infty \leq [\![Q]\!]_\infty \Rightarrow BD(P) \subseteq BD(Q)$ by Lemma 3.

"2": Assume $[\![P]\!]_\infty$ is maximal. By Lemma 2 in [GGS11a], which follows via Zorn's Lemma, there exists some maximal Q with $P \leq Q$. Since $[\![P]\!]_\infty$ is maximal we have $Q \approx_s^\infty P$ and Q is a maximal process within $[\![P]\!]_\infty$. □

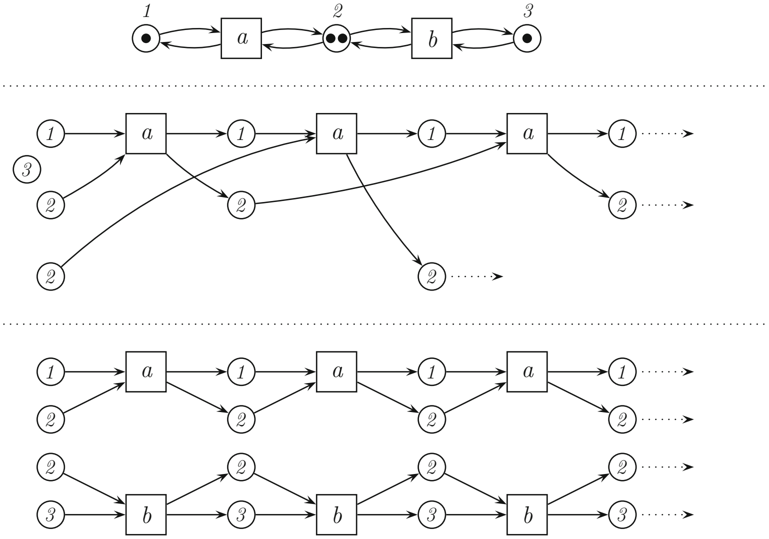


Fig. 4. A net and two weakly maximal processes thereof

The three notions of maximality are all distinct. The first process depicted in Fig. 4 is an example of a weakly maximal BD-process that is not maximal. Namely, the process itself cannot be extended (for none of the tokens in place 2 will in the end come to rest), but the process is swapping equivalent with the top half of the second process (using only one of the tokens in place 2), which can be extended with the bottom half.

The process depicted in Fig. 5 is an example of a BD-process $[\![P]\!]_\infty$ which is maximal, but not run-maximal. It is maximal, because no matter how it is

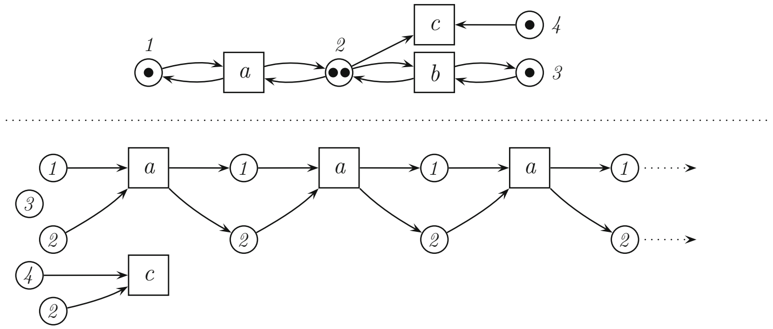


Fig. 5. A net and a maximal process thereof

swapped, at some point the c-transition will fire, and after that the only token left in place 2 will be in use forever, making it impossible to extend the process with any (b-)transition. It is not run-maximal, as the set of all finite processes of N constitutes a larger run. Note that every two finite processes of N mapping to the same multiset of transitions are swapping equivalent.

The following lemmas show that for countable conflict-free nets maximality and run-maximality coincide, and that for structural conflict nets all three notions of maximality coincide.

Lemma 6. Let N be a countable binary-conflict–free net, and P be a GR-process of N.

If $[\![P]\!]_\infty$ is maximal, then $[\![P]\!]_\infty$ is run-maximal.

Lemma 7. Let N be a structural conflict net, and P be a GR-process of N.

If $[\![P]\!]_\infty$ is weakly maximal, then $[\![P]\!]_\infty$ is run-maximal.

Finally, we are able to show, using Theorem 2, that a countable, binary-conflict–free net has only one maximal BD-process. In case of a conflict-free structural conflict net we can do the stronger statement that it has only one weakly maximal BD-process, i.e. only one GR-process up to swapping equivalence.

Lemma 8. Let N be a binary-conflict–free net.

(1) N has at most one run-maximal BD-process.
(2) If N moreover is countable, then it has exactly one run-maximal BD-process.

Proof. Suppose N had two run-maximal BD-processes $[\![P]\!]_\infty$ and $[\![P']\!]_\infty$. Then $BD(P)$ and $BD(P')$ are maximal BD-runs. By Theorem 2 N has only one maximal BD-run. Hence $BD(P) = BD(P')$ and thus $[\![P]\!]_\infty = [\![P']\!]_\infty$.

Now assume that N is countable. By Theorem 2, N has a maximal BD-run $\mathcal{R}$. By Lemma 2 there is a process P with $BD(P) = \mathcal{R}$. By Definition 16 $[\![P]\!]_\infty$ is run-maximal, so at least one run-maximal BD-process exists. □

Theorem 4. Let N be a countable binary-conflict–free net.

N has exactly one maximal BD-process.

Proof. By Lemmas 5 and 6 the notions of maximality and run-maximality coincide for N, and the result follows from Lemma 8. □

The net of Fig. 3 is an example of an uncountable binary-conflict–free net without any maximal or run-maximal BD-process.

Theorem 5. Let N be a conflict-free structural conflict net.

N has exactly one weakly maximal BD-process, i.e. exactly one maximal GR-process up to $\approx_s^\infty$.

Proof. By Lemmas 5 and 7 the three maximality notions coincide for N, and the "at most one"-direction follows from Lemma 8.

Surely, N has at least one process (with an empty set of transitions). By Lemma 2 in [GGS11a], which in turn invokes Zorn's lemma, every GR-process is a prefix of a maximal GR-process. Hence N has a maximal GR-process, and thus a maximal GR-process up to $\approx_s^\infty$. □

The assumption that N is a structural conflict net is essential in Theorem 5. The net in Fig. 4 is countable (even finite) and conflict-free, yet has multiple maximal GR-process up to $\approx_s^\infty$.

We can now justify BD-processes as an abstract notion of process for structural conflict nets since we obtain exactly one maximal abstract process if and only if the underlying net is conflict-free.

Corollary 1. Let N be a structural conflict net.

N is conflict-free iff N has exactly one maximal BD-process, which is the case iff N has exactly one maximal GR-process up to $\approx_s^\infty$.

Proof. All three notions of maximality coincide for structural conflict nets according to Lemma 7 and Lemma 5.

"⇒": By Theorem 5.

"⇐": By Theorem 3. □

References

[BD87] Best, E., Devillers, R.R.: Sequential and concurrent behaviour in Petri net theory. Theoretical Computer Science 55(1), 87–136 (1987), doi:10.1016/0304-3975(87)90090-9; See also: Best, E., Devillers, R.R.: Interleaving and Partial Orders in Concurrency: A Formal Comparison. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III, pp. 299–321. North-Holland, Amsterdam (1987)

[BMO09] Barylska, K., Mikulski, Ł., Ochmański, E.: Nonviolence Petri Nets. In: Proceedings Workshop on Concurrency, Specification and Programming, CS&P, pp. 50–59 (2009)

[DMM89] Degano, P., Meseguer, J., Montanari, U.: Axiomatizing Net Computations and Processes. In: Proceedings Fourth Annual Symposium on Logic in Computer Science, LICS 1989, pp. 175–185. IEEE, Pacific Grove (1989); see also Degano, P., Meseguer, J., Montanari, U.: Axiomatizing Net Computations and Processes. Acta Informatica 33(5), 641–667 (1996), doi:10.1007/BF03036469

[Eng91] Engelfriet, J.: Branching Processes of Petri Nets. Acta Informatica 28(6), 575–591 (1991)

[GG01] van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Informatica 37(4/5), 229–327 (2001), doi:10.1007/s002360000041

[GGS11a] van Glabbeek, R.J., Goltz, U., Schicke, J.-W.: Abstract Processes of Place/Transition Systems. Information Processing Letters 111(13), 626–633 (2011), doi:10.1016/j.ipl.2011.03.013

[GGS11b] van Glabbeek, R.J., Goltz, U., Schicke, J.-W.: On Causal Semantics of Petri Nets. Technical Report 2011-06, TU Braunschweig (2011)

[Gla05] van Glabbeek, R.J.: The Individual and Collective Token Interpretations of Petri Nets. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 323–337. Springer, Heidelberg (2005), doi:10.1007/11539452_26

[Gol86] Goltz, U.: How Many Transitions may be in Conflict? Petri Net Newsletter 25, 4–9 (1986)

[Gol87] Goltz, U.: On condition/event representations of place/transition nets. In: Concurrency and Nets: Advances in Petri Nets. LNCS, pp. 217–231. Springer, Heidelberg (1987)

[GP95] van Glabbeek, R.J., Plotkin, G.D.: Configuration Structures (extended abstract). In: Kozen, D. (ed.) Proceedings LICS 1995, pp. 199–209 (1995); See also van Glabbeek, R.J., Plotkin, G.D.: Configuration Structures, Event Structures and Petri Nets. Theoretical Computer Science 410(41), 4111–4159, doi:10.1016/j.tcs.2009.06.014 (2009)

[GR83] Goltz, U., Reisig, W.: The Non-Sequential Behaviour of Petri Nets. Information and Control 57(2-3), 125–147 (1983)

[GSW80] Genrich, H.J., Stankiewicz-Wiechno, E.: A Dictionary of Some Basic Notions of Net Theory. In: Brauer, W. (ed.) Advanced Course: Net Theory and Applications. LNCS, vol. 84, pp. 519–531. Springer, Heidelberg (1980)

[HKT95] Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: A Trace Semantics for Petri Nets. Information and Computation 117, 98–114 (1995)

[Maz89] Mazurkiewicz, A.W.: Concurrency, Modularity, and Synchronization. In: Kreczmar, A., Mirkowska, G. (eds.) MFCS 1989. LNCS, vol. 379, pp. 577–598. Springer, Heidelberg (1989)

[Maz95] Mazurkiewicz, A.W.: Introduction to Trace Theory. In: Diekert, V., Rozenberg, G. (eds.) The Book of Traces, pp. 3–41. World Scientific, Singapore (1995)

[MM88] Meseguer, J., Montanari, U.: Petri Nets Are Monoids: A New Algebraic Foundation for Net theory. In: Proceedings Third Annual Symposium on Logic in Computer Science, LICS 1988, Edinburgh, Scotland, pp. 155–164. IEEE, Los Alamitos (1988)

[MMS97] Meseguer, J., Montanari, U., Sassone, V.: On the Semantics of Place/Transition Petri Nets. Mathematical Structures in Computer Science 7(4), 359–397 (1997)

[NPW81] Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13(1), 85–108 (1981), doi:10.1016/0304-3975(81)90112-2

[Och89] Ochmański, E.: Personal communication (1989)

[Pet77] Petri, C.A.: Non-sequential Processes. GMD-ISF Report 77.05, GMD (1977)

[Vog91] Vogler, W.: Executions: a new partial-order semantics of Petri nets. Theoretical Computer Science 91(2), 205–238 (1991), doi:10.1016/0304-3975(91)90084-F

On Expressive Powers of Timed Logics: Comparing Boundedness, Non-punctuality, and Deterministic Freezing

Paritosh K. Pandya and Simoni S. Shah

Tata Institute of Fundamental Research, Colaba, Mumbai 400005, India

Abstract. Timed temporal logics exhibit a bewildering diversity of operators and the resulting decidability and expressiveness properties also vary considerably. We study the expressive power of timed logics $TPTL[\mathsf{U},\mathsf{S}]$ and $MTL[\mathsf{U}_I,\mathsf{S}_I]$ as well as of their several fragments. Extending the LTL EF games of Etessami and Wilke, we define MTL Ehrenfeucht-Fraïssé games on a pair of timed words. Using the associated EF theorem, we show that, expressively, the timed logics $BoundedMTL[\mathsf{U}_I,\mathsf{S}_I]$, $MTL[\mathsf{F}_I,\mathsf{P}_I]$ and $MITL[\mathsf{U}_I,\mathsf{S}_I]$ (respectively incorporating the restrictions of boundedness, unary modalities and non-punctuality), are all pairwise incomparable. As our first main result, we show that $MTL[\mathsf{U}_I,\mathsf{S}_I]$ is *strictly contained* within the freeze logic $TPTL[\mathsf{U},\mathsf{S}]$ for both weakly and strictly monotonic timed words, thereby extending the result of Bouyer *et al* and completing the proof of the original conjecture of Alur and Henziger from 1990. We also relate the expressiveness of a recently proposed deterministic freeze logic $TTL[X_\theta,Y_\theta]$ (with NP-complete satisfiability) to MTL. As our second main result, we show by an explicit reduction that $TTL[X_\theta,Y_\theta]$ lies strictly within the unary, non-punctual logic $MITL[\mathsf{F}_I,\mathsf{P}_I]$. This shows that deterministic freezing with punctuality is expressible in the non-punctual $MITL[\mathsf{F}_I,\mathsf{P}_I]$.

1 Introduction

Temporal logics are well established formalisms for specifying qualitative ordering constraints on the sequence of observable events. Real-time temporal logics extend this vocabulary with specification of quantitative timing constraints between these events.

There are two well-established species of timed logics with linear time. The logic $TPTL[\mathsf{U},\mathsf{S}]$ makes use of freeze quantification together with untimed temporal modalities and explicit constraints on frozen time values; the logic $MTL[\mathsf{U}_I,\mathsf{S}_I]$ uses time interval constrained modalities U_I and S_I. For example,the $TPTL[\mathsf{U},\mathsf{S}]$ formula $x.(a\mathsf{U}(b \wedge T - x < 2))$ and the $MTL[\mathsf{U}_I,\mathsf{S}_I]$ formula $a\mathsf{U}_{[0,2)}b$ both characterize the set of words that have a letter b with time stamp < 2 where this b is preceded only by a string of letters a. Timed logics may be defined over timed words (also called pointwise time models) or over signals (also called continuous time models). Weak monotonicity (as against strict monotonicity) allows a sequence of events to occur at the same time point. In this paper we confine ourselves to finite timed words with both weakly and strictly monotonic time, but the results straightforwardly carry over to infinite words too.

In their pioneering studies [1, 3, 4], Alur and Henzinger investigated the expressiveness and decidability properties of timed logics $MTL[\mathsf{U}_I,\mathsf{S}_I]$ and $TPTL[\mathsf{U},\mathsf{S}]$. They

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 60–75.

showed that $MTL[\mathsf{U}_I,\mathsf{S}_I]$ can be easily translated into $TPTL[\mathsf{U},\mathsf{S}]$. Further, they conjectured, giving an intuitive example, that $TPTL[\mathsf{U},\mathsf{S}]$ is more expressive than $MTL[\mathsf{U}_I,\mathsf{S}_I]$ (see [3] section 4.3). Fifteen years later, in a seminal paper, Bouyer *et al* [6] formally proved that the purely future time logic $TPTL[\mathsf{U}]$ is strictly more expressive than $MTL[\mathsf{U}_I]$ and that $MTL[\mathsf{U}_I,\mathsf{S}_I]$ is more expressive than $MTL[\mathsf{U}_I]$, for both pointwise and continuous time. In this paper, we complete the picture by proving the original conjecture of Alur and Henzinger for the full logic $MTL[\mathsf{U}_I,\mathsf{S}_I]$ with both future and past over pointwise time.

In their full generality, $MTL[\mathsf{U}_I,\mathsf{S}_I]$ and $TPTL[\mathsf{U},\mathsf{S}]$ are both undecidable even for finite timed words. Several restrictions have been proposed to get decidable sub-logics (see [12] for a recent survey). Thus, Bouyer *et al.* [7] introduced $BoundedMTL[\mathsf{U}_I,\mathsf{S}_I]$ with "bounded" intervals and showed that its satisfiability is *EXPSPACE*-complete. Alur and Henzinger argued, using reversal bounded 2-way deterministic timed automata *RB2DTA*, that the logic $MITL[\mathsf{U}_I,\mathsf{S}_I]$ permitting only non-singular (or non-punctual) intervals was decidable with *EXPSPACE* complexity [2,5]. Unary modalities have played a special role in untimed logics [9], and we also consider unary fragments $MTL[\mathsf{F}_I,\mathsf{P}_I]$ and $TPTL[\mathsf{F},\mathsf{P}]$ in our study. Further sub-classes can be obtained by combining the restrictions of bounded or non singular intervals and unary modalities.

In this paper, we mainly compare the expressive powers of various real-time temporal logics. As our main tool we define an *m-round MTL EF game* with "until" and "since" moves on two given timed words. As usual, the EF theorem equates the inability of any $MTL[\mathsf{U}_I,\mathsf{S}_I]$ formula with modal depth m from distinguishing two timed words to the existence of a winning strategy for the duplicator in m-round games. Our EF theorem is actually parametrized by a permitted set of time intervals, and it can be used for proving the lack of expressiveness of various fragments of $MTL[\mathsf{U}_I,\mathsf{S}_I]$.

Classically, the EF Theorem has been a useful tool for proving limitations in expressive power of first-order logic [11,16]. In their well-known paper, Etessami and Wilke [10] adapted this to the LTL EF games to show the existence of the "until" hierarchy in LTL definable languages. Our *MTL* EF theorem is a generalization of this to the timed setting. We find that the use of EF theorem often leads to simple game theoretic proofs of seemingly difficult questions about expressiveness of timed logics. The paper contains several examples of such proofs.

Our main expressiveness results are as follows. We show these results for finite timed words with weakly and strictly monotonic time. However, we remark that these results straightforwardly carry over to infinite timed words.

- We show that logics $BoundedMTL[\mathsf{U}_I,\mathsf{S}_I]$, $MITL[\mathsf{U}_I,\mathsf{S}_I]$ and $MTL[\mathsf{F}_I,\mathsf{P}_I]$ are all pairwise incomparable. These results indicate that the restrictions of boundedness, non-punctuality, and unary modalities are all semantically "orthogonal" in context of *MTL*.
- As one of our main results, we show that the unary and future fragment $TPTL[\mathsf{F}]$ of the freeze logic $TPTL[\mathsf{U},\mathsf{S}]$ is not expressively contained within $MTL[\mathsf{U}_I,\mathsf{S}_I]$ for both strictly monotonic and weakly monotonic timed words. Thus, $MTL[\mathsf{U}_I,\mathsf{S}_I]$ is a strict subset of $TPTL[\mathsf{U},\mathsf{S}]$ for pointwise time, as originally conjectured by Alur and Henzinger almost 20 years ago [1,3,6].

- It is easy to show that for strictly monotonic timed words, logic $\mathit{TPTL}[\mathsf{U},\mathsf{S}]$ can be translated to the unary fragment $\mathit{TPTL}[\mathsf{F},\mathsf{P}]$ and for expressiveness the two logics coincide. For weakly monotonic time, we show that $\mathit{MTL}[\mathsf{U}_I,\mathsf{S}_I]$ and $\mathit{TPTL}[\mathsf{F},\mathsf{P}]$ are expressively incomparable.

In the second part of this paper, we explore the expressiveness of a recently proposed "deterministic" and "unary" fragment of $\mathit{TPTL}[\mathsf{F},\mathsf{P}]$ called $\mathit{TTL}[X_\theta,Y_\theta]$. This is an interesting logic with exact automaton characterization as partially ordered two way deterministic timed automata [13]. Moreover, by exploiting the properties of these automata, the logic has been shown to have NP-complete satisfiability. The key feature of this logic is the "unique parsing" of each timed word against a given formula. Our main results on the expressiveness of $\mathit{TTL}[X_\theta,Y_\theta]$ are as follows.

- By an explicit reduction, we show that $\mathit{TTL}[X_\theta,Y_\theta]$ is contained within the unary and non-punctual logic $\mathit{MITL}[\mathsf{F}_I,\mathsf{P}_I]$. The containment holds in spite of the fact that $\mathit{TTL}[X_\theta,Y_\theta]$ can have freeze quantification and punctual constraints (albeit only occurring deterministically).
- Using the unique parsability of $\mathit{TTL}[X_\theta,Y_\theta]$, we show that neither $\mathit{MITL}[\mathsf{F}_I,\mathsf{P}_I]$ nor $\mathit{BoundedMTL}[\mathsf{F}_I,\mathsf{P}_I]$ are expressively contained within $\mathit{TTL}[X_\theta,Y_\theta]$.

Thus, the full logic $\mathit{TPTL}[\mathsf{U},\mathsf{S}]$ is more expressive than $\mathit{MTL}[\mathsf{U}_I,\mathsf{S}_I]$. But its unary fragment with deterministic freezing, $\mathit{TTL}[X_\theta,Y_\theta]$, lies strictly within the unary and non-punctual logic $\mathit{MITL}[\mathsf{F}_I,\mathsf{P}_I]$. Figure 1 provides a succinct pictorial representation of all the expressiveness results achieved.

The rest of the paper is organized as follows. Section 2 defines various timed logics. The MTL EF games and the EF Theorem are given in Section 3. Section 4 explores the relative expressiveness of various fragments of $\mathit{MTL}[\mathsf{U}_I,\mathsf{S}_I]$ and the subsequent section compares $\mathit{TPTL}[\mathsf{U},\mathsf{S}]$ to $\mathit{MTL}[\mathsf{U}_I,\mathsf{S}_I]$. Section 6 studies the expressiveness of $\mathit{TTL}[X_\theta,Y_\theta]$ relative to sub logics of $\mathit{MTL}[\mathsf{U}_I,\mathsf{S}_I]$.

2 Timed Temporal Logics: Syntax and Semantics

We provide a brief introduction of the logics whose expressiveness is investigated in this paper.

2.1 Preliminaries

Let $\mathbb{R},\mathbb{Z}$ and $\mathbb{N}$ be the set of reals, rationals, integers, and natural numbers, respectively and $\mathbb{R}_0$ be the set of non-negative reals. An *interval* is a convex subset of $\mathbb{R}_0$, bounded by non-negative integer constants or ∞. The left and right ends of an interval may be open ("(" or ")") or closed ("[" or "]"). We denote by $\langle x,y\rangle$ a generic interval whose ends may be open or closed. An interval is said to be *bounded* if it does not extend to infinity. It is said to be *singular* if it is of the form $[c,c]$ for some constant c, and non-singular (or non-punctual) otherwise. We denote by $\mathbb{Z}I$ all the intervals (including singular intervals $[c,c]$ and unbounded intervals $[c,\infty)$), by $\mathbb{Z}\mathit{IExt}$ the set of all non-punctual (or extended) intervals, and by $\mathit{Bd}\mathbb{Z}I$ the set of all bounded intervals. Given

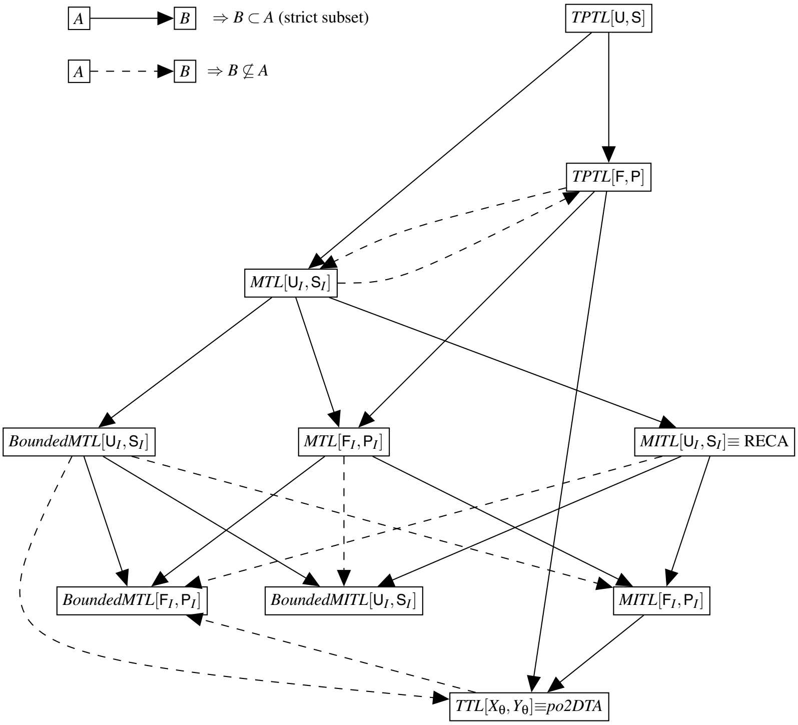

A → B $\Rightarrow B \subset A$ (strict subset)
A - - → B $\Rightarrow B \not\subseteq A$
$TPTL[\mathsf{U},\mathsf{S}]$
$TPTL[\mathsf{F},\mathsf{P}]$
$MTL[\mathsf{U}_I,\mathsf{S}_I]$
$BoundedMTL[\mathsf{U}_I,\mathsf{S}_I]$
$MTL[\mathsf{F}_I,\mathsf{P}_I]$
$MITL[\mathsf{U}_I,\mathsf{S}_I] \equiv$ RECA
$BoundedMTL[\mathsf{F}_I,\mathsf{P}_I]$
$BoundedMITL[\mathsf{U}_I,\mathsf{S}_I]$
$MITL[\mathsf{F}_I,\mathsf{P}_I]$
$TTL[X_\theta,Y_\theta] \equiv po2DTA$

Fig. 1. Expressiveness of Timed Logics for Pointwise Time

an alphabet Σ, its elements are used also as atomic propositions in logic, i.e. the set of atomic propositions $AP = \Sigma$.

A finite timed word is a finite sequence $\rho = (\sigma_1,\tau_1),(\sigma_2,\tau_2),\cdots,(\sigma_n,\tau_n)$, of event-time stamp pairs such that the sequence of time stamps is non-decreasing: $\forall i < n \,.\, \tau_i \leq \tau_{i+1}$. This gives weakly monotonic timed words. If time stamps are strictly increasing, i.e. $\forall i < n \,.\, \tau_i < \tau_{i+1}$, the word is strictly monotonic. The length of ρ is denoted by $\#\rho$, and $dom(\rho) = \{1,...\#\rho\}$. For convenience, we assume that $\tau_1 = 0$ as this simplifies the treatment of "freeze" logics. The timed word ρ can alternately be represented as $\rho = (\overline{\sigma},\overline{\tau})$ with $\overline{\sigma} = \sigma_1,\cdots,\sigma_n$ and $\overline{\tau} = \tau_1,\cdots,\tau_n$. Let $untime(\rho) = \overline{\sigma}$. We shall use the two representations interchangeably. Let $T\Sigma^*$ be the set of timed words over the alphabet Σ.

2.2 Metric Temporal Logics

The logic MTL extends Linear Temporal Logic by adding timing constraints to the "Until" and "Since" modalities of LTL. We parametrize this logic by a permitted set of intervals Iv and denote the resulting logic as $IvMTL[\mathsf{U}_I,\mathsf{S}_I]$. Let ϕ range over $IvMTL[\mathsf{U}_I,\mathsf{S}_I]$ formulas, $a \in \Sigma$ and $I \in Iv$. The syntax of $IvMTL[\mathsf{U}_I,\mathsf{S}_I]$ is as follows:

$$\phi ::= a \mid \phi \wedge \phi \mid \neg\phi \mid \phi\mathsf{U}_I\phi \mid \phi\mathsf{S}_I\phi$$

Let $\rho = (\overline{\sigma}, \overline{\tau})$ be a timed word and let $i \in dom(\rho)$. The semantics of $MTL[\mathsf{U}_I, \mathsf{S}_I]$ formulas is as below:

$$\begin{array}{rll}
\rho, i \models a & \text{iff} & \sigma_i = a \\
\rho, i \models \neg\phi & \text{iff} & \rho, i \not\models \phi \\
\rho, i \models \phi_1 \vee \phi_2 & \text{iff} & \rho, i \models \phi_1 \text{ or } \rho, i \models \phi_2 \\
\rho, i \models \phi_1 \mathsf{U}_I \phi_2 & \text{iff} & \exists j > i.\ \rho, j \models \phi_2 \text{ and } \tau_j - \tau_i \in I \\
& & \text{and } \forall i < k < j.\ \rho, k \models \phi_1 \\
\rho, i \models \phi_1 \mathsf{S}_I \phi_2 & \text{iff} & \exists j < i\ .\ \rho, j \models \phi_2 \text{ and } \tau_i - \tau_j \in I \\
& & \text{and } \forall j < k < i.\ \rho, k \models \phi_1
\end{array}$$

The language of an $IvMTL[\mathsf{U}_I, \mathsf{S}_I]$ formula ϕ is given by $\mathcal{L}(\phi) = \{\rho \mid \rho, 1 \models \phi\}$. Note that we use the "strict" semantics of U_I and S_I modalities. We can define unary *"future"* and *"past"* modalities as: $\mathsf{F}_I\phi := \top \mathsf{U}_I \phi$ and $\mathsf{P}_I\phi := \top \mathsf{S}_I \phi$. The subset of $IvMTL[\mathsf{U}_I, \mathsf{S}_I]$ using only these modalities is called $IvMTL[\mathsf{F}_I, \mathsf{P}_I]$. We can now define various well known variants of MTL.

- Metric Temporal Logic [1,3], denoted $MTL[\mathsf{U}_I, \mathsf{S}_I] = \mathbb{Z}IMTL[\mathsf{U}_I, \mathsf{S}_I]$. This is obtained by choosing the set of intervals $Iv = \mathbb{Z}I$.
- Unary MTL, denoted $MTL[\mathsf{F}_I, \mathsf{P}_I]$= $\mathbb{Z}IMTL[\mathsf{F}_I, \mathsf{P}_I]$ uses only unary modalities. It is a timed extension of the untimed unary temporal logic UTL studied by [9].
- Metric Interval Temporal Logic [5], denoted $MITL[\mathsf{U}_I, \mathsf{S}_I] = \mathbb{Z}IExtMTL[\mathsf{U}_I, \mathsf{S}_I]$. In this logic, the timing constraints in the formulas are restricted to non-punctual (non-singular) intervals. $MITL[\mathsf{F}_I, \mathsf{P}_I]$ is $MITL[\mathsf{U}_I, \mathsf{S}_I]$ confined to the unary modalities F_I and P_I.
- Bounded MTL [7], denoted $BoundedMTL[\mathsf{U}_I, \mathsf{S}_I] = Bd\mathbb{Z}IMTL[\mathsf{U}_I, \mathsf{S}_I]$. Other logics can be obtained as intersections of the above logics. Specifically, the logics $BoundedMTL[\mathsf{F}_I, \mathsf{P}_I]$, $BoundedMITL[\mathsf{U}_I, \mathsf{S}_I]$, and $BoundedMITL[\mathsf{F}_I, \mathsf{P}_I]$ are defined respectively as $Bd\mathbb{Z}IMTL[\mathsf{F}_I, \mathsf{P}_I]$, $Bd\mathbb{Z}IExtMTL[\mathsf{U}_I, \mathsf{S}_I]$, and $Bd\mathbb{Z}IExtMTL[\mathsf{F}_I, \mathsf{P}_I]$.
- Let $\mathbb{Z}I^k$ denote the set of all intervals of the form $\langle i, j\rangle$ or $\langle i, \infty)$, with $i, j \leq k$. Let $Bd\mathbb{Z}I^k$ denote the set of all bounded (i.e. non-infinite) $\mathbb{Z}I^k$ intervals. Then $MTL[\mathsf{U}_I, \mathsf{S}_I]^k$ and $BoundedMTL[\mathsf{U}_I, \mathsf{S}_I]^k$ are respectively the logic $\mathbb{Z}I^kMTL[\mathsf{U}_I, \mathsf{S}_I]$ and $Bd\mathbb{Z}I^kMTL[\mathsf{U}_I, \mathsf{S}_I]$. Also, given an $MTL[\mathsf{U}_I, \mathsf{S}_I]$ formula ϕ, let $MaxInt(\phi)$ denote the maximum integer constant (apart from ∞) appearing in its interval constraints.

2.3 Freeze Logics

These logics specify timing constraints by conditions on special variables, called freeze variables which memorize the time stamp at which a subformula is evaluated. Let $\mathcal{X}$ be a finite set of freeze variables. Let $x \in \mathcal{X}$ and let $\nu : \mathcal{X} \to \mathbb{R}_0$ be a valuation which assigns a non-negative real number to each freeze variable. Let ν_0 be the initial valuation such that $\forall x\ .\ \nu_0(x) = 0$ and let $\nu(x \leftarrow r)$ denote the valuation such that $\nu(x \leftarrow r)(x) = r$ and $\nu(x \leftarrow r)(y) = \nu(y)$ if $x \neq y$.

A *timing constraint* g in freeze logics has the form:

$g\ :=\ g_1 \wedge g_2 \ \mid\ x - T \approx c$ where $\approx \in \{<, \leq, >, \geq, =\}$ and $c \in \mathbb{Z}$.

Let $\nu, t \models g$ denote that the timing constraint g evaluates to *true* in valuation ν with $t \in \mathbb{R}_0$ assigned to the variable T.

TPTL$[U,S]$. given by [4,15], is an extension of LTL with freeze variables. Let g be a guard as defined above. The syntax of a *TPTL*$[\mathsf{U},\mathsf{S}]$ formula ϕ is as follows:

$$\phi := a \mid g \mid \phi \mathsf{U} \phi \mid \phi \mathsf{S} \phi \mid x.\phi \mid \phi \vee \phi \mid \neg\phi$$

The semantics of *TPTL*$[\mathsf{U},\mathsf{S}]$ formulas over a timed word ρ with $i \in dom(\rho)$ and valuation ν is as follows. The boolean connectives have their usual meaning.

$$\begin{aligned}
\rho,i,\nu \models a &\text{ iff } \sigma_i = a\\
\rho,i,\nu \models \phi_1 \mathsf{U} \phi_2 &\text{ iff } \exists j > i \,.\, \rho,j,\nu \models \phi_2 \text{ and } \forall i < k < j \,.\, \rho,k,\nu \models \phi_1\\
\rho,i,\nu \models \phi_1 \mathsf{S} \phi_2 &\text{ iff } \exists j < i \,.\, \rho,j,\nu \models \phi_2 \text{ and } \forall j < k < i \,.\, \rho,k,\nu \models \phi_1\\
\rho,i,\nu \models x.\phi &\text{ iff } \rho,i,\nu(x \rightarrow \tau_i) \models \phi\\
\rho,i,\nu \models g &\text{ iff } \nu,\tau_i \models g
\end{aligned}$$

The language defined by a *TPTL*$[\mathsf{U},\mathsf{S}]$ formula ϕ is given by $L(\phi) = \{\rho \mid \rho,1,\nu_0 \models \phi\}$. Also, *TPTL*$[\mathsf{F},\mathsf{P}]$ is the unary sub logic of *TPTL*$[\mathsf{U},\mathsf{S}]$.

Deterministic Freeze Logic. *TTL*$[X_\theta, Y_\theta]$ is a sub logic of *TPTL*$[\mathsf{U},\mathsf{S}]$. A *guarded event* over an alphabet Σ and a finite set of freeze variables $\mathcal{X}$ is a pair $\theta = (a,g)$ where $a \in \Sigma$ is an event and g is a timing constraint over $\mathcal{X}$ as defined before. Logic *TTL*$[X_\theta, Y_\theta]$ uses the deterministic modalities X_θ and Y_θ which access the position with the *next* and *previous* occurrence of a guarded event, respectively. This is the timed extension of logic $TL[X_a, Y_a]$ [8] using freeze quantification. The syntax of a *TTL*$[X_\theta, Y_\theta]$ formula ϕ is as follows:

$$\phi := \top \mid \theta \mid SP\phi \mid EP\phi \mid X_\theta\phi \mid Y_\theta\phi \mid x.\phi \mid \phi \vee \phi \mid \neg\phi$$

The semantics of *TTL*$[X_\theta, Y_\theta]$ formulas over timed words is as given below. $\top$ denotes the formula *true*. This and the boolean operators have their usual meaning.

$$\begin{aligned}
\rho,i,\nu \models \theta &\text{ iff } \sigma_i = a \text{ and } \nu,\tau_i \models g \text{ where } \theta = (a,g)\\
\rho,i,\nu \models SP\phi &\text{ iff } \rho,1,\nu \models \phi\\
\rho,i,\nu \models EP\phi &\text{ iff } \rho,\#\rho,\nu \models \phi\\
\rho,i,\nu \models X_\theta\phi &\text{ iff } \exists j > i \,.\, \rho,j,\nu \models \theta \text{ and } \forall i < k < j.\\
&\qquad \rho,k,\nu \not\models \theta \text{ and } \rho,j,\nu \models \phi\\
\rho,i,\nu \models Y_\theta\phi &\text{ iff } \exists j < i \,.\, \rho,j,\nu \models \theta \text{ and } \forall j < k < i.\\
&\qquad \rho,k,\nu \not\models \theta \text{ and } \rho,j,\nu \models \phi\\
\rho,i,\nu \models x.\phi &\text{ iff } \rho,i,\nu(x \leftarrow \tau_i) \models \phi
\end{aligned}$$

3 EF Games for *IvMTL*$[\mathsf{U}_I, \mathsf{S}_I]$

We extend the LTL EF games of [10] to timed logics, and use these to compare expressiveness of various instances of the generic logic *IvMTL*$[\mathsf{U}_I, \mathsf{S}_I]$. Let *Iv* be a given set of intervals. A k-round *IvMTL*$[\mathsf{U}_I, \mathsf{S}_I]$-EF game is played between two players, called *Spoiler* and *Duplicator*, on a pair of timed words ρ_0 and ρ_1. A configuration of the game (after any number of rounds) is a pair of positions (i_0, i_1) with $i_0 \in dom(\rho_0)$ and $i_1 \in dom(\rho_1)$. A configuration is called partially isomorphic, denoted $isop(i_0,i_1)$ iff $\sigma_{i_0} = \sigma_{i_1}$.

The game is defined inductively on k from a starting configuration (i_0, i_1) and results in either the *Spoiler* or *Duplicator* winning the game. The *Duplicator* wins the 0-round game iff $isop(i_0,i_1)$. The $k+1$ round game is played by first playing one round from the starting position. Either the spoiler wins in this round (and the game is terminated) or the game results into a new configuration (i'_0,i'_1). The game then proceeds inductively with k-round play from the configuration (i'_0,i'_1). The *Duplicator* wins the game only if it wins every round of the game. We now describe one round of play from a starting configuration (i_0,i_1).

- At the start of the round, if $\neg isop(i_0,i_1)$ then the *Spoiler* wins the game and the game is terminated. Otherwise,
- The *Spoiler* chooses one of the words by choosing $\delta \in \{0,1\}$. Then $\overline{\delta} = (1-\delta)$ gives the other word. The *Spoiler* also chooses either an U_I-move or a S_I move, including an interval $I \in Iv$. The remaining round is played in two parts.

U_I Move

- *Part I:* The *Spoiler* chooses a position i'_δ such that $i_\delta < i'_\delta \leq \#\rho_\delta$ and $(\tau_\delta[i'_\delta] - \tau_\delta[i_\delta]) \in I$.
- The *Duplicator* responds[1] by choosing a position $i'_{\overline{\delta}}$ in the other word s.t. $i_{\overline{\delta}} < i'_{\overline{\delta}} \leq \#\rho_{\overline{\delta}}$ and $(\tau_{\overline{\delta}}[i'_{\overline{\delta}}] - \tau_{\overline{\delta}}[i_{\overline{\delta}}]) \in I$. If the *Duplicator* cannot find such a position, the *Spoiler* wins the game. Otherwise the play continues to Part II.
- *Part II*: *Spoiler* chooses to play either F-part or U-part.
  - F-part: the round ends with configuration (i'_0,i'_1).
  - U-part: *Spoiler* verifies that $i'_\delta - i_\delta = 1$ iff $i'_{\overline{\delta}} - i_{\overline{\delta}} = 1$ and *Spoiler* wins the game if this does not hold. Otherwise *Spoiler* checks whether $i'_\delta - i_\delta = 1$. If *yes*, the round ends with configuration (i'_0,i'_1). If *no*, *Spoiler* chooses a position $i''_{\overline{\delta}}$ in the other word such that $i_{\overline{\delta}} < i''_{\overline{\delta}} < i'_{\overline{\delta}}$. The *Duplicator* responds by choosing i''_δ such that $i_\delta < i''_\delta < i'_\delta$. The round ends with the configuration (i''_0,i''_1).

S_I Move This move is symmetric to U_I where the *Spoiler* chooses positions i'_δ as well as $i''_{\overline{\delta}}$ in "past" and the *Duplicator* also responds accordingly. In Part II, the *Spoiler* will a have choice of P-part or S-part. We omit the details. This completes the description of the game.

Definition 1. *Given two timed words ρ_0,ρ_1 and $i_0 \in dom(\rho_0), i_1 \in dom(\rho_1)$, we define*

- $(\rho_0,i_0) \approx_k^{Iv} (\rho_1,i_1)$ *iff for every k-round IvMTL$[\mathsf{U}_I,\mathsf{S}_I]$ EF-game over the words ρ_0,ρ_1 and starting from the configuration (i_0,i_1), the Duplicator always has a winning strategy.*
- $(\rho_0,i_0) \equiv_k^{Iv} (\rho_1,i_1)$ *iff for every IvMTL$[\mathsf{U}_I,\mathsf{S}_I]$ formula ϕ of operator depth $\leq k$, $\rho_0,i_0 \models \phi \Leftrightarrow \rho_1,i_1 \models \phi$.* □

We shall now state the *IvMTL*$[\mathsf{U}_I,\mathsf{S}_I]$ EF theorem. Its proof is a straight-forward extension of the proof of LTL EF theorem of [10]. The only point of interest is that there is no a priori bound on the set of intervals that a modal depth n formula can use and hence the set of isomorphism types seems potentially infinite. However, given timed words ρ_0

[1] The *Duplicator* can make use of the knowledge of I to choose his move. This is needed as illustrated in the proof of Theorem 3.

and ρ_1, we can always restrict these intervals to not go beyond a constant k where k is the smallest integer larger than the biggest time stamps in ρ_0 and ρ_1. This restricts the isomorphism types to a finite cardinality. The complete proof can be found in the full version of this paper.

Theorem 1. $(\rho_0, i_0) \approx_k^{Iv} (\rho_1, i_1)$ *if and only if* $(\rho_0, i_0) \equiv_k^{Iv} (\rho_1, i_1)$ □

When clear from context, we shall abbreviate $\approx_k^{Iv}$ by $\approx_k$ and $\equiv_k^{Iv}$ by $\equiv_{Iv}$. As temporal logic formulas are anchored to initial position 1, define $\rho_0 \equiv_k \rho_1 \iff (\rho_0, 1) \equiv_k (\rho_1, 1)$ and $\rho_0 \approx_k \rho_1 \iff (\rho_0, 1) \approx_k (\rho_1, 1)$. It follows from the EF Theorem that $\rho_0 \equiv_k \rho_1$ if and only if $\rho_0 \approx_k \rho_1$.

We can modify the *IvMTL*$[\mathsf{U}_I, \mathsf{S}_I]$ EF game to match the sub logic *IvMTL*$[\mathsf{F}_I, \mathsf{P}_I]$. An *IvMTL*$[\mathsf{F}_I, \mathsf{P}_I]$ game is obtained by the restricting *IvMTL*$[\mathsf{U}_I, \mathsf{S}_I]$ game such that in PART II of any round, the *Spoiler* always chooses an F-part or a P-part. The corresponding *IvMTL*$[\mathsf{F}_I, \mathsf{P}_I]$ EF Theorem also holds.

4 Separating Sub Logics of *MTL*$[\mathsf{U}_I, \mathsf{S}_I]$

Each formula of a timed logic defines a timed language. Let $\mathcal{L}(\mathcal{G})$ denote the set of languages definable by the formulas of logic $\mathcal{G}$. A logic $\mathcal{G}_1$ is at least as expressive as (or contains) logic $\mathcal{G}_2$ if $\mathcal{L}(G_2) \subseteq \mathcal{L}(G_1)$. This is written as $\mathcal{G}_2 \subseteq \mathcal{G}_1$. Similarly, we can define $\mathcal{G}_2 \subsetneq \mathcal{G}_1$ (strictly contained within), $\mathcal{G}_2 \nsubseteq \mathcal{G}_1$ (not contained within), $\mathcal{G}_2 \# \mathcal{G}_1$ (incomparable), and $\mathcal{G}_2 \equiv \mathcal{G}_1$ (equally expressive).

We consider three sub logics of *MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ namely *MTL*$[\mathsf{F}_I, \mathsf{P}_I]$, *MITL*$[\mathsf{U}_I, \mathsf{S}_I]$ and *BoundedMTL*$[\mathsf{U}_I, \mathsf{S}_I]$.These have fundamentally different restrictions and using their corresponding EF-games, we show that they are all incomparable with each other.[2]

Theorem 2. *MITL*$[F_I, P_I] \nsubseteq$ *BoundedMTL*$[U_I, S_I]$

Proof. Consider the *MITL*$[\mathsf{F}_I, \mathsf{P}_I]$ formula $\phi := \mathsf{F}_{[0,\infty)}(a \wedge \mathsf{F}_{(1,2)} c)$. Consider a family of words $\mathcal{A}_n$ and $\mathcal{B}_n$. We have $untime(\mathcal{A}_n) = untime(\mathcal{B}_n) = a^{n+1}c$ with the a's occurring at integral time stamps $0, 1, \ldots, n$ in both words. In $\mathcal{A}_n$, the letter c occurs at time $n + 2.5$ and hence time distance between any a and c is more than 2. In $\mathcal{B}_n$, the c occurs at time $n + 1.5$ and the time distance between the c and the preceding a is in $(1, 2)$. Clearly, $\mathcal{A}_n \not\models \phi$ whereas $\mathcal{B}_n \models \phi$ for any $n > 0$.

We prove the theorem using an m-round *BdℤI*^{k}*MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ EF game on the words $\mathcal{A}_n$ and $\mathcal{B}_n$ where $n = mk$. We show that *Duplicator* has a winning strategy. Note that in such a game the *Spoiler* is allowed to choose intervals at every round with maximum upper bound of k and hence can shift the pebble at most k positions to the right. It is easy to see that the *Spoiler* is never able to place a pebble on the last c. Hence, the *Duplicator* has a winning strategy where she exactly copies the *Spoiler* moves. Using the EF theorem, we conclude that no modal depth n formula of logic *BdℤI*^{k}*MTL*$[\mathsf{U}_I, \mathsf{S}_I]$ can separate the words $\mathcal{A}_n$ and $\mathcal{B}_n$. Hence, there doesn't exist a *BoundedMTL*$[\mathsf{U}_I, \mathsf{S}_I]$ formula giving the language $L(\phi)$. □

[2] It was already observed by Bouyer *et al* [7] that *BoundedMTL*$[\mathsf{U}_I, \mathsf{S}_I]$ and *MITL*$[\mathsf{U}_I, \mathsf{S}_I]$ have separate expressiveness.

Theorem 3. *BoundedMTL*$[\mathsf{F}_I,\mathsf{P}_I] \not\subseteq$ *MITL*$[\mathsf{U}_I,\mathsf{S}_I]$

Proof. Consider the *BoundedMTL*$[\mathsf{F}_I,\mathsf{P}_I]$ formula $\phi := \mathsf{F}_{(0,1)}(a \wedge \mathsf{F}_{[3,3]}c)$. Consider a family of words A_n such that $untime(A_n) = a^{2n+1}c^{2n+1}$. Let $\delta = 1/(2n+2)^2$ and $\varepsilon = 1/(2n+2)^4$. All the a's are in the interval (0,1) at time stamps $i\delta$ and all the c's are in the interval $(3,4)$, at time stamps $3+i\delta+\varepsilon$ for $1 \le i \le 2n+1$. Every a has a paired c, which is at a distance $3+\varepsilon$ from it. Hence, $\forall n \,.\, A_n \not\models \phi$. Let B_n be a word identical to A_n but with the middle c shifted leftwards by ε, so that it is exactly at a distance of 3 t.u. (time units) from the middle a. Thus, $B_n \models \phi$.

We prove the theorem using the n-round $\mathbb{Z}IExtMTL[\mathsf{U}_I,\mathsf{S}_I]$EF game on the words A_{2n} and $\mathcal{B}_{2n}$ where we can show that *Duplicator* has a winning strategy. This proves that no modal depth n formula of logic *MITL*$[\mathsf{U}_I,\mathsf{S}_I]$ can separate A_{2n} and B_{2n}. Hence, there is no *MITL*$[\mathsf{U}_I,\mathsf{S}_I]$ formula giving $\mathcal{L}(\phi)$ The full description of the *Duplicator* strategy can be found in the full version of this paper. □

Theorem 4.
- *BoundedMTL*$[\mathsf{U}_I,\mathsf{S}_I] \not\subseteq$ *MTL*$[\mathsf{F}_I,\mathsf{P}_I]$ *over strict monotonic timed words (and hence also over weakly monotonic timed words).*
- *BoundedMTL*$[\mathsf{U}_I,\mathsf{S}_I] \not\subseteq$ *TPTL*$[\mathsf{F},\mathsf{P}]$ *over weakly monotonic timed words.* □

These results follow by embedding untimed LTL into logics MTL as well as TPTL. The proof can be found in the full version of this paper.

5 TPTL and MTL

Consider the *TPTL*$[\mathsf{F}]$ formula $\phi_1 \overset{\text{def}}{=} x.\mathsf{F}(b \wedge \mathsf{F}(c \wedge T - x \le 2))$. Bouyer *et al* [6] showed that this formula cannot be expressed in *MTL*$[\mathsf{U}_I]$ for pointwise models. They also gave an *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$ formula equivalent to it thereby showing that *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$ is strictly more expressive than *MTL*$[\mathsf{U}_I]$. Prior to this, Alur and Henzinger [3] considered the formula $\Box(a \Rightarrow \phi_1)$ and they conjectured that this cannot be expressed within *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$. Using a variant of this formula and the *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$ EF games, we now show that *TPTL*$[\mathsf{F}]$ is indeed expressively incomparable with *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$.

In Theorem 4 we showed that *BoundedMTL*$[\mathsf{U}_I,\mathsf{S}_I] \not\subseteq$ *TPTL*$[\mathsf{F},\mathsf{P}]$ over weakly monotonic timed words. We now consider the converse.

Theorem 5. *TPTL*$[F] \not\subseteq$ *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$ *over strictly monotonic timed words (and hence also for weakly monotonic timed words).*

Proof. Let the *TPTL*$[F]$ formula $\phi := \mathsf{F}p.[a \wedge \{\mathsf{F}(b \wedge (T-p \in (1,2)) \wedge \mathsf{F}(c \wedge (T-p \in (1,2))))\}]$. This formula characterizes the set of timed words which have an a followed by a b and then a c such that the time lag between the a and b is in the interval $(1,2)$ and the time lag between the a and c is also in $(1,2)$. We show that there is no *MTL*$[\mathsf{U}_I,\mathsf{S}_I]$ formula that expresses the language defined by ϕ.

The idea behind the proof is the following. We will design two families of strictly monotonic timed words $\mathcal{A}_{n,k}$ and $\mathcal{B}_{n,k}$ $(n > 0)$, such that $\mathcal{A}_{n,k} \models \phi$ and $\mathcal{B}_{n,k} \not\models \phi$. We will then show that for n round $\mathbb{Z}I^kMTL[\mathsf{U}_I,\mathsf{S}_I]$ EF games over $\mathcal{A}_{n,k}$ and $\mathcal{B}_{n,k}$ the duplicator has a winning strategy. Hence, no n modal depth $\mathbb{Z}I^kMTL[\mathsf{U}_I,\mathsf{S}_I]$ formula can distinguish words $\mathcal{A}_{n,k}$ and $\mathcal{B}_{n,k}$. Thus, there is no formula in $\mathbb{Z}IMTL[\mathsf{U}_I,\mathsf{S}_I]$ giving $L(\phi)$.

Designing the Words. Fix some n,k. Let $m = 2n(k+1)+1$, $\delta = 1/2m$ and $\varepsilon << \delta$. First, we shall describe $\mathcal{B}_{n,k}$. The first event is an a at time stamp 0. (This event is included since all words must begin with time stamp 0.) Following this, there are no events in the interval $(0,k]$. From $k+1$ onwards, it has m copies of identical and overlapping segments of length $2+\varepsilon$ time units each. If the i^{th} segment seg_i begins at some time stamp (say t) then seg_{i+1} begins at $(t+1-\delta)$. The beginning of each segment is marked by an a at t, followed by a b in the interval $(t+2-2\delta+2\varepsilon, t+2-\delta-2\varepsilon)$, and a c in the interval $(t+2, t+2+\varepsilon)$, as shown in figure 2. Note that all the events must be placed such that no two events are exactly at an integral distance from each other (this is possible, since n and k are finite and time is dense). Let $X = n(k+1)+1$. The X^{th} segment is the middle segment, which is padded by $n(k+1)$ segments on either side. Let seg_X begin at time stamp x and the following segments begin at y and z respectively, as shown in the figure 3. Let p_x denote the position corresponding to the time stamp x in both words.

$\mathcal{A}_{n,k}$ is identical to $\mathcal{B}_{n,k}$ except for the X^{th} segment where the corresponding c is shifted leftwards to be in the interval $(x+2-\varepsilon, x+2)$. Let p^A and $p^{A'}$ denote the positions of c corresponding to seg_X and seg_{X+1} in $\mathcal{A}_{n,k}$ respectively. Similarly, let p^B and $p^{B'}$ denote the positions of c corresponding to seg_X and seg_{X-1} in $\mathcal{B}_{n,k}$ respectively.

Note that $\mathcal{B}_{n,k}$ is such that for every a, there exists a c at a distance $(1,2)$ from it, but the b between them is at a distance < 1 t.u. from the a. In addition, every a has a b at a distance $(1,2)$ from it, but the subsequent c is at a distance > 2 t.u. from the a. See Figure 3. Hence, $\forall n,k > 0$, $\mathcal{B}_{n,k} \not\models \phi$. On the other hand, $\mathcal{A}_{n,k}$ is identical to $\mathcal{B}_{n,k}$ except for the $(n(k+1)+1)^{st}$ segment for which the c is shifted left so that a has a b followed by c, both of which are within time distance $(1,2)$ from the a. Hence, $\forall n,k > 0$, $\mathcal{A}_{n,k} \models \phi$. Since all the events occur at time stamps $> k$, the *Spoiler* cannot differentiate between integer boundaries. This enables us to disregard the integer boundaries between the events through the play of the game. Moreover, since the words are such that no two events are exactly integral distance apart from each other, the *Spoiler* is forced to choose a non-singular interval in every round.

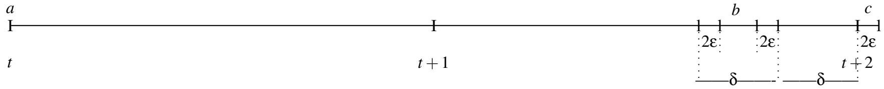


Fig. 2. $MTL[\mathsf{U}_I, \mathsf{S}_I]$ EF game : A single segment in $\mathcal{B}_{n,k}$

Key moves of Duplicator. As the two words are identical except for the time stamp of the middle c, the strategy of *Duplicator* is to play a configuration of the form (i,i) whenever possible. Such a configuration (i,i) is called an identical configuration. The optimal strategy of *Spoiler* is to get out of identical configurations as quickly as possible. We give two example plays, where the *Spoiler* can force non-identical configuration (depicted by dotted arrows in figure 3). In first move, the *Spoiler* plays position p_x which *Duplicator* duplicates giving the initial configuration of (p_x, p_x).

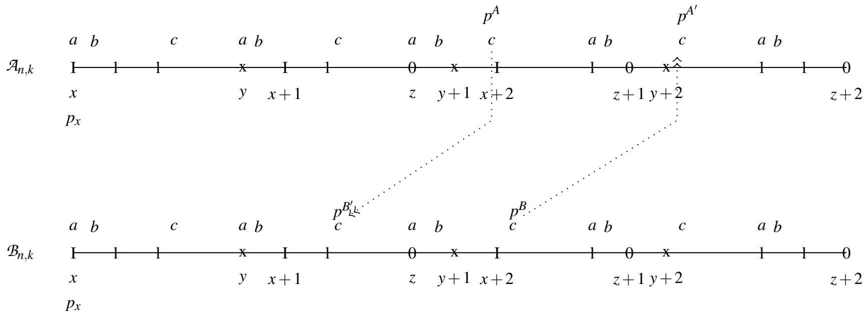

p^A
$p^{A'}$
$\mathcal{A}_{n,k}$
a b c a b c a b c a b c a b
x y $x+1$ z $y+1$ $x+2$ $z+1$ $y+2$ $z+2$
p_x
$p^{B'}$
p^B
$\mathcal{B}_{n,k}$
a b c a b c a b c a b c a b
x y $x+1$ z $y+1$ $x+2$ $z+1$ $y+2$ $z+2$
p_x

Fig. 3. $MTL[\mathsf{U}_I, \mathsf{S}_I]$ EF game : Duplicator's Strategy

1. If the *Spoiler* chooses the interval $(1,2)$ and places its pebble at p^A, then the *Duplicator* will be forced to place its pebble at $p^{B'}$, which also occurs in the interval $x+(1,2)$. This is shown by downward dotted arrow in the figure.
2. Alternatively, if the *Spoiler* chooses the interval $(2,3)$ and places a pebble at p^B in $\mathcal{B}_{n,k}$, then the *Duplicator* is forced to place its pebble on $p^{A'}$, which is also in the interval $x+(2,3)$.

In both cases, if (i,j) is the resulting configuration, then $seg(i)-seg(j)=1$.

Duplicator's copy-cat Strategy. Consider the p^{th} round of the game, with an initial configuration (i_p, j_p). If the *Duplicator* plays in a manner such that the configuration for the next round is (i_{p+1}, j_{p+1}) with $seg(i_p)-seg(i_{p+1}) = seg(j_p)-seg(j_{p+1})$, then it is said to have followed the *copy-cat* strategy for the p^{th} round.

Proposition 1. *The only case when the Duplicator can not follow the copy-cat strategy in a round with initial configuration (i,j), is when $i=j=p_x$ and the Spoiler chooses to first place its pebble on either p^A or p^B or when $i=p^A$ and $j=p^B$ and Spoiler chooses to place a pebble at p_x in either word.*

Proof. Firstly, note that $untime(\mathcal{A}_{n,k}) = untime(\mathcal{B}_{n,k})$ and the only position at which the two words differ is at p^A (and correspondingly p^B), where $\tau_{p^B}-\tau_{p^A} < 2\varepsilon$. By observing the construction of the words, we can infer that $\forall p \in dom(\mathcal{A}_{n,k})$, if $p \neq p_x$ then $\forall i \in \mathbb{Z}$ we have $\tau_p - \tau_{p^A} \in (i, i+1)$ iff $\tau_p - \tau_{p^B} \in (i, i+1)$. However, if the initial configuration is (p_x, p_x) or (p^A, p^B), then *Duplicator* may not be able to follow the *copy-cat* strategy, since p_A and p_B lie on either side of $x+2$. □

The lemma below shows that in an n round game, for each round, the *Duplicator* can either achieve an identical configuration, or restrict the segment difference between words to a maximum of 1 in which case there are sufficient number of segments on either side for the *Duplicator* to be able to duplicate the *Spoiler*'s moves for the remaining rounds.

Lemma 1. *For an n round $\mathbb{Z}I^k$ $MTL[U_I, S_I]$ EF game over the words $\mathcal{A}_{n,k}, \mathcal{B}_{n,k}$ the Duplicator always has a winning strategy such that for any $1 \leq p \leq n$, if (i_p, j_p) is the initial configuration of the p^{th} round then*

- $seg(i_p) - seg(j_p) \leq 1$ *AND*
- *If* $seg(i_p) \neq seg(j_p)$ *then*
  $Min\{seg(i_p), seg(j_p)\} > (n-p+1)(k+1)$
  $Max\{seg(i_p), seg(j_p)\} < m-(n-p+1)(k+1)$

Proof. The duplicator always follows *copy-cat* strategy in any configuration whenever possible. We can prove the lemma by induction on p.
Base step: The lemma holds trivially for $p = 1$, as starting configuration $(i_1, j_1) = (1, 1)$.
Induction Step: Assume that the lemma is true for some $p < n$. We shall prove that the lemma holds for $p+1$. Consider the p^{th} round, with initial configuration (i_p, j_p).
Case 1: The *Duplicator* can follow *copy-cat* strategy :
Then, $seg(i_{p+1}) - seg(j_{p+1}) = seg(i_p) - seg(j_p)$. By induction hypothesis, $seg(i_p) - seg(j_p) \leq 1$ giving $seg(i_{p+1}) - seg(j_{p+1}) \leq 1$. Also, since exactly k number of segments begin within a time span of k time units, if the *Spoiler* chooses an interval of the form (h, l), with $l \leq k$, then we know that $seg(i_{p+1}) - seg(i_p) \leq k$ and the lemma will hold for $p+1$. If the *Spoiler* chooses an interval (k, ∞) and places a pebble $k+1$ segments away in $\mathcal{A}_{n,k}$, the *Duplicator* also has to place its pebble at least $k+1$ segments away, thereby, either making $seg(i_{p+1}) = seg(j_{p+1})$ or making i_{p+1} and j_{p+1} come closer to either end by at most $k+1$ segments.
Case 2: The *Duplicator* can not follow *copy-cat* strategy:
From proposition 1, this can happen only if $seg(i_p) = seg(j_p) = X$, the middle segment. In this case, we know that $seg(i_{p+1}) - seg(j_{p+1}) = 1$, $X-2 \leq seg(i_{p+1}) \leq X+2$ and $X-2 \leq seg(j_{p+1}) \leq X+2$. Hence the lemma holds in this case too. □

6 Comparing $TTL[X_\theta, Y_\theta]$ with $MTL[\mathsf{U}_I, \mathsf{S}_I]$ Fragments

6.1 Embedding $TTL[X_\theta, Y_\theta]$ into $MITL[\mathsf{F}_I, \mathsf{P}_I]$:

Fix a formula $\phi \in TTL[X_\theta, Y_\theta]$. The formula ϕ may be represented by its parse tree T_ϕ, such that the subformulas of ϕ form the subtrees of T_ϕ. Let $Subf(n)$ denote the subformula corresponding to the subtree rooted at node n, and let n be labelled by $Opr(n)$ which is the outermost operator (such as $X_\theta, \vee, \neg, x.$ etc.) if n is an interior node, and by the corresponding atomic proposition, if it is a leaf node. We will use the notion of subformulas and nodes interchangeably. The *ancestry* of a subformula n is the set of nodes in the path from the root up to (and including) n.
Let η to range over subformulas of ϕ with η_{root} denoting ϕ. Logic $TTL[X_\theta, Y_\theta]$ is a deterministic freeze logic. Hence, given a timed word ρ, in evaluating $\rho, 1, \nu_0 \models \phi$, any subformula η of ϕ needs to be evaluated only at a uniquely determined position in $dom(\rho) \cup \{\perp\}$ called $pos_\rho(\eta)$. We call this the *Unique Parsability* property of $TTL[X_\theta, Y_\theta]$ formulas. Here, notation $pos_\rho(\eta) = \perp$ indicates that such a position does not exist in ρ and that the subformula η plays no role in evaluating $\rho, 1, \nu_0 \models \phi$. Also, $val_\rho(\eta)$ is the unique valuation function of freeze variables under which η is evaluated. Note that pos is strict w.r.t. $\perp$, i.e. if $\eta = OP(\ldots, \eta_1, \ldots)$ and $pos_\rho(\eta) = \perp$ then $pos_\rho(\eta_1) = \perp$. Also, val is a partial function where $val_\rho(\eta)$ is defined only when $pos_\rho(\eta) \neq \perp$. We define $pos_\rho(\eta)$ together with $val_\rho(\eta)$ which are both simultaneously defined by induction on the depth of η. Firstly, define $pos_\rho(\eta_{root}) = 1$ and $val_\rho(\eta_{root}) = \nu_0$. Now consider cases where $pos_\rho(\eta) = i\ (\neq \perp)$ and $val_\rho(\eta) = \nu$.

- If $\eta = SP\eta_1$ then $pos_\rho(\eta_1) = 1$ and $val_\rho(\eta_1) = \nu$.
- If $\eta = EP\eta_1$ then $pos_\rho(\eta_1) = \#\rho$ and $val_\rho(\eta_1) = \nu$.
- If $\eta = \eta_1 \vee \eta_2$ or $\eta = \neg\eta_1$ then $pos_\rho(\eta_1) = pos_\rho(\eta_2) = i$ and $val_\rho(\eta_1) = val_\rho(\eta_2) = \nu$.
- If $\eta = x.\eta_1$ then $pos_\rho(\eta_1) = i$ and $val_\rho(\eta_1) = \nu(x \leftarrow \tau_i)$.
- Let $\eta = X_\theta\eta_1$. Then, $pos_\rho(\eta_1) = \bot$ if $\forall k > i,\ \rho,k,\nu \not\models \theta$. Otherwise, $pos_\rho(\eta_1) = j$ s.t. $j > i$ and $\rho, j, \nu \models \theta$ and $\forall i < k < j,\ \rho,k,\nu \not\models \theta$. Moreover, $val_\rho(\eta_1) = \nu$.
- The case of $\eta = Y_\theta\eta_1$ is symmetric to that of $\eta = X_\theta\eta_1$.

Given a freeze variable x, let $anc_x(\eta)$ be the node in the ancestry of η and nearest to it, at which x is frozen. Hence, $anc_x(\eta)$ is the smallest ancestor η' of η, which is of the form $x.\eta'$. If there is no such ancestor, then let $anc_x(\eta) = \eta_{root}$. The following proposition follows from this definition.

Proposition 2. $val_\rho(\eta)(x) = \tau_{pos_\rho(anc_x(\eta))}$

Lemma 2. *For any subformula η of a $TTL[X_\theta, Y_\theta]$ formula ϕ, we can effectively construct an $MITL[F_I, P_I]$ formula $\alpha(\eta)$ such that $\forall\rho \in T\Sigma^*$ we have $pos_\rho(\eta) = j$ iff $\rho, j \models \alpha(\eta)$.*

Proof. The construction of $\alpha(\eta)$ follows the inductive definition of $pos_\rho(\eta)$, and the lemma may be proved by induction on the depth of η. Consider any timed word ρ.

- Firstly, $\alpha(\eta_{root}) = \neg P_{(0,\infty)}\top$. Therefore, $\rho, i \models \alpha(\eta_{root})$ iff $i = 1$.
- Similarly, if $\eta = SP\eta_1$ then $\alpha(\eta_1) = \neg P_{(0,\infty)}\top$. Hence, $\rho, i \models \alpha(\eta_{root})$ iff $i = 1 = pos_\rho(\eta_1)$.
- If $\eta = EP\eta_1$ then $\alpha(\eta_1) = \neg F_{(0,\infty)}\top$. Hence, $\rho, i \models \alpha(\eta_{root})$ iff $i = \#\rho = pos_\rho(\eta_1)$.
- If η is of the form $\eta_1 \vee \eta_2$ or $\neg\eta$ or $x.\eta_1$ then $\alpha(\eta_1) = \alpha(\eta)$. This follows from the fact that $pos_\rho(\eta) = pos_\rho(\eta_1) = pos_\rho(\eta_2)$.
- Now consider the main case of $\eta = X_\theta\eta_1$ with $\theta = (a, g)$. For given θ, we define a corresponding $MITL[\mathsf{F}_I, \mathsf{P}_I]$ formula $\mathcal{CF}(\theta, \eta)$ such that the following proposition holds:

  Proposition 3. $\rho, i, val_\rho(\eta) \models \theta$ *iff* $\rho, i \models \mathcal{CF}(\theta, \eta)$.

  Using this we define $\alpha(\eta_1)$ and show that $\rho, i \models \alpha(\eta_1)$ iff $i = pos_\rho(\eta_1)$.
- The case of $\eta = Y_\theta\eta_1$ is symmetric to the above case.

Given $\theta = (a, g)$, define $\mathcal{CF}(\theta, \eta) = a \wedge \mathcal{C}(g, \eta)$ where the construction of $\mathcal{C}(g, \eta)$ is given in Table 1. Note that any constraint of the form $x - T = c$ can be replaced by equivalent constraint $(x - T \leq c) \wedge (x - T \geq c)$. Similarly, for $T - x = c$ too. We omit from Table 1, the remaining cases of $T - x \approx c$ which are similar. To sketch the proof of proposition 3, we first show that $\rho, i \models \mathcal{C}(g, \eta)$ iff $val_\rho(\eta), \tau_i \models g$. From this, it follows that $\rho, i, val_\rho(\eta) \models \theta$ iff $\rho, i \models \mathcal{CF}(\theta, \eta)$. Consider the case where $g = x - T < c$. Then, $\mathcal{C}(g, \eta) = F_{[0,c)}\alpha(anc_x(\eta))$. By semantics of $MITL[\mathsf{F}_I, \mathsf{P}_I]$, we know that $\rho, i \models \mathcal{C}(g, \eta)$ iff $\exists j > i$ such that (i) $j = pos_\rho(anc_x(\eta))$ (using the inductive hypothesis) and (ii) $\tau_j - \tau_i \in [0, c)$. However, from proposition 2, we know that $val_\rho(\eta)(x) = \tau_j$. Hence, (i) and (ii) hold iff $val_\rho(\eta), \tau_i \models g$. The other cases may be proved similarly.

Table 1.

g	$\mathcal{C}(g,\eta)$
$x-T<c$	$F_{[0,c)}\alpha(anc_x(\eta))$
$x-T\leq c$	$F_{[0,c]}\alpha(anc_x(\eta))$
$x-T>c$	$F_{(c,\infty)}\alpha(anc_x(\eta))$
$x-T\geq c$	$F_{[c,\infty)}\alpha(anc_x(\eta))$
$T-x<c$	$P_{[0,c)}\alpha(anc_x(\eta))$
$g_1\wedge g_2$	$\mathcal{C}(g_1,\eta)\wedge\mathcal{C}(g_2,\eta)$

Now, define $\alpha(\eta_1)=\mathcal{CF}(\theta,\eta)\ \wedge\ (P_{(0,\infty)}\alpha(\eta))\ \wedge\ (\neg P_{(0,\infty)}(\mathcal{CF}(\theta,\eta)\wedge P_{(0,\infty)}\alpha(\eta)))$. The three conjuncts of the above formula respectively give the following observations. $\rho,i\models\alpha(\eta_1)$ iff (i) $\rho,i,val_\rho(\eta_1)\models\theta$ (from proposition 3), (ii) $\exists j<i\ .\ j=pos_\rho(\eta)$ (from induction hypothesis), and (iii) $\forall k\ .\ pos_\rho(\eta)<k<i\ .\ \rho,k,val_\rho(\eta_1)\not\models\theta$ □

Now define the evaluation $eval_\rho(\eta)$ of a subformula as its truth value at its deterministic position $pos_\rho(\eta)$. This can be defined as follows: If $pos_\rho(\eta)\neq\bot$ then $eval_\rho(\eta)=(\rho,val_\rho(\eta),pos_\rho(\eta)\models\eta)$ and *false* otherwise. Clearly, since $pos_\rho(\eta_{root})=1$ and $val_\rho(\eta_{root})=\nu_0$, it follows that $eval_\rho(\eta_{root})=((\rho,1,\nu_0)\models\eta_{root})$.

Theorem 6. *For every subformula* η*, we construct an* $MITL[\mathsf{F}_I,\mathsf{P}_I]$ *formula* $\beta(\eta)$ *such that* $eval_\rho(\eta)$ *iff* $pos_\rho(\eta)\neq\bot$ *and* $\rho,pos_\rho(\eta)\models\beta(\eta)$.

The construction of $\beta(\eta)$ is by induction on the structure of η. In its construction, we use the formula $\alpha(\eta)$ given earlier. If $\eta=\top$ then $\beta(\eta)=\alpha(\eta)$. If $\eta=\theta$ then $\beta(\eta)=\alpha(\eta)\wedge\mathcal{C}F(\theta,\eta)$. If $\eta=\eta_1\vee\eta_2$ then $\beta(\eta)=\alpha(\eta)\wedge(\beta(\eta_1)\vee\beta(\eta_2))$. If $\eta=x.\eta_1$ then $\beta(\eta)=\beta(\eta_1)$. If $\eta=\neg\eta_1$ then $\beta(\eta)=\alpha(\eta)\wedge\neg\beta(\eta_1)$. Now, we consider the main case. Let $\eta=X_\theta\eta_1$. Then, $\beta(\eta)=\alpha(\eta)\wedge\mathsf{F}(\alpha(\eta_1)\wedge\beta(\eta_1))$. It is easy to prove by induction on the height of η that Theorem 6 holds.

6.2 On Limited Expressive Power of $TTL[X_\theta,Y_\theta]$

Given any $TTL[X_\theta,Y_\theta]$ formula, its *modal depth* corresponds to the maximum number of modal operators in any path of its parse tree and its *modal count* corresponds to the total number of modal operators in the the formula.
A $TTL[X_\theta,Y_\theta]$ formula ϕ is said to *reach* a position i in a word w, if there exists a subformula η of ψ such that $Pos_w(\eta)=i$.

Theorem 7. *1.* $BoundedMTL[\mathsf{F}_I,\mathsf{P}_I]\not\subseteq TTL[X_\theta,Y_\theta]$
2. $MITL[\mathsf{F}_I,\mathsf{P}_I]\not\subseteq TTL[X_\theta,Y_\theta]$

Proof. (i) Consider the $BoundedMTL[\mathsf{F}_I,\mathsf{P}_I]$ formula $\phi:=\mathsf{F}_{(0,1)}(a\wedge\mathsf{F}_{[3,3]}c)$ given in the proof of Theorem 3 and A_n and B_n be as defined in that proof. Let $w_n=A_{n+1}$ and $v_n=B_{n+1}$. Thus, both w_n and v_n consist of events $a^{2n+3}c^{2n+3}$. Then, $\forall n\ .\ w_n\notin\mathcal{L}(\phi)$ and $v_n\in\mathcal{L}(\phi)$.

Proposition 4. *For* $n>1$*, no* $TTL[X_\theta,Y_\theta]$ *formula of modal depth* $1\leq m\leq n$ *can reach the middle* $2n-2m+3$ *a's or the middle* $2n-2m+3$ *c's in* w_n.

Proof. Firstly, note that if no $TTL[X_\theta, Y_\theta]$ formula of depth m can reach a position i in a word, then its boolean combinations also cannot reach i in the word. We now prove the claim by induction on m, for some fixed n. *Base step:* $m = 1$: Since all a satisfy the same integral guards and all c also satisfy the same set of intergral guards, the topmost modality may match either the first or last a or the first or last c in w_n (irrespective of the guard that is chosen). Hence, the middle $2n-2+3$ a's and c's cannot be reached. *Induction Step:* Let the proposition be true for some $1 \leq m < n$. Hence for every ψ of modal depth m, ψ cannot reach the middle $(2n-2m+3)$ a's and c's in w_n. Every $TTL[X_\theta, Y_\theta]$ formula ψ' of modal depth $m+1$ may be obtained from some $TTL[X_\theta, Y_\theta]$ formula ψ of modal depth m, by extending every path in parse tree of ψ by at most one modality in the end. However, since all the middle $2n-2m+3$ a's and c's satisfy the same integral guards with respect to the time stamps of the peripheral a's and c's, adding another modality to ψ can make ψ' reach at most the $(m+1)^{th}$ or $n-(m+1)^{th}$ a or c. This leaves us with $2n-2m+3-2 \ = \ 2n-2(m+1)+3$ middle a's and c's which remain unreachable. □

Consider a $TTL[X_\theta, Y_\theta]$formula of modal depth $\leq n$. From proposition 4, the middle 3 a's and c's are unreachable. Moreover, they satisfy the same set of time constraints with respect to the reachable events. Hence, perturbing the middle c alone will not change the truth of the formula as c it will continue to satisfy the same set of timing constraints w.r.t. the reachable events. Hence $w_n \models \psi$ iff $v_n \models \psi$. Since $w_n \not\models \phi$ and $v_n \models \phi$, no ψ of modal depth $\leq n$ can distinguish between w_n and v_n. Hence, we can conclude that there is no $TTL[X_\theta, Y_\theta]$ formula equivalent to ϕ.

(ii) Consider the $MITL[\mathsf{F}_I, \mathsf{P}_I]$ formula $\phi := \mathsf{F}_{[0,\infty)}(a \wedge \mathsf{F}_{(1,2)}c)$ and (assuming to contrary) let ψ be a $TTL[X_\theta, Y_\theta]$ formula of modal count m such that $\mathcal{L}(\phi) = \mathcal{L}(\psi)$. Assuming that freeze variables in ψ are not reused, there are a maximum number of m freeze variables in ψ. Now consider the word w consisting of event sequence $(ac)^{4m+1}$ where the x'th ac pair gives the timed subword $(a, 2x)(c, 2x+0.5)$. Thus, each c is 0.5 t.u. away from its paired a, and 2.5 units away from the a of the previous pair. Hence, $w \notin \mathcal{L}(\phi)$.

Consider the evaluation of ψ over w. Each of the m freeze variables is frozen at most once, in the evaluation of ψ. By a counting argument, there are at least $m+1$ (possibly overlapping but distinct) subwords of the form $acacac$, none of whose elements are frozen. Call each such subword a group. Enumerate the groups sequentially. Let v_j be a word identical to w except that the j^{th} group is altered, such that its middle c is shifted by 0.7 t.u. to the right, so that v_j satisfies the property ϕ. Note that there are at least $m+1$ such distinct v_j's and for all j, $v_j \in \mathcal{L}(\phi)$.

Claim: Given a v_j, if there exists a subformula η of ψ such that $Pos_{v_j}(\eta)$ matches the altered c, then for all $k \neq j$, $Pos_{v_k}(\eta)$ does not match its altered c. (This is because, the altered c in v_j must satisfy a guard which none of its two surrounding c's in the group can satisfy).

From the above claim, we know that the m modalities in ψ, may match its position in at most m of the altered words v_j. However, the family $\{v_j\}$ has at least $m+1$ members.

Hence, there exists a k such that the altered c of v_k, (and the k^{th} group) is not reachable by ψ in w or any of the $\{v_j\}$. Hence $w \models \psi$ iff $v_k \models \psi$. But this is a contradiction as $w \notin L(\phi)$ and $v_k \in L(\phi)$ with $L(\phi) = L(\psi)$.

Therefore, there is no $TTL[X_\theta, Y_\theta]$ formula which can express the language $\mathcal{L}(\phi)$. □

References

1. Alur, R., Henzinger, T.: Real Time Logics: Complexity and Expressiveness. In: Proceedings of LICS, pp. 390–401 (1990)
2. Alur, R., Henzinger, T.: Back to the Future: Towards a Theory of Timed Regular Languages. In: Proceedings of FOCS, pp. 177–186 (1992)
3. Alur, R., Henzinger, T.: Real Time Logics: Complexity and Expressiveness. Inf. and Comput. 104, 35–77 (1993)
4. Alur, R., Henzinger, T.: A Really Temporal Logic. Jour. of the ACM 41(1), 181–204 (1994)
5. Alur, R., Feder, Henzinger, T.: The Benefits of Relaxing Punctuality. Journal of the ACM 43, 116–146 (1996)
6. Bouyer, P., Chevalier, F., Markey, N.: On the expressiveness of TPTL and MTL. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 432–443. Springer, Heidelberg (2005)
7. Bouyer, P., Markey, N., Ouaknine, J., Worrell, J.: On expressiveness and complexity in real-time model checking. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 124–135. Springer, Heidelberg (2008)
8. Dartois, L., Kufleitner, M., Lauser, A.: Rankers over infinite words. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 148–159. Springer, Heidelberg (2010)
9. Etessami, K., Vardi, M.Y., Wilke, T.: First-order logic with two variables and unary temporal logic. Inf. and Comput. 179, 279–295 (2002)
10. Etessami, K., Wilke, T.: An Until Hierarchy for Temporal Logic. In: Proceedings of LICS, pp. 108–117 (1996)
11. Immermann, N., Kozen, D.: Definability with Bounded Number of Bound Variables. Inf. and Comput., 121–139 (1989)
12. Ouaknine, J., Worrell, J.: Some recent results in metric temporal logic. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 1–13. Springer, Heidelberg (2008)
13. Pandya, P.K., Shah, S.S.: Unambiguity in timed regular languages: Automata and logics. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 168–182. Springer, Heidelberg (2010)
14. Pandya, P.K., Shah, S.S.: On Expressive Powers of Timed Logics: Comparing Boundedness, Non-punctuality, and Deterministic Freezing. CoRR, abs/1102.5638 (2011)
15. Raskin, J.-F.: Logics, Automata and Classical Theories for Deciding Real Time. Ph.D. Thesis, Univ. of Namur, Belgium (1999)
16. Thomas, W.: On The Ehrenfeucht-Fraïssé Game in Theoretical Computer Science. In: Gaudel, M.-C., Jouannaud, J.-P. (eds.) CAAP 1993, FASE 1993, and TAPSOFT 1993. LNCS, vol. 668, pp. 559–568. Springer, Heidelberg (1993)

Timed Automata Can Always Be Made Implementable*

Patricia Bouyer[1], Kim G. Larsen[2], Nicolas Markey[1],
Ocan Sankur[1], and Claus Thrane[2]

[1] LSV, CNRS & ENS Cachan, France
{bouyer,markey,sankur}@lsv.ens-cachan.fr
[2] Dept. Computer Science, Aalborg University, Denmark
{kgl,crt}@cs.aau.dk

Abstract. Timed automata follow a mathematical semantics, which assumes perfect precision and synchrony of clocks. Since this hypothesis does not hold in digital systems, properties proven formally on a timed automaton may be lost at implementation. In order to ensure implementability, several approaches have been considered, corresponding to different hypotheses on the implementation platform. We address two of these: A timed automaton is samplable if its semantics is preserved under a discretization of time; it is robust if its semantics is preserved when all timing constraints are relaxed by some small positive parameter.

We propose a construction which makes timed automata implementable in the above sense: From any timed automaton $\mathcal{A}$, we build a timed automaton $\mathcal{A}'$ that exhibits the same behaviour as $\mathcal{A}$, and moreover $\mathcal{A}'$ is both robust and samplable by construction.

1 Introduction

Timed automata [3] extend finite-state automata with real-valued variables which measure delays between actions. They provide a powerful yet natural way of modelling real-time systems. They also enjoy decidability of several important problems, which makes them a model of choice for the verification of real-time systems. This has been witnessed over the last twenty years by substantial effort from the verification community to equip timed automata with efficient tool support, which was accompanied by successful applications.

However, timed automata are governed by a mathematical semantics, which assumes continuous and infinitely precise measurement of time, while hardware is digital and imprecise. Hence properties proven at the formal level might be lost when implementing the abstract model of the automaton as a digital circuit or as a program on a physical CPU. Several approaches have been proposed to overcome this discrepancy, with different hypotheses on the implementation

* This work has been partly supported by EU FP7 project Quasimodo (ICT-214755), and by French ANR projects DOTS (ANR-06-SETI-003) and ImpRo (ANR-10-BLAN-0317).

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 76–91, 2011.

platform (*e.g.* [4,15,20,12,5,21]). In this work, we address two such approaches, namely, the sampled semantics and the robustness, which we now detail.

Sampled semantics for timed automata, where all time delays are integer multiples of a rational sampling rate, have been studied in order to capture, for example the behaviour of digital circuits (*e.g.* [4,8]). In fact, only such instants are observable in a digital circuit, under the timing of a quartz clock. However, for some timed automata, any sampling rate may disable some (possibly required) behaviour [9]. Consequently, a natural problem which has been studied is that of choosing a sampling rate under which a property is satisfied. For safety properties, this problem is undecidable for timed automata [9]; but it becomes decidable for reachability under a slightly different setting [17]. Recently, [1] showed the decidability of the existence of a sampling rate under which the continuous and the sampled semantics recognize the same untimed language.

A prominent approach, originating from [20,12], for verifying the behavior of real-time programs executed on CPUs, is robust model-checking. It consists in studying the *enlarged semantics* of the timed automaton, where all the constraints are enlarged by a small (positive) perturbation Δ, in order to model the imprecisions of the clock. In some cases [11], this may allow new behaviours in the system, regardless of Δ (See Fig. 2 on page 83). Such automata are said to be not *robust* to small perturbations. On the other hand, if no new behaviour is added to the system, that is, if the system is robust, then *implementability* on a fast-enough CPU will be ensured [12]. Since its introduction, robust model-checking has been solved for safety properties [20,11], and for richer linear-time properties [6,7]. See also [21] for a variant of the implementation model of [12] and a new approach to obtain implementations.

In this paper, we show that timed automata can always be made implementable in both senses. More precisely, given a timed automaton $\mathcal{A}$, we build another timed automaton $\mathcal{B}$ whose semantics under enlargement and under sampling is bisimilar to $\mathcal{A}$. We use a quantitative variant of bisimulation from [14] where the differences between the timings in two systems are bounded above by a parameter ε (see also [16] for a similar quantitative notion of bisimulation). Our construction is parameterized and provides a bisimilar implementation for any desired precision $\varepsilon > 0$. Moreover, we prove that in timed automata, this notion of bisimulation preserves, up to an error of ε, all properties expressed in a quantitative extension of CTL, also studied in [14].

2 Timed Models and Specifications

2.1 Timed Transition Systems and Behavioural Relations

A *timed transition system (TTS)* is a tuple $\mathcal{S} = (S, s_0, \Sigma, \mathbb{K}, \rightarrow)$, where S is the set of *states*, $s_0 \in S$ the initial state, Σ a finite alphabet, $\mathbb{K} \subseteq \mathbb{R}_{\geq 0}$ the time domain which contains 0 and is closed under addition, and $\rightarrow \subseteq S \times (\Sigma \cup \mathbb{K}) \times S$ the *transition* relation. We write $s \xrightarrow{\sigma} s'$ instead of $(s, \sigma, s') \in \rightarrow$; we also write $s \xrightarrow{d,\sigma} s'$ if $s \xrightarrow{d} s'' \xrightarrow{\sigma} s'$ for $d \in \mathbb{K}$, $\sigma \in \Sigma$ and some state s'', and $s \overset{\sigma}{\Longrightarrow} s'$

if $s \xrightarrow{d',\sigma} s'$ for some $d' \in \mathbb{K}$. A *run* ρ of $\mathcal{S}$ is a finite or infinite sequence $q_0 \xrightarrow{\tau_0} q'_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\tau_1} q'_1 \xrightarrow{\sigma_1} \ldots$, where $q_i \in S$, $\sigma_i \in \Sigma$ and $\tau_i \in \mathbb{K}$ for all i. The word $\sigma_0\sigma_1 \ldots \in \Sigma^*$ is the *trace* of ρ. We denote by $\mathsf{Trace}(\mathcal{S})$ the set of finite and infinite traces of the runs of $\mathcal{S}$. We define the set of *reachable states* of $\mathcal{S}$, denoted by $\mathsf{Reach}(\mathcal{S})$, as the set of states s' for which some finite run of $\mathcal{S}$ starts from state s_0 and ends in state s'. A run written on the form $\gamma = q_0 \xrightarrow{d_0,\sigma_0} q_1 \xrightarrow{d_1,\sigma_1} q_2 \ldots$ is a *timed-action path* (or simply path). Each state $q_0 \in S$ admits a set $P(q_0)$ of paths starting at q_0. For any path γ, the suffix γ^j is obtained by deleting the first j transitions in γ, and $\gamma(j) = q_j \xrightarrow{d_j,\sigma_j} q_{j+1}$ is the j-th transition in γ; we also let $\mathsf{state}_j(\gamma) = q_j$, $\gamma(j)_\sigma = \sigma_j$, and $\gamma(j)_d = d_j$.

We consider a quantitative extension of timed bisimilarity introduced in [22]. This spans the gap between timed and time-abstract bisimulations: while the former requires time delays to be matched exactly, the latter ignores timing information altogether. Intuitively, we define two states to be ε-bisimilar, for a given parameter $\varepsilon \geq 0$, if there is a (time-abstract) bisimulation which relates these states in such a way that, at each step, the difference between the time delays of corresponding delay transitions is at most ε. Thus, this parameter allows one to quantify the "timing error" made during the bisimulation. A strong and a weak variant of this notion is given in the following definition.

Definition 1. *Given a TTS $(S, s_0, \Sigma, \mathbb{K}, \rightarrow)$, and $\varepsilon \geq 0$, a symmetric relation $R_\varepsilon \subseteq S \times S$ is a*

- strong timed ε-bisimulation, *if for any $(s,t) \in R_\varepsilon$ and $\sigma \in \Sigma, d \in \mathbb{K}$,*
  - $s \xrightarrow{\sigma} s'$ *implies* $t \xrightarrow{\sigma} t'$ *for some* $t' \in S$ *with* $(s',t') \in R_\varepsilon$,
  - $s \xrightarrow{d} s'$ *implies* $t \xrightarrow{d'} t'$ *for some* $t' \in S$ *and* $d' \in \mathbb{K}$ *with* $|d - d'| \leq \varepsilon$ *and* $(s',t') \in R_\varepsilon$.
- timed-action ε-bisimulation, *if for any $(s,t) \in R_\varepsilon$, and $\sigma \in \Sigma$, $d \in \mathbb{K}$,*
  - $s \xrightarrow{d,\sigma} s'$ *implies* $t \xrightarrow{d',\sigma} t'$ *for some* $t' \in S$ *and* $d' \in \mathbb{K}$ *with* $|d - d'| \leq \varepsilon$ *and* $(s',t') \in R_\varepsilon$.

If there exists a strong timed ε-bisimulation (resp. timed-action ε-bisimulation) R_ε such that $(s,t) \in R_\varepsilon$, then we write $s \sim_\varepsilon t$ (resp. $s \approx_\varepsilon t$). Furthermore we write $s \sim_{\varepsilon^+} t$ (resp. $s \approx_{\varepsilon^+} t$) whenever for every $\varepsilon' > \varepsilon$, $s \sim_{\varepsilon'} t$ (resp. $s \approx_{\varepsilon'} t$).

Observe that $s \sim_\varepsilon t$ implies $s \sim_{\varepsilon'} t$ for every $\varepsilon' > \varepsilon$. Also, $s \sim_{\varepsilon^+} t$ does not imply $s \sim_\varepsilon t$ in general (see Fig. 1), and if $s \sim_{\varepsilon^+} t$ but $s \not\sim_\varepsilon t$, then $\varepsilon = \inf\{\varepsilon' > 0 \mid s \sim_{\varepsilon'} t\}$. These observations hold true in the timed-action bisimulation setting as well. Note also that $s \sim_\varepsilon t$ implies $s \approx_\varepsilon t$. Finally, for $\varepsilon > 0$, strong timed or timed-action ε-bisimilarity relations are not equivalence relations in general, but they are when $\varepsilon = 0$.

Last, we define a variant of *ready-simulation* [18] for timed transition systems. For $\mathsf{Bad} \subseteq \Sigma$, we will write $I \sqsubseteq^{\mathsf{Bad}} S$ when I is simulated by S (and time delays are matched exactly) in such a way that at any time during the simulation, any failure (*i.e.*, any action in Bad) enabled in S is also enabled in I. So, if $I \sqsubseteq^{\mathsf{Bad}} S$ and S is safe w.r.t. Bad (*i.e.*, Bad actions are never enabled), then any

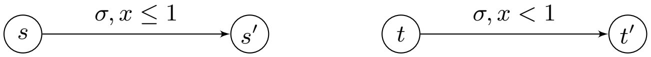


Fig. 1. An automaton in which $(s,0) \sim_{0+} (t,0)$ but $(s,0) \not\sim_0 (t,0)$

run of I can be executed in S (with exact timings) without enabling any of the Bad-actions. Fig. 2 will provide an automaton illustrating the importance of this notion. More formally:

Definition 2. *Given a TTS* $(S, s_0, \Sigma, \mathbb{K}, \rightarrow)$*, and a set* $\mathsf{Bad} \subseteq \Sigma$*, a relation* $R \subseteq S \times S$ *is a* ready-simulation w.r.t. Bad *if, whenever* $(s,t) \in R$*:*

- *for all* $\sigma \in \Sigma$ *and* $d \in \mathbb{K}$*,* $s \xrightarrow{d,\sigma} s'$ *implies* $t \xrightarrow{d,\sigma} t'$ *for some* $t' \in S$ *with* $(s',t') \in R$*,*
- *for all* $\sigma \in \mathsf{Bad}$*,* $t \overset{\sigma}{\Longrightarrow} t'$ *implies* $s \overset{\sigma}{\Longrightarrow} s'$ *for some* $s' \in S$*.*

We write $s \sqsubseteq^{\mathsf{Bad}} t$ *if* $(s,t) \in R$ *for some ready-simulation* R *w.r.t.* Bad*.*

2.2 Timed Automata

Given a set of clocks $\mathcal{C}$, the elements of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ are referred to as *valuations*. For a subset $X \subseteq \mathcal{C}$, and a valuation v, we define $v[X \leftarrow 0]$ as the valuation $v[X \leftarrow 0](x) = v(x)$ for all $x \in \mathcal{C} \setminus X$ and $v[X \leftarrow 0](x) = 0$ for $x \in X$. For any $d \in \mathbb{R}_{\geq 0}$, $v + d$ is the valuation defined by $(v+d)(x) = v(x) + d$ for all $x \in \mathcal{C}$. For any $\alpha \in \mathbb{R}$, we define αv as the valuation obtained by multiplying all components of v by α, that is $(\alpha v)(x) = \alpha v(x)$ for all $x \in \mathcal{C}$. Given two valuations v and v', we denote by $v + v'$ the valuation that is the componentwise sum of v and v', that is $(v+v')(x) = v(x) + v'(x)$ for all $x \in \mathcal{C}$.

Let $\mathbb{Q}_\infty = \mathbb{Q} \cup \{-\infty, \infty\}$. An *atomic clock constraint* is a formula of the form $k \preceq x \preceq' l$ or $k \preceq x - y \preceq' l$ where $x, y \in \mathcal{C}$, $k, l \in \mathbb{Q}_{\geq 0}$ and $\preceq, \preceq' \in \{<, \leq\}$. A *guard* is a conjunction of atomic clock constraints. For $M, \eta \in \mathbb{Q}_{>0}$ such that $\frac{1}{\eta} \in \mathbb{N}$, we denote by $\Phi_{\mathcal{C}}(\eta, M)$ the set of guards on the clock set $\mathcal{C}$, whose constants are either $\pm\infty$ or less than or equal to M in absolute value and are integer multiples of η. Let $\Phi_{\mathcal{C}}$ denote the set of all guards on clock set $\mathcal{C}$. A valuation v satisfies $\varphi \in \Phi_{\mathcal{C}}$ if all atomic clock constraints of φ are satisfied when each $x \in \mathcal{C}$ is replaced by $v(x)$. Let $[\![\varphi]\!]$ denote the set of valuations that satisfy φ. We define the *enlargement* of atomic clock constraints by $\Delta \in \mathbb{Q}$ as

$$\begin{aligned} &\langle k \preceq x - y \preceq' l \rangle_\Delta = k - \Delta \preceq x - y \preceq' l + \Delta, \\ \text{and} \qquad &\langle k \preceq x \preceq' l \rangle_\Delta = k - \Delta \preceq x \preceq' l + \Delta. \end{aligned}$$

for $x, y \in \mathcal{C}$ and $k, l \in \mathbb{Q}_{>0}$. The enlargement of a guard φ, denoted by $\langle \varphi \rangle_\Delta$, is obtained by enlarging all its atomic clock constraints.

Definition 3. *A timed automaton* $\mathcal{A}$ *is a tuple* $(\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$*, consisting of a finite set* $\mathcal{L}$ *of locations, a finite set* $\mathcal{C}$ *of clocks, a finite alphabet* Σ *of labels, a finite set* $E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times \Sigma \times 2^{\mathcal{C}} \times \mathcal{L}$ *of edges, and an initial location* $l_0 \in \mathcal{L}$*.*

We write $l \xrightarrow{\varphi,\sigma,R} l'$ if $e = (l, \varphi, \sigma, R, l') \in E$, and call φ the guard of e. $\mathcal{A}$ is an integral *timed automaton if all constants that appear in its guards are integers.*

We call the inverses of positive integers *granularities*. The *granularity* of a timed automaton is the inverse of the least common denominator of the finite constants in its guards. For any timed automaton $\mathcal{A}$ and rational $\Delta \geq 0$, let $\mathcal{A}_\Delta$ denote the timed automaton obtained from $\mathcal{A}$ where each guard φ is replaced with $\langle\varphi\rangle_\Delta$.

Definition 4. *The semantics of a timed automaton $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$ is a TTS over alphabet Σ, denoted $[\![\mathcal{A}]\!]$, whose state space is $\mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$. The initial state is $(l_0, \mathbf{0})$, where $\mathbf{0}$ denotes the valuation where all clocks have value* 0. Delay transitions *are defined as $(l, v) \xrightarrow{\tau} (l, v+\tau)$ for any state (l, v) and $\tau \in \mathbb{K}$.* Action transitions *are defined as $(l, v) \xrightarrow{\sigma} (l', v')$, for any edge $l \xrightarrow{g,\sigma,R} l'$ in $\mathcal{A}$ such that $v \models g$ and $v' = v[R \leftarrow 0]$.*

For any $k \in \mathbb{N}_{>0}$, we define the sampled semantics *of $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]^{\frac{1}{k}}$ as the TTS defined similarly to $[\![\mathcal{A}]\!]$ by taking the time domain as $\mathbb{K} = \frac{1}{k}\mathbb{N}$.*

We write $[\![\mathcal{A}]\!] \sim_\varepsilon [\![\mathcal{B}]\!]$, $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{B}]\!]$ and $[\![\mathcal{A}]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{B}]\!]$ when the initial states of timed automata $\mathcal{A}$ and $\mathcal{B}$ are related accordingly in the disjoint union of the transition systems, defined in the usual way.

We define the usual notion of region equivalence [3]. Let M be the maximum (rational) constant that appears in the guards of $\mathcal{A}$, let η be the granularity of $\mathcal{A}$. Multiplying any constant in $\mathcal{A}$ by $\frac{1}{\eta}$, we obtain an integral timed automaton.

Given valuations $u, v \in \mathbb{R}^{\mathcal{C}}_{\geq 0}$ and rationals M, η, define $v \simeq^M_\eta u$ to hold if, and only if, for all formulas $\varphi \in \Phi_{\mathcal{C}}(\eta, M)$, $u \models \varphi$ if and only if $v \models \varphi$. The equivalence class of a valuation u for the relation $\simeq^M_\eta$ is denoted by $\mathsf{reg}(u)^M_\eta = \{v \mid u \simeq^M_\eta v\}$. Each such class is called an (η, M)-region. In the rest, when constant M is (resp. M and η are) clear from context, we simply write $\mathsf{reg}(u)_\eta$ (resp. $\mathsf{reg}(u)$) and call these η-regions (resp. regions). We denote by $\overline{\mathsf{reg}(u)^M_\eta}$ the topological closure of $\mathsf{reg}(u)^M_\eta$. The number of (η, M)-regions is bounded by $O(2^{|\mathcal{C}|}|\mathcal{C}|!(M/\eta)^{|\mathcal{C}|})$ [3].

For a region r, we denote by $r[R \leftarrow 0]$, the region obtained by resetting clocks in R. We define $\mathsf{tsucc}^*(r)$ as the set of *time-successor regions of* r, that is, the set of η-regions r' such that $u + d \in r'$ for some $u \in r$ and $d \in \mathbb{R}_{\geq 0}$.

We now associate with each (η, M)-region a guard that defines it. Assume we number the clocks with indices so that $\mathcal{C} = \{x_1, \ldots, x_m\}$, and fix any (η, M)-region r. Let us define $x_0 = 0$, and $\mathcal{C}_0 = \mathcal{C} \cup \{x_0\}$. Then, for each pair $i, j \in \mathcal{C}_0$, there exists a number $A_{i,j} \in \eta\mathbb{Z} \cap [-M, M] \cup \{\infty\}$ and $\preceq_{i,j} \in \{<, \leq\}$ s.t. φ_r, defined as

$$\varphi_r = \bigwedge_{(x_i, x_j) \in \mathcal{C}_0} -A_{j,i} \preceq_{j,i} x_i - x_j \preceq_{i,j} A_{i,j},$$

is such that $[\![\varphi_r]\!] = r$. Moreover, we assume that for all $i, j, k \in \mathcal{C}_0$, $A_{i,i} = 0$ and $A_{i,j} \leq A_{i,k} + A_{k,j}$. Note that this is a standard definition: the matrix $(A_{i,j})_{i,j}$ is a *difference-bound matrix (DBM)* that defines region r, and the latter condition defines its *canonical form* [13]. Later we will refer to matrix $(A_{i,j})_{i,j}$ as the DBM that defines region r.

2.3 Quantitative Extension of Computation Tree Logic

In the style of [10,16,14] we present a quantitative extension of CTL, which measures (in a sense that we make clear below) how far a formula is from being satisfied in a given state.

Definition 5. *Let $\mathbb{I}$ be the set of closed nonempty intervals of $\mathbb{R}_{\geq 0}$, and Σ be a finite alphabet. We define the set of state- and path-formulas as follows*[1]

$$\Psi ::= \top \mid \bot \mid \Psi_1 \wedge \Psi_2 \mid \Psi_1 \vee \Psi_2 \mid \mathsf{E}\Pi \mid \mathsf{A}\Pi$$
$$\Pi ::= \mathsf{X}^I_A \Psi \mid \bar{\mathsf{X}}^I_A \Psi \mid \Psi_1 \mathsf{R}^I \Psi_2 \mid \Psi_1 \mathsf{U}^I \Psi_2$$

for $I \in \mathbb{I}$ and $A \subseteq \Sigma$. We write $\mathcal{L}_T(\Sigma)$ or simply $\mathcal{L}_T$ for the set of state formulae.

To define the semantics of $\mathcal{L}_T$, we introduce the distance between a point and an interval: $|z,[x,y]| = 0$ when $z \in [x,y]$, and $|z,[x,y]| = \min\{|x-z|,|y-z|\}$ otherwise. Now, given a state s, the value of a state formula is defined inductively as follows: $\llparenthesis \top \rrparenthesis(s) = 0$, $\llparenthesis \bot \rrparenthesis(s) = \infty$, and

$$\llparenthesis \psi_1 \vee \psi_2 \rrparenthesis(s) = \inf\left\{\llparenthesis \psi_1 \rrparenthesis(s), \llparenthesis \psi_2 \rrparenthesis(s)\right\} \qquad \llparenthesis \mathsf{E}\pi \rrparenthesis(s) = \inf\left\{\llparenthesis \pi \rrparenthesis(\gamma) \mid \gamma \in P(s)\right\}$$

$$\llparenthesis \psi_1 \wedge \psi_2 \rrparenthesis(s) = \sup\left\{\llparenthesis \psi_1 \rrparenthesis(s), \llparenthesis \psi_2 \rrparenthesis(s)\right\} \qquad \llparenthesis \mathsf{A}\pi \rrparenthesis(s) = \sup\left\{\llparenthesis \pi \rrparenthesis(\gamma) \mid \gamma \in P(s)\right\}$$

For a path γ, it is defined as:

$$\begin{aligned}
&\llparenthesis \mathsf{X}^I_A \psi \rrparenthesis(\gamma) = \llparenthesis \bar{\mathsf{X}}^I_A \psi \rrparenthesis(\gamma) = \max\{|\gamma(0)_d, I|, \llparenthesis \psi \rrparenthesis(\mathsf{state}_1(\gamma))\} && \text{if } \gamma(0)_\sigma \in A \\
&\llparenthesis \mathsf{X}^I_A \psi \rrparenthesis(\gamma) = +\infty \quad \text{and} \quad \llparenthesis \bar{\mathsf{X}}^I_A \psi \rrparenthesis(\gamma) = 0 && \text{if } \gamma(0)_\sigma \notin A
\end{aligned}$$

$$\llparenthesis \psi_1 \mathsf{U}^I \psi_2 \rrparenthesis(\gamma) = \inf_k \left(\max\left\{ \max_{0 \leq j < k} |\llparenthesis \psi_1 \rrparenthesis(\mathsf{state}_j(\gamma)), I|,\ \llparenthesis \psi_2 \rrparenthesis(\mathsf{state}_k(\gamma)) \right\} \right)$$

$$\llparenthesis \psi_1 \mathsf{R}^I \psi_2 \rrparenthesis(\gamma) = \sup_k \left(\min\left\{ \max_{0 \leq j < k} |\llparenthesis \psi_1 \rrparenthesis(\mathsf{state}_j(\gamma)), I|, \llparenthesis \psi_2 \rrparenthesis(\mathsf{state}_k(\gamma)) \right\} \right)$$

For instance, $\llparenthesis \mathsf{E}\mathsf{X}^{[2,5]}_{\{a\}} \top \rrparenthesis(s)$ is the lower bound of the set

$$\{\big|d,[2,5]\big| \mid \text{there is a transition } s \xrightarrow{d,a} s'\}.$$

Intuitively, this semantics measures the amount of point-wise modifications (in the timing constraints of the formula) that are needed for this formula to hold at a given state. Notice that untimed[2] formulas of $\mathcal{L}_T$ can only be evaluated to 0 or $+\infty$, and this value reflects the Boolean value of the underlying CTL formula.

It is shown in [22] that $\mathcal{L}_T$ characterizes ε-bisimilarity between the states of *weighted Kripke structures*. In the following proposition, we generalize one direction of this result to timed automata, showing that ε-bisimilar states have close satisfaction values for all formulas of $\mathcal{L}_T$, which implies that these properties (and their values) are preserved upto ε by the constructions we give in Section 4.

[1] To establish the relationship to timed-action ε-bisimulation, the logic uses actions on transitions instead of the more usual atomic propositions on states. It is easy to encode the latter by adding extra transitions to sink states.

[2] *I.e.*, when all timing constraints are $[0,+\infty)$.

Proposition 1. *For any timed automaton $\mathcal{A}$ and states s, s' of $[\![\mathcal{A}]\!]$, for all $\varepsilon \geq 0$, if $s \approx_{\varepsilon+} s'$ then for any $\psi \in \mathcal{L}_T$ either $(\!|\psi|\!)(s) = (\!|\psi|\!)(s') = \infty$ or $|(\!|\psi|\!)(s) - (\!|\psi|\!)(s')| \leq \varepsilon$.*

3 Implementability

As explained in the introduction, even the smallest enlargement of the guards may yield extra behaviour in timed automata. Similarly, any sampling of the time domain may remove behaviours. Here, we give several definitions of *robustness* and *samplability*, which distinguish timed automata whose enlargement (resp. whose sampled semantics) is ε-bisimilar to the original automaton, for some ε.

3.1 Robustness

Earlier work on robustness based on enlargement, such as [11,6,7] concentrated on deciding the existence of a positive Δ under which the enlarged automaton is correct w.r.t. a given property. Here, we consider a stronger notion of robustness, which requires systems to be ε-bisimilar for some ε.

Definition 6. *A timed automaton $\mathcal{A}$ is* ε-bisimulation-robust *(or simply ε-robust), where $\varepsilon \geq 0$, if there exists $\Delta > 0$ such that $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{A}_\Delta]\!]$.*

Note that not all timed automata are robust. In fact, in the automaton $\mathcal{A}$ of Fig. 2, location ℓ_3 is not reachable in $[\![\mathcal{A}]\!]$, but it becomes reachable in $[\![\mathcal{A}_\Delta]\!]$ for any $\Delta > 0$ (see [11]).

We do not know whether a timed automaton that is robust for some Δ is still robust for any $\Delta' < \Delta$, that is, whether $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{A}_\Delta]\!]$ implies $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{A}_{\Delta'}]\!]$ for $\Delta' < \Delta$, in general. This is the so-called "faster-is-better" property [2,12], which means that if a property holds in some platform, it also holds in a faster or more precise platform. This is known to be satisfied for simpler notions of robustness mentioned above.

In the next section, we will present our construction which, for any $\mathcal{A}$, produces an alternative automaton $\mathcal{A}'$ which is robust and satisfies $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{A}'_\Delta]\!]$ for all small enough Δ.

Bisimulation is not always sufficient when one wants to preserve state-based safety properties proven for $\mathcal{A}$. For instance, removing edges leading to unsafe states in $\mathcal{A}$ may provide us with a trivially safe automaton under any enlargement. However, edges leading to such states are used to detect failures, so removing these will not necessarily remove the failure (since the states that immediately trigger a failure may still be reachable). Fig. 3 gives such an "incorrect" construction. To cope with this problem, we rely on ready-simulation and require $\mathcal{A}'$ to satisfy $[\![\mathcal{A}'_\Delta]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$, where Bad are distinguished actions leading to unsafe states in $\mathcal{A}$. This means that any run of $[\![\mathcal{A}'_\Delta]\!]$ can be realized in $[\![\mathcal{A}_\Delta]\!]$, and that no Bad-action is enabled in that run in the latter (and hence no unsafe state is reached). Thus, intuitively, no state reached in $[\![\mathcal{A}'_\Delta]\!]$ corresponds to an unsafe state in $[\![\mathcal{A}_\Delta]\!]$, and in particular, if $\mathcal{A}_\Delta$ has unsafe runs (leading

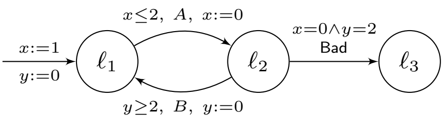


Fig. 2. A non-robust timed automaton [20]

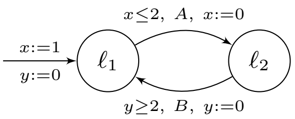


Fig. 3. A robust but unsafe alternative

to unsafe states), then these cannot be realized in $\mathcal{A}'_\Delta$. Clearly, the automaton in Fig. 3 does not satisfy this. We formalize this idea here.

Definition 7. *A timed automaton* $\mathcal{A}$ *is* safe w.r.t. a set of actions $\mathsf{Bad} \subseteq \Sigma$*, if* $d_\infty\big(\mathsf{Reach}(\llbracket\mathcal{A}\rrbracket), \mathit{Pre}(\mathsf{Bad})\big) > 0$*, where* d_∞ *is the standard supremum metric, and* $\mathit{Pre}(\mathsf{Bad}) = \bigcup_{\sigma\in\mathsf{Bad},(l,\sigma,g,R,l')\in E} \{l\} \times \llbracket g \rrbracket$ *is the precondition of* Bad*-actions*

Notice that $\mathit{Pre}(\mathsf{Bad})$ is the set of states from which a Bad action can be done, and that $d_\infty(\mathsf{Reach}\llbracket\mathcal{A}\rrbracket, \mathit{Pre}(\mathsf{Bad})) = 0$ does not imply that a state of $\mathit{Pre}(\mathsf{Bad})$ is reachable in $\mathcal{A}$. But we still consider such an automaton as unsafe, since, intuitively, any enlargement of the guards may lead to a state of $\mathit{Pre}(\mathsf{Bad})$. It can be seen that automaton of Fig. 2 is safe w.r.t. action Bad. Note that a closed timed automaton is safe w.r.t. Bad iff Bad is not reachable.

Recall the standard notion of robustness, used *e.g.* in [11]:

Definition 8. *A timed automaton* $\mathcal{A}$ *is* safety-robust (w.r.t. Bad) *if there exists* $\Delta > 0$ *such that* $\llbracket\mathcal{A}_\Delta\rrbracket$ *is safe w.r.t.* Bad*.*

In the rest, Bad will refer to a set of actions given with the timed automaton we consider. When we say that a timed automaton is safe, or safety-robust, these actions will be implicit.

We introduce the notion of *safety-robust implementation* (parameterized by a bisimilarity relation $\equiv$, which will range over $\{\sim_0, \sim_{0^+}, \approx_0, \approx_{0^+}\}$), where we only require the alternative automaton to preserve a given safety specification.

Definition 9 (Safety-Robust Implementation). *Let* $\mathcal{A}$ *be a timed automaton which is safe w.r.t. actions* Bad*, and* $\equiv$ *denote any bisimilarity relation. A* safety-robust implementation of $\mathcal{A}$ w.r.t $\equiv$ *is a timed automaton* $\mathcal{A}'$ *such that:*

(i) $\mathcal{A}'$ *is safety-robust;*
(ii) $\llbracket\mathcal{A}'\rrbracket \equiv \llbracket\mathcal{A}\rrbracket$*;*
(iii) *there exists* $\Delta_0 > 0$ *s.t. for all* $0 < \Delta' < \Delta < \Delta_0$, $\llbracket\mathcal{A}'_{\Delta'}\rrbracket \sqsubseteq^{\mathsf{Bad}} \llbracket\mathcal{A}_\Delta\rrbracket$.

Now we define the notion of *robust implementation.* We require such an implementation to be robust and equivalent to the original automaton, and to preserve safety specifications.

Definition 10 (Robust Implementation). *Let* $\mathcal{A}$ *be a timed automaton which is safe w.r.t.actions* Bad*, and* $\equiv$ *denote any bisimilarity. An* ε-robust implementation of $\mathcal{A}$ w.r.t. $\equiv$ *is a timed automaton* $\mathcal{A}'$ *such that:*

(i) $\mathcal{A}'$ *is* ε*-robust;*
(ii) $[\![\mathcal{A}']\!] \equiv [\![\mathcal{A}]\!]$*;*
(iii) *there exists* $\Delta_0 > 0$ *s.t. for all* $0 < \Delta' < \Delta < \Delta_0$*,* $[\![\mathcal{A}'_{\Delta'}]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_{\Delta}]\!]$.

3.2 Samplability

As we noted in the introduction, some desired behaviours of a given timed automaton may be removed in the sampled semantics. Preservation of the untimed language under some sampling rate was shown decidable in [1]. The proof is highly technical (it is based on the limitedness problem for a special kind of counter automata).We are interested in the stronger notion of *bisimulation-samplability*, which, in particular, implies the preservation of untimed language.

Definition 11. *A timed automaton is said to be* ε-bisimulation-samplable *(or simply* ε-samplable*) if there exists a granularity* η *such that* $[\![\mathcal{A}]\!] \approx_\varepsilon [\![\mathcal{A}]\!]^\eta$*.*

Note that not all timed automata are bisimulation-samplable: [17] describes timed automata $\mathcal{A}$ which are not (time-abstract) bisimilar to their sampled semantics for any granularity η. We define a *sampled implementation* as follows.

Definition 12 (Sampled Implementation). *Let* $\mathcal{A}$ *be a timed automaton, and* $\equiv$ *denote any bisimilarity relation. A* ε-sampled implementation w.r.t. $\equiv$ *is a timed automaton* $\mathcal{A}'$ *such that*

(i) $\mathcal{A}'$ *is* ε*-samplable;*
(ii) $[\![\mathcal{A}']\!] \equiv [\![\mathcal{A}]\!]$.

Note that a similar phenomenon as in Fig. 2 does not occur in sampled semantics since sampling does not add extra behaviour, but may only remove some.

3.3 Main Result of the Paper

We will present two constructions which yield an implementation for any timed automaton. In our first construction, for any timed automaton given with a safety specification, we construct a safety robust implementation. Our second construction is stronger: Given any timed automaton $\mathcal{A}$ and any desired $\varepsilon > 0$, we construct a timed automaton $\mathcal{A}'$ which is both an ε-robust implementation and an ε-sampled implementation of $\mathcal{A}$ w.r.t. $\approx_{0^+}$ (we also give a variant w.r.t. $\sim_0$ for robustness).

Since, $\mathcal{A}$ and $\mathcal{A}'_\Delta$ are timed-action ε-bisimilar, the satisfaction values of the formulas in $\mathcal{L}_T$ are preserved up to ε (Proposition 1). In particular, all standard untimed linear- and branching-time properties (e.g. expressible in LTL, resp. CTL) proven for the original automaton are preserved in the implementation. An example of such a property is deadlock-freedom, which is an important property of programs.

Theorem 1. *Let* $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$ *be an integral timed automaton which is safe w.r.t. some set* $\mathsf{Bad} \subseteq \Sigma$*. Let* W *denote the number of regions of* $\mathcal{A}$*. Then,*

1. *There exists a safety robust implementation of $\mathcal{A}$ w.r.t $\sim_0$, with $|\mathcal{L}|$ locations, the same number of clocks and at most $|E| \cdot W$ edges.*
2. *For all $\varepsilon > 0$, there exists a timed automaton $\mathcal{A}'$ which is a ε-robust implementation w.r.t. $\sim_0$; and a timed automaton $\mathcal{A}''$ which is both a ε-sampled and ε-robust implementation w.r.t. $\approx_{0^+}$. Both timed automata have the same number of clocks as $\mathcal{A}$, and the number of their locations and edges is bounded by $O(|\mathcal{L}| \cdot W \cdot (\frac{1}{\varepsilon})^{|\mathcal{C}|})$.*

The rest of the paper is devoted to the proof of this theorem. The two constructions are presented in the next section, and proved thereafter.

4 Making Timed Automata Robust and Samplable

For any timed automaton $\mathcal{A}$ and any location l of $\mathcal{A}$, let $\mathsf{Reach}([\![\mathcal{A}]\!])|_l$ denote the projection of the set of reachable states at location l to $\mathbb{R}^{\mathcal{C}}_{\geq 0}$. For any l, there exist guards $\varphi^l_1, \ldots, \varphi^l_{n_l}$ such that $\bigcup_i [\![\varphi^l_i]\!] = \mathsf{Reach}([\![\mathcal{A}]\!])|_l$ (in fact, the set of reachable states at a given location is a union of regions but not necessarily convex). We use these formulas to construct a new automaton where we restrict all transitions to be activated only at reachable states.

Definition 13. *Let $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$ be any integral timed automaton. Define timed automaton $\mathsf{safe}(\mathcal{A})$ from $\mathcal{A}$ by replacing each edge $l \xrightarrow{\varphi,\sigma,R} l'$, by edges $l \xrightarrow{\varphi\wedge\varphi^l_i,\sigma,R} l'$ for all $i \in \{1, \ldots, n_l\}$.*

As stated in Theorem 1 the worst-case complexity of this construction is exponential. However, in practice, $\mathsf{Reach}([\![\mathcal{A}]\!])|_l$ may have a simple shape, which can be captured by few formulas φ^l_i.

Although the above construction will be enough to obtain a safety-robust timed automaton w.r.t. a given set Bad, it may not be bisimulation-robust. The following construction ensures this.

Definition 14. *Let $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$ be an integral timed automaton. Let M be the largest constant that appears in $\mathcal{A}$, and let η be any granularity. We define $\mathsf{impl}_\eta(\mathcal{A})$ as a timed automaton over the set of locations l_r where l is a location of $\mathcal{A}$ and r is an (η, M)-region, and over the same set of clocks. Edges are defined as follows. Whenever there is an edge $l \xrightarrow{\varphi,\sigma,R} l'$ in $\mathcal{A}$, we let $l_r \xrightarrow{\varphi\wedge\varphi_s,\sigma,R} l'_{s[R\leftarrow 0]}$, for all (η, M)-regions r and $s \in \mathsf{tsucc}^*(r)$ such that $[\![\varphi_s]\!] \subseteq [\![\varphi]\!]$.*

We define $\overline{\mathsf{impl}_\eta}(\mathcal{A})$ as the closed timed automaton obtained from $\mathsf{impl}_\eta(\mathcal{A})$ where each guard is replaced by its closed counterpart[3].

Throughout this paper, we always consider integral timed automata as input, and the only non-integer constants are those added by our construction. Observe that the size of $\mathsf{impl}_\eta(\mathcal{A})$ depends on η, since a smaller granularity yields a greater number of (η, M)-regions.

[3] That is, all $<$ are replaced by $\leq$, and $>$ by $\geq$.

The main theorem is a direct corollary of the following lemma, where we state our results in detail. The bounds on the size of the constructed implementations follow by construction.

Lemma 1. *Let $\mathcal{A} = (\mathcal{L}, \mathcal{C}, \Sigma, l_0, E)$ be an integral timed automaton and fix any $\varepsilon > 0$. Assume that $\mathcal{A}$ is safe w.r.t. some set* $\mathsf{Bad} \subseteq \Sigma$. *Then,*

1. $\mathsf{safe}(\mathcal{A})$ *is safety-robust,* $[\![\mathcal{A}]\!] \sim_0 [\![\mathsf{safe}(\mathcal{A})]\!]$ *and for any* $\Delta < \frac{1}{2|\mathcal{C}|}$, $[\![\mathsf{safe}(\mathcal{A})_\Delta]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$.
2. *For any granularity η and $\Delta > 0$ such that $2(\eta + \Delta) < \varepsilon$, we have* $[\![\mathcal{A}]\!] \approx_{0+} [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!]$ *and* $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!] \approx_\varepsilon [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta]\!]$. *Moreover, for any* $0 < \Delta' < \Delta < \frac{1}{|\mathcal{C}|}$, $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_{\Delta'}]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$.
3. *For any granularity η and $\Delta > 0$ such that $2(\eta + \Delta) < \varepsilon$, we have* $[\![\mathcal{A}]\!] \sim_0 [\![\mathsf{impl}_\eta(\mathcal{A})]\!]$ *and* $[\![\mathsf{impl}_\eta(\mathcal{A})]\!] \approx_\varepsilon [\![\mathsf{impl}_\eta(\mathcal{A})_\Delta]\!]$. *Moreover, whenever* $\Delta < \frac{1}{|\mathcal{C}|}$, $[\![\mathsf{impl}_\eta(\mathcal{A})_\Delta]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$.
4. *For any granularities η and α such that $\eta = k\alpha$ for some $k \in \mathbb{N}_{>0}$ and $\eta < \varepsilon/2$,* $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!] \approx_\varepsilon [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!]^\alpha$.

Note that both $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ and $\mathsf{impl}_\eta(\mathcal{A})$ provide the relation $\approx_{\varepsilon+}$ between the specification (that is, $[\![\mathcal{A}]\!]$) and the implementation (that is, $[\![\mathcal{A}'_\Delta]\!]$). However, the latter has a stronger relation with $[\![\mathcal{A}]\!]$, so we also study it separately.

Trading Precision against Complexity. The choice of the granularity in $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ and $\mathsf{impl}_\eta(\mathcal{A})$ allows one to obtain an implementation of $\mathcal{A}$ with any desired precision. However, this comes with a cost since the size of $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ is exponential in the granularity η. But it is also possible to give up on precision in order to reduce the size of the implementation. In fact, one could define $\mathsf{impl}_\equiv(\mathcal{A})$ where the regions are replaced by the equivalence classes of any finite time-abstract bisimulation $\equiv$. Then, we get $[\![\mathcal{A}]\!] \approx_0 [\![\mathsf{impl}_\equiv(\mathcal{A})]\!]$ and $[\![\mathsf{impl}_\equiv(\mathcal{A})]\!]$ is time-abstract bisimilar to $[\![\mathsf{impl}_\equiv(\mathcal{A})_\Delta]\!]$ for any $\Delta > 0$. In order to obtain, say $[\![\mathsf{impl}_\equiv(\mathcal{A})]\!] \approx_K [\![\mathsf{impl}_\equiv(\mathcal{A})_\Delta]\!]$, for some desired $K \geq 1$, one could, roughly, split these bisimulation classes to sets of delay-width at most $O(K)$, that is the maximal delay within a bounded bisimulation class (there is a subtlety with unbounded classes, where, moreover, all states must have arbitrarily large time-successors within the class). Note however that safety specifications are only guaranteed to be preserved for small enough K (see Lemma 1).

5 Proof of Correctness

This section is devoted to the proof of Lemma 1. We start with general properties of regions, in subsection 5.1. In subsection 5.2, we prove the robustness of $\mathsf{impl}_\eta(\mathcal{A})$, $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ and $\mathsf{safe}(\mathcal{A})$, as stated in points 1 through 3 of Lemma 1. In subsection 5.3, we prove that $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ is bisimulation-samplable (point 4). Last, the ready simulation is proved for all the systems in subsection 5.4.

5.1 Properties of Regions

We give several properties of the enlargement of regions. Fixing constants η and M, we refer to any (η, M)-region simply as a region.

Proposition 2. *Let $u \in \mathbb{R}^{\mathcal{C}}_{\geq 0}$ such that $u \in [\![\langle \varphi_s \rangle_\Delta]\!]$ for some region s. Then for any subset of clocks $R \subseteq C$, $u[R \leftarrow 0] \in [\![\langle \varphi_{s[R\leftarrow 0]} \rangle_\Delta]\!]$.*

The following proposition shows, intuitively, that enlarged guards cannot distinguish the points of an "enlarged region". The proof is straightforward using difference bound matrices in canonical form. Note that the property does not hold if φ_s is not in canonical form.

Proposition 3. *Let s denote a region, and φ a guard. If $[\![\varphi_s]\!] \subseteq [\![\varphi]\!]$, then $[\![\langle \varphi_s \rangle_\Delta]\!] \subseteq [\![\langle \varphi \rangle_\Delta]\!]$.*

Proposition 4. *Let $u \in \mathbb{R}^{\mathcal{C}}_{\geq 0}$ such that $u \in [\![\langle \varphi_s \rangle_\Delta]\!]$ for some region s. Then for all $s' \in \mathsf{tsucc}^*(s)$, there exists $d \geq 0$ such that $u + d \in [\![\langle \varphi_{s'} \rangle_\Delta]\!]$.*

The previous proposition is no longer valid if φ_s is not canonical. As an example, take the region defined by $x = 1 \wedge y = 0$, whose immediate successor is $1 < x < 2 \wedge 0 < y < 1 \wedge x - y = 1$. The enlargement of the former formula is satisfied by valuation $(x = 1 - \Delta, y = \Delta)$ but this has no time-successor that satisfies the enlargement of the latter.

Last, we need the following proposition which provides a bound on the delay that it takes to go from a region to another.

Proposition 5. *Let r be a region, and s a time-successor region of r, and $\Delta \geq 0$. Suppose that $u \in [\![\varphi_r]\!]$ and $u + d \in [\![\varphi_s]\!]$ for some $d \geq 0$. Then for any $v \in [\![\langle \varphi_r \rangle_\Delta]\!]$, there exists $d' \geq 0$ such that $v + d' \in [\![\langle \varphi_s \rangle_\Delta]\!]$ and $|d' - d| \leq 2\eta + 2\Delta$.*

5.2 Proof of Robustness

We first prove that $\mathsf{impl}_\eta(\mathcal{A})$ and $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ are bisimulation-robust, for an appropriate ε, that is $[\![\mathcal{A}']\!] \approx_\varepsilon [\![\mathcal{A}'_\Delta]\!]$ where $\mathcal{A}'$ denotes any of these (Lemma 2). Then we show "faithfulness" results: Lemma 3 shows that $[\![\mathcal{A}]\!] \sim_0 [\![\mathsf{impl}_\eta(\mathcal{A})]\!]$ and $[\![\mathsf{safe}(\mathcal{A})]\!] \sim_0 [\![\mathcal{A}]\!]$, and Lemma 4 shows that $[\![\mathcal{A}]\!] \approx_{0+} [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!]$.

Lemma 2. *For any timed automaton $\mathcal{A}$, any granularity η, and any $\Delta > 0$, we have $[\![\mathsf{impl}_\eta(\mathcal{A})]\!] \approx_{2\Delta+2\eta} [\![\mathsf{impl}_\eta(\mathcal{A})_\Delta]\!]$ and $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!] \approx_{2\Delta+2\eta} [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta]\!]$.*

Proof (Sketch). We fix any η and Δ. Let us consider $\mathsf{impl}_\eta(\mathcal{A})$. The case of $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ is similar. We define relation $\mathcal{R} \subseteq (\mathcal{L} \times \mathbb{R}^C) \times (\mathcal{L} \times \mathbb{R}^C)$ between $[\![\mathsf{impl}_\eta(\mathcal{A})]\!]$ and $[\![\mathsf{impl}_\eta(\mathcal{A})_\Delta]\!]$ by $(l_r, u)\mathcal{R}(l'_{r'}, u')$ whenever $l_r = l'_{r'}$ and

$$\forall s \in \mathsf{tsucc}^*(r), \exists d \geq 0, u + d \in [\![\varphi_s]\!] \iff \exists d' \geq 0, u' + d' \in [\![\langle \varphi_s \rangle_\Delta]\!]. \qquad (1)$$

Intuitively, relation $\mathcal{R}$ relates states which can reach, by a delay, the same set of regions: we require the first system to reach $[\![\varphi_s]\!]$, while it is sufficient that the second one reaches $\llbracket\langle\varphi_s\rangle_\Delta\rrbracket$, since its guards are enlarged by Δ. Then, Propositions 2, 3, and 4 ensure that this relation is maintained after each transition, proving that $\mathcal{R}$ is a timed-action bisimulation. The parameter $2\Delta + 2\eta$ is given by Proposition 5, applied on relation (1). □

The parameter which we provide for the timed-action bisimilarity is (almost) tight. In fact, consider the automaton in Figure 2, where the guard of the edge entering ℓ_1 is changed to $x \leq 1$. Fix any η and Δ and consider the following cycle in $\overline{\mathsf{impl}}_\eta(\mathcal{A})$: $(\ell_{1,r_1}) \to (\ell_{2,r_2}) \to (\ell_{1,r_1})$, where r_1 is the region $1-\eta < x < 1 \wedge y = 0$, and r_2 is the region $x = 0 \wedge 1 < y < 1+\eta$. Suppose $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta\rrbracket$ first goes to location (ℓ_{1,r_1}) with $x = 1+\Delta, y = 0$, and that this is matched in $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$ by $(\ell_{1,r'_1}, (x = 1-\alpha, y = 0))$ where necessarily $\alpha \geq 0$. It is shown in [11] that in any such cycle, the enlarged automaton can reach (by iterating the cycle) all states of the region r_1 at location ℓ_1. In particular, $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta\rrbracket$ can go to state $(\ell_{1,r_1}, (x = 1-\eta, y = 0))$. However, without enlargement, all states $(\ell_1, r'_1, (v'_x, v'_y))$ reached from a state $(\ell_{1,r_1}, (v_x, v_y))$ with $v_y = 0$ satisfy $v'_x \geq v_x$, that is, the value of the clock x at location ℓ_1 cannot decrease along any run ([11]). Thus, the state $(\ell_{1,r_1}, (x = 1-\eta, y = 0))$ of $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta\rrbracket$ is matched in $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$ by some state $(\ell_{1,r''_1}, (v'_x, 0))$ where $v'_x \geq 1-\alpha$. Now, from there, $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta\rrbracket$ can delay $1+\Delta+\eta$ and go to ℓ_2, whereas $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$ can delay at most $1+\alpha$ to take the same transition. The difference between the delays at the first and the last step is then at least $\max\big(\Delta+\alpha, 1+\Delta+\eta-(1+\alpha)\big) \geq \Delta+\eta/2$.

Next, we show that $\mathsf{safe}(\mathcal{A})$ and $\mathsf{impl}_\eta(\mathcal{A})$ are strongly 0-bisimilar to $\mathcal{A}$. The proof is omitted.

Lemma 3. *For any timed automaton $\mathcal{A}$, we have $\llbracket\mathsf{safe}(\mathcal{A})\rrbracket \sim_0 \llbracket\mathcal{A}\rrbracket$, and $\llbracket\mathcal{A}\rrbracket \sim_0 \llbracket\mathsf{impl}_\eta(\mathcal{A})\rrbracket$ for any granularity η.* □

The proof of $\llbracket\mathcal{A}\rrbracket \approx_{0^+} \llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$ is trickier. In fact, since all guards are closed in $\overline{\mathsf{impl}}_\eta(\mathcal{A})$, but not necessarily in $\mathcal{A}$, all time delays may not be matched exactly. The first part of the proof follows the lines of Proposition 16 of [19], who, by a similar construction, prove that the finite timed traces of $\llbracket\mathcal{A}\rrbracket$ are dense in those of $\llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$, for an appropriate topology. Their result has a similar flavor, but we consider 0^+-bisimulation which cannot be interpreted in terms of density in an obvious way.

Lemma 4. *For any timed automaton $\mathcal{A}$ and granularity η, $\llbracket\mathcal{A}\rrbracket \approx_{0^+} \llbracket\overline{\mathsf{impl}}_\eta(\mathcal{A})\rrbracket$.*

Proof (Sketch). We fix any η and $\delta \in (0,1)$. We define $(l,v)\mathcal{R}(l_r,v')$ iff

$$r = \mathsf{reg}(v), v' \in \overline{\mathsf{reg}(v)} \text{ and } \exists v'' \in \mathsf{reg}(v), v = \delta v'' + (1-\delta)v'. \qquad (2)$$

We show that $\mathcal{R}$ is a timed-action 0^+-bisimulation. One direction of the bisimulation follows from convexity of regions, while the other direction is less obvious, and necessitates the following technical lemma. □

Lemma 5. *Let $v, v', v'' \in \mathbb{R}_{\geq 0}$ such that $v'' \in \mathsf{reg}(v)$ and $v' \in \overline{\mathsf{reg}(v)}$, and $v = \varepsilon v'' + (1-\varepsilon)v'$ for some $\varepsilon \in (0,1)$. Then for all $d \geq 0$, there exists $d', d'' \geq 0$ s.t. $v+d = \varepsilon(v''+d'') + (1-\varepsilon)(v'+d')$, $v''+d'' \in \mathsf{reg}(v+d)$ and $v'+d' \in \overline{\mathsf{reg}(v+d)}$.*

5.3 Proof of Samplability

We now show that $\overline{\mathsf{impl}}_\eta(\mathcal{A})$ is a sampled implementation for any timed automaton $\mathcal{A}$. This result follows from the following lemma and Lemma 4. The proof is similar to Lemma 2.

Lemma 6. *Let $\mathcal{A}$ be any integral timed automaton. For any granularities η and α such that $\eta = k\alpha$ for some $k \in \mathbb{N}_{>0}$, we have $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!] \approx_{2\eta} [\![\overline{\mathsf{impl}}_\eta(\mathcal{A})]\!]^\alpha$.*

5.4 Proof of Safety Preservation (Ready Simulation)

Lemma 7. *We have $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_{\Delta'}]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$ and $[\![\mathsf{impl}_\eta(\mathcal{A})_\Delta]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$ for any $0 < \Delta' < \Delta < \frac{1}{|C|}$; and $[\![\mathsf{safe}(\mathcal{A})_\Delta]\!] \sqsubseteq^{\mathsf{Bad}} [\![\mathcal{A}_\Delta]\!]$ for any $\Delta < \frac{1}{2|C|}$.*

Proof (Sketch). The simulation can be shown similarly to Lemma 3. We show that actions Bad are not enabled in any state of the simulating run, whenever $\mathcal{A}$ is safe w.r.t. Bad. Let us consider the first statement. Informally, this is due to two facts: (1) the set of reachable states in $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta]\!]$ have a small distance (at most Δ) to the corresponding reachable states in $[\![\mathcal{A}]\!]$; (2) the states of $[\![\mathcal{A}]\!]$ have a positive distance to $\mathrm{Pre}_{\mathcal{A}}(\mathsf{Bad})$, which can be bounded from below by $\frac{1}{|C|}$. Thus, choosing $\frac{1}{|C|} - \Delta > 0$, we prove that $[\![\overline{\mathsf{impl}}_\eta(\mathcal{A})_{\Delta'}]\!]$ is also safe w.r.t. Bad. □

6 Conclusion

We have presented a way to transform any timed automaton into robust and samplable ones, while preserving the original semantics with any desired precision. Such a transformation is interesting if the timed automaton under study is not robust (or not samplable), or cannot be certified as such. In this case, one can simply model-check the original automaton for desired properties, then apply our constructions, which will preserve the specification.

Our constructions also allow one to solve the *robust synthesis* problem. In the synthesis problem, the goal is to obtain automatically (i.e. to synthesize) a timed automaton which satisfies a given property. If one solves this problem for timed automata and obtain a synthesized system $\mathcal{A}$, then applying our constructions, we get that $\overline{\mathsf{impl}}_\eta(\mathcal{A})_\Delta$ and $\overline{\mathsf{impl}}_\eta(\mathcal{A})^\eta$ satisfy the same (say, untimed) properties.

As a future work, we will be interested in *robust controller synthesis*. In this problem, we are given a system $\mathcal{S}$ which we cannot change, and we are asked to synthesize a system $\mathcal{C}$, called controller, such that the parallel composition of the two satisfies a given property. The *robust* controller synthesis is the controller synthesis problem where the behaviour of the controller is $\mathcal{C}_\Delta$ (the controller has imprecise clocks), and we need to decide whether there is some Δ for which the parallel composition still satisfies the property.

Acknowledgement. We thank David N. Jansen for his detailed and insightful comments.

References

1. Abdulla, P., Krčál, P., Yi, W.: Sampled semantics of timed automata. Logical Methods in Computer Science 6(3:14) (2010)
2. Altisen, K., Tripakis, S.: Implementation of timed automata: An issue of semantics or modeling? In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 273–288. Springer, Heidelberg (2005)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Asarin, E., Maler, O., Pnueli, A.: On discretization of delays in timed automata and digital circuits. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 470–484. Springer, Heidelberg (1998)
5. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T., Größer, M.: Probabilistic and topological semantics for timed automata. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 179–191. Springer, Heidelberg (2007)
6. Bouyer, P., Markey, N., Reynier, P.-A.: Robust model-checking of linear-time properties in timed automata. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 238–249. Springer, Heidelberg (2006)
7. Bouyer, P., Markey, N., Reynier, P.-A.: Robust analysis of timed automata *via* channel machines. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 157–171. Springer, Heidelberg (2008)
8. Bozga, M., Maler, O., Pnueli, A., Yovine, S.: Some progress in the symbolic verification of timed automata. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 179–190. Springer, Heidelberg (1997)
9. Cassez, F., Henzinger, T.A., Raskin, J.-F.: A comparison of control problems for timed and hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)
10. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1022–1037. Springer, Heidelberg (2003)
11. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. Formal Methods in System Design 33(1-3), 45–84 (2008)
12. De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: From timed models to timed implementations. Formal Aspects of Computing 17(3), 319–341 (2005)
13. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
14. Fahrenberg, U., Larsen, K.G., Thrane, C.: A quantitative characterization of weighted Kripke structures in temporal logic. Journal of Computing and Informatics 29(6+), 1311–1324 (2010)
15. Gupta, V., Henzinger, T.A., Jagadeesan, R.: Robust timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 331–345. Springer, Heidelberg (1997)
16. Henzinger, T.A., Majumdar, R., Prabhu, V.: Quantifying similarities between timed systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 226–241. Springer, Heidelberg (2005)
17. Krčál, P., Pelánek, R.: On sampled semantics of timed systems. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 310–321. Springer, Heidelberg (2005)
18. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. In: POPL 1989, pp. 344–352 (1989)

19. Ouaknine, J., Worrell, J.: Revisiting digitization, robustness, and decidability for timed automata. In: LICS 2003, pp. 198–207. IEEE Computer Society, Los Alamitos (2003)
20. Puri, A.: Dynamical properties of timed systems. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)
21. Sankur, O., Bouyer, P., Markey, N.: Shrinking timed automata (2011) (submitted)
22. Thrane, C., Fahrenberg, U., Larsen, K.G.: Quantitative analysis of weighted transition systems. Journal Logic and Algebraic Programming 79(7), 689–703 (2010)

Coarse Abstractions Make Zeno Behaviours Difficult to Detect

Frédéric Herbreteau and B. Srivathsan

Université de Bordeaux, IPB, Université Bordeaux 1, CNRS, LaBRI UMR5800
Bât A30, 351 crs Libération, 33405 Talence, France

Abstract. An infinite run of a timed automaton is Zeno if it spans only a finite amount of time. Such runs are considered unfeasible and hence it is important to detect them, or dually, find runs that are non-Zeno. Over the years important improvements have been obtained in checking reachability properties for timed automata. We show that some of these very efficient optimizations make testing for Zeno runs costly. In particular we show NP-completeness for the LU-extrapolation of Behrmann et al. We analyze the source of this complexity in detail and give general conditions on extrapolation operators that guarantee a (low) polynomial complexity of Zenoness checking. We propose a slight weakening of the LU-extrapolation that satisfies these conditions.

1 Introduction

Timed automata [1] are finite automata augmented with a finite number of clocks. The values of the clocks increase synchronously along with time in the states of the automaton and these values can be compared to a constant and reset to zero while crossing a transition. This model has been successfully used for verification of timed systems thanks to a number of tools [3,6,15].

Since timed automata model reactive systems that continuously interact with the environment, it is interesting to consider questions related to their infinite executions. An execution is said to be *Zeno* if an infinite number of events happen in a finite time interval. Such executions are clearly unfeasible. During verification, the aim is to detect if there exists a *non-Zeno* execution that violates a certain property. On the other hand while implementing timed automata, it is required to check the presence of pathological Zeno executions. This brings the motivation to analyze an automaton for the presence of such executions.

The analysis of timed automata faces the challenge of handling its uncountably many configurations. To tackle this problem, one considers a finite graph called the *abstract zone graph* (also known as simulation graph) of the automaton. This finite graph captures the semantics of the automaton. In this paper, we consider the problems of deciding if an automaton has a non-Zeno execution, dually a Zeno execution, given its abstract zone graph as input.

An abstract zone graph is obtained by over-approximating each zone of the so-called *zone graph* with an abstraction function. The zone graph in principle could be infinite and an abstraction function is necessary for reducing it to a

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 92–107, 2011.

finite graph. The coarser the abstraction, the smaller the abstract zone graph, and hence the quicker the analysis of the automaton. This has motivated a lot of research towards finding coarser abstraction functions [2]. The classic maximum-bound abstraction uses as a parameter the maximal constant a clock gets compared to in a transition. A coarser abstraction called the LU-extrapolation was introduced in Behrmann et al. [2] for checking state reachability in timed automata. This is the coarsest among all the implemented approximations and is at present efficiently used in tools like UPPAAL.

It was shown in [13,14] that even infinite executions of the automaton directly correspond to infinite paths in the abstract zone graph when one uses the maximum-bound approximation. In addition, it was proved that the existence of a non-Zeno infinite execution could be determined by adding an extra clock to the automaton to keep track of time and analyzing the abstract zone graph of this transformed automaton. A similar correspondence was established in the case of the LU-extrapolation by Li [11]. These results answer our question about deciding non-Zeno infinite executions of the automaton from its abstract zone graph. However, it was shown in [10] that adding a clock has an exponential worst case complexity. A new polynomial construction was proposed for the case of the classic maximum-bound approximation. But, the case of the LU-extrapolation was not addressed.

In this paper, we prove that the non-Zenoness question turns out to be NP-complete for the LU-extrapolation, that is, given the abstract zone graph over the LU-extrapolation, deciding if the automaton has a non-Zeno execution is NP-complete. We study the source of this complexity in detail and give conditions on abstraction operators to ensure a polynomial complexity. To this regard, we extend the polynomial construction given in [10] to an arbitrary abstraction function and analyze when it stays polynomial. It then follows that a slight weakening of the LU-extrapolation makes the construction polynomial. In the second part of the paper, we repeat the same for the dual question: given an automaton's abstract zone graph, decide if it has Zeno executions. Yet again, we notice NP-completeness for the LU-extrapolation. We introduce an algorithm for checking Zenoness over an abstract zone graph with conditions on the abstraction operator to ensure a polynomial complexity. We provide a different weakening of LU-extrapolation that gives a polynomial solution to the Zenoness question.

Related Work. As mentioned above, the LU-extrapolation was proposed in [2] and shown how it could be efficiently used in UPPAAL for the purpose of reachability. The correctness of the classic maximum-bound abstraction was shown in [4]. Extensions of these results to infinite executions occur in [14,11]. The trick involving adding an extra clock for non-Zenoness is discussed in [12,14,10]. For the case of checking existence of Zeno runs in timed automata, a bulk of the literature directs to [8,5]. They provide a sufficient-only condition for the absence of Zeno runs. This is different from our proposed solution which gives a complete solution (necessary and sufficient conditions) by analyzing the abstract zone graph of the automaton.

Organization of the Paper. We start with the formal definitions of timed automata, abstract zone graphs, the Zenoness and Non-Zenoness problems in Section 2. Subsequently, we prove the NP-completeness of the non-Zenoness problem for the LU-extrapolation in Section 3. We then recall the construction proposed for non-Zenoness in [10] and extend it to a general abstraction operator giving conditions for polynomial complexity. Section 5 talks about the dual Zenoness problem and Section 6 concludes the paper with some perspectives.

2 Zeno-Related Problems for Timed Automata

2.1 Timed Automata

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers. Let X be a set of variables, named *clocks* hereafter. A *valuation* is a function $\nu : X \mapsto \mathbb{R}_{\geq 0}$ that maps every clock in X to a non-negative real value. We denote the set of all valuations by $\mathbb{R}_{\geq 0}^X$, and $\mathbf{0}$ the valuation that maps every clock in X to 0. For $\delta \in \mathbb{R}_{\geq 0}$, we denote $\nu + \delta$ the valuation mapping each $x \in X$ to the value $\nu(x) + \delta$. For a subset R of X, let $[R]\nu$ be the valuation that sets x to 0 if $x \in R$ and assigns $\nu(x)$ otherwise. A *clock constraint* is a conjunction of constraints $x\#c$ for $x \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$. We denote $\Phi(X)$ the set of clock constraints over clock variables X. For a valuation ν and a constraint ϕ we write $\nu \vDash \phi$ when ν satisfies ϕ, that is, when ϕ holds after replacing every x by $\nu(x)$.

A *Timed Automaton (TA)* [1] $\mathcal{A}$ is a finite automaton extended with clocks that enable or disable transitions. Formally, $\mathcal{A}$ is a tuple (Q, q_0, X, T) where Q is a finite set of states, $q_0 \in Q$ is the initial state, X is a finite set of clocks and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions. For each transition $(q, g, R, q') \in T$, g is a guard that defines the valuations of the clocks that allow to cross the transition, and R is a set of clocks that are reset on the transition.

A configuration of $\mathcal{A}$ is a pair $(q, \nu) \in Q \times \mathbb{R}_{\geq 0}^X$. A transition $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$ with $t = (q, g, R, q') \in T$ and $\delta \in \mathbb{R}_{\geq 0}$ is enabled when $\nu + \delta \vDash g$ and $\nu' = [R](\nu + \delta)$. A *run* ρ of $\mathcal{A}$ is a (finite or infinite) sequence of transitions starting from the initial configuration: $(q_0, \mathbf{0}) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots (q_i, \nu_i) \xrightarrow{\delta_i, t_i} \cdots$

Definition 1 (Zeno/non-Zeno runs). *A run* $(q_0, \mathbf{0}) \xrightarrow{\delta_0, t_0} \ldots (q_i, \nu_i) \xrightarrow{\delta_i, t_i} \ldots$ *is* non-Zeno *if time diverges, that is,* $\sum_{i \geq 0} \delta_i = \infty$*. Otherwise it is* Zeno*.*

Notice that only infinite sequences can be non-Zeno. As can be seen, the number of configurations (q, ν) could be uncountable. We now define the abstract semantics for timed automata.

2.2 Symbolic Semantics, Zenoness and Non-zenoness Problems

A *zone* is a set of clock valuations that satisfy a conjunction of constraints of the form $x_i\#c$ and $x_i - x_j\#c$ with $x_i, x_j \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$. For instance, $(x_1 \leq 1 \wedge x_1 - x_2 \geq 0)$ is a zone. Zones can be efficiently represented by

Difference Bound Matrices (DBMs) [7]. A DBM representation of a zone Z is a $|X|+1$ square matrix $(Z_{ij})_{i,j\in[0;|X|]}$ where each entry $Z_{ij} = (c_{ij}, \preccurlyeq_{ij})$ represents the constraint $x_i - x_j \preccurlyeq_{ij} c_{ij}$ for $c_{ij} \in \mathbb{Z}\cup\{\infty\}$ and $\preccurlyeq_{ij} \in \{<, \leq\}$. A special clock x_0 encodes the value 0.

The *symbolic semantics* (or *zone graph*) of $\mathcal{A}$ is the transition system $ZG(\mathcal{A}) = (S, s_0, \Rightarrow)$ where S is the set of nodes (q, Z) with q a state of $\mathcal{A}$ and Z a zone; $s_0 = (q_0, Z_0)$ with $Z_0 = \{\mathbf{0} + \delta \,|\, \delta \in \mathbb{R}_{\geq 0}\}$ as the initial node. There exists a transition $(q, Z) \overset{t}{\Rightarrow} (q', Z')$ with $t = (q, g, R, q') \in T$ if Z' is the set of valuations $[R]\nu + \delta$ for some $\delta \in \mathbb{R}_{\geq 0}$ and some valuation $\nu \in Z$ such that $\nu \vDash g$. If Z is a zone, then Z' is a zone. Moreover, a DBM representation of Z' can be computed from the DBM representation of Z (see for instance [4]).

However $ZG(\mathcal{A})$ may still be infinite. Several abstractions have been introduced to obtain a finite graph from $ZG(\mathcal{A})$. A *finite abstraction* $\mathfrak{a}$ is a map from $\mathcal{P}(\mathbb{R}^X_{\geq 0})$ to $\mathcal{P}(\mathbb{R}^X_{\geq 0})$ such that for every zone Z: $\mathfrak{a}(Z)$ is a zone, $Z \subseteq \mathfrak{a}(Z)$, $\mathfrak{a}(\mathfrak{a}(Z)) = \mathfrak{a}(Z)$ and $\mathfrak{a}$ has a finite range. In particular Extra_M [4], Extra^+_M, Extra_{LU} and Extra^+_{LU} [2] are well-known finite abstractions. The last two abstractions are usually preferred as they are coarser and hence lead to more efficient algorithms. We recall the definitions below.

Let $L : X \mapsto \mathbb{N}\cup\{-\infty\}$ and $U : X \mapsto \mathbb{N}\cup\{-\infty\}$ be two maps that associate to each clock in $\mathcal{A}$ its maximal lower bound and its maximal upper bound respectively: that is, for every $x \in X$, $L(x)$ is the maximal integer c such that $x > c$ or $x \geq c$ appears in some guard of $\mathcal{A}$. We let $L(x) = -\infty$ if no such c exists. Similarly, we define $U(x)$ with respect to clock constraints like $x \leq c$ and $x < c$. We define $\mathsf{Extra}_{LU}(Z) = Z^{LU}$ and $\mathsf{Extra}^+_{LU}(Z) = Z^{LU+}$ as:

$$Z^{LU}_{ij} = \begin{cases} (\infty, <) & \text{if } c_{ij} > L(x_i) \\ (-U(x_j), <) & \text{if } -c_{ij} > U(x_j) \\ Z_{ij} & \text{otherwise} \end{cases}$$

$$Z^{LU+}_{ij} = \begin{cases} (\infty, <) & \text{if } c_{ij} > L(x_i) \\ (\infty, <) & \text{if } -c_{0i} > L(x_i) \\ (\infty, <) & \text{if } -c_{0j} > U(x_j), i \neq 0 \\ (-U(x_j), <) & \text{if } -c_{0j} > U(x_j), i = 0 \\ Z_{ij} & \text{otherwise} \end{cases}$$

where $L(x_0) = U(x_0) = 0$ for the special clock x_0. The abstraction Extra_M (resp. Extra^+_M) is obtained from Extra_{LU} (resp. Extra^+_{LU}) by replacing every occurrence of L and U by M which maps every clock x to $\max(L(x), U(x))$. These abstractions compare in the following way.

Theorem 1 ([2]). *For each zone Z, we have: $Z \subseteq \mathsf{Extra}_M(Z) \subseteq \mathsf{Extra}^+_M(Z)$; $Z \subseteq \mathsf{Extra}_{LU}(Z) \subseteq \mathsf{Extra}^+_{LU}(Z)$ and $\mathsf{Extra}^+_M(Z) \subseteq \mathsf{Extra}^+_{LU}(Z)$.*

For two nodes (q, Z) and (q', Z'), we define the relation $(q, Z) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z')$ if $(q, Z) \overset{t}{\Rightarrow} (q', Z'')$ in $ZG(\mathcal{A})$, $Z = \mathfrak{a}(Z)$ and $Z' = \mathfrak{a}(Z'')$. The *abstract symbolic semantics* (or the *abstract zone graph*) of $\mathcal{A}$ is the transition system $ZG^{\mathfrak{a}}(\mathcal{A})$ induced by $\Rightarrow_{\mathfrak{a}}$ with the intial node $(q_0, \mathfrak{a}(Z_0))$, where (q_0, Z_0) is the initial node of $ZG(\mathcal{A})$. We denote by $ZG^{LU}(\mathcal{A})$ the abstract symbolic semantics when abstraction Extra_{LU} is considered, and $ZG^M(\mathcal{A})$ when the abstraction $\mathfrak{a}$ is Extra_M.

A *path* π in $ZG^{\mathfrak{a}}(\mathcal{A})$: $(q_0, Z'_0) \overset{t_0}{\Rightarrow}_{\mathfrak{a}} (q_1, Z'_1) \overset{t_1}{\Rightarrow}_{\mathfrak{a}} \cdots (q_i, Z'_i) \overset{t_i}{\Rightarrow}_{\mathfrak{a}} \cdots$ is a (finite or infinite) sequence of transitions. We say a run $(q_0, \mathbf{0}) \xrightarrow{\delta_0, t_0} \ldots (q_i, \nu_i) \xrightarrow{\delta_i, t_i} \ldots$

of $\mathcal{A}$ is an *instance* of a path π of $ZG^{\mathfrak{a}}(\mathcal{A})$ if they agree on the sequence of transitions $t_0, t_1, \ldots$, and if for every $i \geq 0$, (q_i, ν_i) and (q_i, Z'_i) coincide on q_i, and $\nu_i \in Z'_i$. By definition of Z'_i this implies $\nu_i + \delta_i \in Z'_i$. We say an abstraction $\mathfrak{a}$ is *sound* if every path can be instantiated as a run of $\mathcal{A}$. Conversely, $\mathfrak{a}$ is *complete* when every run of $\mathcal{A}$ is an instance of some path in $ZG^{\mathfrak{a}}(\mathcal{A})$.

A classical verification problem for Timed Automata is to answer state reachability queries. For that purpose, runs of $\mathcal{A}$ and paths in $ZG^{\mathfrak{a}}(\mathcal{A})$ are defined as *finite* sequences of transitions. A reachability query asks for the existence of a finite run leading to a given state. Such problems can be solved using $ZG^{\mathfrak{a}}(\mathcal{A})$ when $\mathfrak{a}$ is sound and complete and this is true for the classical abstractions.

Theorem 2 ([4,2]). *Extra_M, Extra^+_M, Extra_{LU} and Extra^+_{LU} are sound and complete for finite sequences of transitions.*

Liveness properties ask for the existence of an infinite run satisfying a given property. For instance, does $\mathcal{A}$ visit state q infinitely often? Soundness and completeness of $\mathfrak{a}$ with respect to infinite runs allow to solve such problems from $ZG^{\mathfrak{a}}(\mathcal{A})$. Recently, it has also been proved that classical abstractions are also sound and complete for infinite paths/runs.

Theorem 3 ([13,11]). *Extra_M, Extra^+_M, Extra_{LU} and Extra^+_{LU} are sound and complete for infinite sequences of transitions.*

Thanks to Theorem 3, we know that every path π in $ZG^{\mathfrak{a}}(\mathcal{A})$ can be instantiated to a run of $\mathcal{A}$. However, soundness is not sufficient to know if π can be instantiated as a *non-Zeno* run. In the sequel, we consider the following problems, given an automaton $\mathcal{A}$ and an abstract zone graph $ZG^{\mathfrak{a}}(\mathcal{A})$.

Input	$\mathcal{A}$ and $ZG^{\mathfrak{a}}(\mathcal{A})$
Non-Zenoness problem ($\mathsf{NZP}^{\mathfrak{a}}$)	Does $\mathcal{A}$ have a non-Zeno run?
Zenoness problem ($\mathsf{ZP}^{\mathfrak{a}}$)	Does $\mathcal{A}$ have a Zeno run?

Observe that solving $\mathsf{ZP}^{\mathfrak{a}}$ does not solve $\mathsf{NZP}^{\mathfrak{a}}$ and vice-versa: one is not the negation of the other. In this paper, we focus on the complexity of deciding $\mathsf{ZP}^{\mathfrak{a}}$ and $\mathsf{NZP}^{\mathfrak{a}}$ for different abstractions $\mathfrak{a}$. We denote NZP^M and ZP^M when abstraction Extra_M is considered. We similarly define NZP^{LU} and ZP^{LU} for abstraction Extra_{LU}. The non-Zenoness problem is solved in polynomial time when abstraction Extra_M is considered [10]. Surprisingly, this is not true for abstraction Extra_{LU}: in Section 3 we show that NZP^{LU} is NP-complete. The same asymmetry appears in the Zenoness problem as well, which is shown in Section 5.

3 Non-zenoness is NP-complete for Extra_{LU}

We give a reduction from the 3SAT problem: given a 3CNF formula ϕ, we build an automaton $\mathcal{A}^{NZ}_{\phi}$ that has a non-Zeno run iff ϕ is satisfiable. The size of the automaton will be linear in the size of ϕ. We will then show that the abstract zone graph $ZG^{LU}(\mathcal{A}^{NZ}_{\phi})$ is isomorphic to the automaton $\mathcal{A}^{NZ}_{\phi}$, thus completing the polynomial reduction from 3SAT to NZP^{LU}.

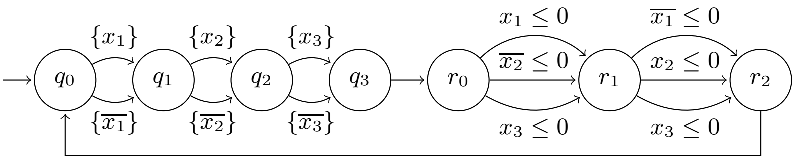


Fig. 1. $\mathcal{A}_\phi^{NZ}$ for $\phi = (p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$

Let $P = \{p_1, \dots, p_k\}$ be a set of propositional variables and let $\phi = C_1 \wedge \cdots \wedge C_n$ be a 3CNF formula with n clauses. We define the timed automaton $\mathcal{A}_\phi^{NZ}$ as follows. Its set of clocks X equals $\{x_1, \dots, x_k, \overline{x_1}, \dots, \overline{x_k}\}$. For a literal λ, let $cl(\lambda)$ denote the clock x_i when $\lambda = p_i$ and the clock $\overline{x_i}$ when $\lambda = \neg p_i$. The set of states of $\mathcal{A}_\phi^{NZ}$ is $\{q_0, \dots, q_k, r_0, \dots, r_n\}$ where q_0 is the initial state. The transitions are as follows:

- for each p_i we have transitions $q_{i-1} \xrightarrow{\{x_i\}} q_i$ and $q_{i-1} \xrightarrow{\{\overline{x_i}\}} q_i$,
- for each clause $C_m = \lambda_1^m \vee \lambda_2^m \vee \lambda_3^m$, $m = 1, \dots, n$, there are three transitions $r_{m-1} \xrightarrow{cl(\lambda_j^m) \leq 0} r_m$ where $\lambda_j^m \in \{\lambda_1^m, \lambda_2^m, \lambda_3^m\}$,
- transitions $q_k \rightarrow r_0$ and $r_n \rightarrow q_0$ with no guards and resets.

Figure 1 shows the automaton for the formula $(p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$. Intuitively, a reset of x_i represents $p_i \mapsto true$ and a reset of $\overline{x_i}$ means $p_i \mapsto false$. From r_0 to r_2 we check if the formula is satisfied by this guessed assignment. This formula is satisfied by every assignment that maps p_3 to *true*. This can be seen from the automaton by picking a cycle containing the transitions $q_2 \xrightarrow{\{x_3\}} q_3$, $r_0 \xrightarrow{x_3 \leq 0} r_1$ and $r_1 \xrightarrow{x_3 \leq 0} r_2$. On that path, time can elapse for instance in state q_0, since x_3 is reset before being zero-checked. Conversely, consider the assignment $p_1 \mapsto false$, $p_2 \mapsto true$ and $p_3 \mapsto false$ that does not satisfy the formula. Take a cycle that resets $\overline{x_1}$, x_2 and $\overline{x_3}$ corresponding to the assignment. Then none of the clocks that are checked for zero on the transitions from r_0 to r_1 has been reset. Notice that these transitions come from the first clause in the formula that evaluates to *false* according to the assignment. To take a transition from r_0, one of x_1, $\overline{x_2}$ and x_3 must be zero and hence time cannot elapse.

Lemma 1 below states that if the formula is satisfiable, there exists a sequence of resets that allows time elapse in every loop. Conversely, if the formula is unsatisfiable, in every iteration of the loop, there is a zero-check that prevents time from elapsing.

Lemma 1. *A 3CNF formula ϕ is satisfiable iff $\mathcal{A}_\phi^{NZ}$ has a non-Zeno run.*

Proof of Lemma 1 can be found in [9]. The NP-hardness of NZP^{LU} then follows due to the small size of $ZG^{LU}(\mathcal{A}_\phi^{NZ})$.

Theorem 4. *The abstract zone graph $ZG^{LU}(\mathcal{A}_\phi^{NZ})$ is isomorphic to $\mathcal{A}_\phi^{NZ}$. The non-Zenoness problem is* NP*-complete for abstractions* Extra_{LU} *and* Extra_{LU}^+.

Proof. We first prove that $ZG^{LU}(\mathcal{A}_\phi^{NZ})$ is isomorphic to $\mathcal{A}_\phi^{NZ}$. For every clock x, $L(x) = -\infty$, hence Extra_{LU} abstracts all the constraints $x_i - x_j \preccurlyeq_{ij} c_{ij}$ to $x_i - x_j < \infty$ except those of the form $x_0 - x_i \preccurlyeq_{0i} c_{0i}$ that are kept unchanged. Due to the guards in $\mathcal{A}_\phi^{NZ}$, for every reachable zone in $ZG(\mathcal{A}_\phi^{NZ})$ we have $x_0 - x_i \leq 0$ (i.e. $x_i \geq 0$). Therefore $\mathsf{Extra}_{LU}(Z)$ is the zone defined by $\bigwedge_{x \in X} x \geq 0$ which is $\mathbb{R}^X_{\geq 0}$. For each state of $\mathcal{A}_\phi^{NZ}$, the zone $\mathbb{R}^X_{\geq 0}$ is the only reachable zone in $ZG^{LU}(\mathcal{A}_\phi^{NZ})$, hence showing the isomorphism. The result transfers to Extra^+_{LU} thanks to Theorem 1.

The NP-hardness of NZP^{LU} then follows from Lemma 1. The membership to NP will be proved in Lemma 3 in the next section. □

Notice that the type of zero checks in $\mathcal{A}_\phi^{NZ}$ is crucial to Theorem 4. Replacing zero-checks of the form $x \leq 0$ by $x = 0$ does not modify the semantics of $\mathcal{A}_\phi^{NZ}$. However, this yields $L(x) = 0$ for every clock x. Hence, the constraints of the form $x_i - x_j \leq 0$ are not abstracted: Extra_{LU} then preserves the ordering among the clocks. Each sequence of clock resets leading from q_0 to q_k yields a distinct ordering on the clocks. Thus, there are exponentially many LU-abstracted zones with state q_k. As a consequence, the polynomial reduction from 3SAT is lost. We indeed provide in Section 4 below an algorithm for detecting non-Zeno runs from $ZG^{LU}(\mathcal{A})$ that runs in polynomial time when $L(x) = 0$ for every clock x.

4 Finding Non-zeno Runs

Recall the non-Zenoness problem ($\mathsf{NZP}^{\mathfrak{a}}$):

> Given an automaton $\mathcal{A}$ and its abstract zone graph $ZG^{\mathfrak{a}}(\mathcal{A})$, decide if $\mathcal{A}$ has a non-Zeno run.

A standard solution to this problem involves adding one auxiliary clock to $\mathcal{A}$ to detect non-Zenoness [13]. This solution was shown to cause an exponential blowup in [10]. In the same paper, a polynomial method has been proposed in the case of the Extra_M abstraction. We briefly recall this construction below.

An infinite run of the timed automaton could be Zeno due to two factors: *blocking clocks*, which are clocks that are bounded from above (i.e. $x \leq c$ for some $c > 0$) but are never reset in the run and *zero checks*, which are guards of the form $x \leq 0$ or $x = 0$ that prevent time elapse in the run. The method in [10] tackles these two problems as follows. Blocking clocks are handled by first detecting a maximal strongly connected component (SCC) of the zone graph and repeatedly discarding the transitions that bound some blocking clock until a non-trivial SCC with no such clocks is obtained. This algorithm runs in time polynomial for every abstraction that is sound and complete. For zero checks, a *guessing zone graph* construction has been introduced to detect nodes where time can elapse. We now extend this construction to an arbitrary abstraction.

4.1 Reduced Guessing Zone Graph $rGZG^{\mathfrak{a}}(\mathcal{A})$

The necessary and sufficient condition for time elapse in a node despite zero-checks is to have every reachable zero-check from that node preceded by a corresponding reset. The nodes of the guessing zone graph are triples (q, Z, Y) where $Y \subseteq X$ is the set of clocks that can potentially be checked for zero before being reset in a path from (q, Z, Y). In particular, in a node with $Y = \emptyset$ zero-checks do not hinder time elapse.

A clock that is never checked for zero need not be remembered in sets Y. In order to lift the construction in [10], we restrict Y sets to only contain clocks that can indeed be checked for zero. We say that a clock x is *relevant* if there exists a guard $x \leq 0$ or $x = 0$ in the automaton. We denote the set of relevant clocks by $\mathrm{Rl}(\mathcal{A})$. For a zone Z, let $\mathcal{C}_0(Z)$ denote the set of clocks x such that there exists a valuation $\nu \in Z$ with $\nu(x) = 0$. The clocks that can be checked for zero from (q, Z) lie in $\mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$.

Definition 2. *Let $\mathcal{A}$ be a timed automaton with clocks X. The* reduced guessing zone graph *$rGZG^{\mathfrak{a}}(\mathcal{A})$ has nodes of the form (q, Z, Y) where (q, Z) is a node in $ZG^{\mathfrak{a}}(\mathcal{A})$ and $Y \subseteq \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$. The initial node is $(q_0, Z_0, \mathrm{Rl}(\mathcal{A}))$, with (q_0, Z_0) the initial node of $ZG^{\mathfrak{a}}(\mathcal{A})$. For $t = (q, g, R, q')$, there is a transition $(q, Z, Y) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z', Y')$ with $Y' = (Y \cup R) \cap \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z')$ if there is $(q, Z) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z')$ in $ZG^{\mathfrak{a}}(\mathcal{A})$ and some valuation $\nu \in Z$ such that $\nu \vDash (\mathrm{Rl}(\mathcal{A}) - Y) > 0$ and $\nu \vDash g$. A new auxiliary letter τ is introduced that adds transitions $(q, Z, Y) \overset{\tau}{\Rightarrow}_{\mathfrak{a}} (q, Z, Y')$ for $Y' = \emptyset$ or $Y' = Y$.*

Observe that as we require $\nu \vDash (\mathrm{Rl}(\mathcal{A}) - Y) > 0$ and $\nu \vDash g$ for some $\nu \in Z$, a transition that checks $x \leq 0$ (or $x = 0$) is allowed from a node (q, Z, Y) only if $x \in Y$. Thus, from a node $(q, Z, \emptyset)$ every reachable zero-check $x = 0$ should be preceded by a transition that resets x, and hence adds it to the guess set. Such a node is called *clear*. Time can elapse in clear nodes. A variable x is *bounded* in a transition of $rGZG^{\mathfrak{a}}$ if the guard of the transition implies $x \leq c$ for some constant c. A path of $rGZG^{\mathfrak{a}}$ is said to be *blocked* if there is a variable that is bounded infinitely often and reset only finitely often by the transitions on the path. Otherwise the path is called *unblocked*. An unblocked path says that there are no blocking clocks to bound time and clear nodes suggest that inspite of zero-checks that might possibly occur in the future, time can still elapse. We get the following theorem.

Theorem 5. *A timed automaton $\mathcal{A}$ has a non-Zeno run iff there exists an unblocked path in $rGZG^{\mathfrak{a}}(\mathcal{A})$ visiting a clear node infinitely often.*

The proof of Theorem 5 is in the same lines as for the guessing zone graph in [10]. We provide a proof in [9].

4.2 Polynomial Algorithms for $\mathsf{NZP}^{\mathfrak{a}}$

Since we have a node in $rGZG^{\mathfrak{a}}(\mathcal{A})$ for every (q, Z) in $ZG^{\mathfrak{a}}(\mathcal{A})$ and every subset Y of $\mathrm{Rl}(\mathcal{A})$, it can in principle be exponentially bigger than $ZG^{\mathfrak{a}}(\mathcal{A})$. Below, we see that depending on abstraction $\mathfrak{a}$, not all subsets Y need to be considered.

Let X' be a subset of X. We say that a zone Z *orders the clocks in* X' if for all clocks $x, y \in X'$, Z implies that at least one of $x \leq y$ or $y \leq x$ hold.

Definition 3 (Weakly order-preserving abstractions). *An abstraction* $\mathfrak{a}$ weakly preserves orders *if for all clocks* $x, y \in \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$*,* $Z \vDash x \leq y$ *iff* $\mathfrak{a}(Z) \vDash x \leq y$*.*

It has been observed in [10] that all the zones that are reachable in the unabstracted zone graph $ZG(\mathcal{A})$ order the entire set of clocks X. Assume that $\mathfrak{a}$ weakly preserves orders, then for every reachable node (q, Z, Y) in $rGZG^{\mathfrak{a}}(\mathcal{A})$, the zone Z orders the clocks in $\mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$. We now show that Y is downward closed with respect to this order given by Z: for clocks $x, y \in \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$, if $Z \vDash x \leq y$ and $y \in Y$, then $x \in Y$. This entails that there are at most $|\mathrm{Rl}(\mathcal{A})| + 1$ downward closed sets to consider, thus giving a polynomial complexity.

Theorem 6. *Let* $\mathcal{A}$ *be a timed automaton. If* $\mathfrak{a}$ *weakly preserves orders, then the reachable part of* $rGZG^{\mathfrak{a}}(\mathcal{A})$ *is* $\mathcal{O}(|\mathrm{Rl}(\mathcal{A})|)$ *bigger than the reachable part of* $ZG^{\mathfrak{a}}(\mathcal{A})$*.*

Proof. We prove by induction on the transitions in $rGZG^{\mathfrak{a}}(\mathcal{A})$ that for every reachable node (q, Z, Y) the set Y is downward closed with respect to Z on the clocks in $\mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$. This is true for the initial node $(q_0, Z_0, \mathrm{Rl}(\mathcal{A}))$.

Now, assume that this is true for (q, Z, Y). Take a transition $(q, Z, Y) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z', Y')$ with $t = (q, g, R, q')$. By definition, $Y' = (Y \cup R) \cap \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z')$. Suppose $Z' \vDash x \leq y$ for some $x, y \in \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z')$ and suppose $y \in Y'$. This could mean $y \in Y$ or $y \in R$. If $y \in R$, then x is also in R since $Z' \vDash x \leq y$. If $y \notin R$ then we get $y \in Y$ and $Z \vDash x \leq y$. By hypothesis that Y is downward closed, $x \in Y$. In both cases $x \in Y'$. □

The following lemma shows that Extra_M and Extra_M^+ weakly preserve orders. Hence, $rGZG^M(\mathcal{A})$ yields a polynomial algorithm for NZP^M. Thanks to the reduction of the guessing zone graph to the relevant clocks, this algorithm is more efficient than the algorithm in [10] despite using the same abstraction.

Lemma 2. *The abstractions* Extra_M *and* Extra_M^+ *weakly preserve orders.*

Proof. It has been proved in [10] that Extra_M weakly preserves orders. We now prove this for Extra_M^+. Firstly note that for a clock x in $\mathrm{Rl}(\mathcal{A})$ we have $M(x) \geq 0$. Moreover if $x \in \mathcal{C}_0(Z)$ we have that Z is consistent with $x \leq 0$. Hence, for a clock $x \in \mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$, Z is consistent with $x \leq M(x)$. Therefore, by definition, $\mathsf{Extra}_M^+(Z)$ restricted to clocks in $\mathrm{Rl}(\mathcal{A}) \cap \mathcal{C}_0(Z)$ is identical to $\mathsf{Extra}_M(Z)$ restricted to the same set of clocks. Since Extra_M weakly preserves orders, we get that Extra_M^+ weakly preserves orders too. □

However, the polynomial complexity is not preserved by coarser abstractions Extra_{LU} and Extra^+_{LU}.

Lemma 3. *The abstractions* Extra_{LU} *and* Extra^+_{LU} *do not weakly preserve orders. The non-Zenoness problem is in* NP *for* Extra_{LU} *and* Extra^+_{LU}*.*

Proof. The proof of Theorem 4 gives an example that illustrates Extra_{LU} does not weakly preserve orders. This also holds for Extra^+_{LU} by Theorem 1.

For the NP membership, let N be the number of nodes in $ZG^{LU}(\mathcal{A})$. Let us non-deterministically choose a node (q, Z). We assume that (q, Z) is reachable as this can be checked in polynomial time on $ZG^{LU}(\mathcal{A})$.

We augment (q, Z) with an empty guess set of clocks. From $(q, Z, \emptyset)$, we non-deterministically simulate a path π of the (non-reduced) guessing zone graph [10] obtained from Definition 2 with $\mathrm{Rl}(\mathcal{A}) = X$ and $\mathcal{C}_0(Z) = X$ for every zone Z. We avoid taking τ transitions on this path. This ensures that the guess sets accumulate all the resets on π. During the simulation, we also keep track of a separate set U containing all the clocks that are bounded from above on a transition in π.

If during the simulation one reaches a node (q, Z, Y) such that $U \subseteq Y$, then we have a cycle $(q, Z, \emptyset) \Rightarrow^*_{\mathfrak{a}} (q, Z, Y) \overset{\tau}{\Rightarrow}_{\mathfrak{a}} (q, Z, \emptyset)$ that is unblocked and that visits a clear node infinitely often. Also, since (q, Z) is reachable in $ZG^{LU}(\mathcal{A})$, (q, Z, X) is reachable in the guessing zone graph. Then $(q, Z, \emptyset)$ is reachable from (q, Z, X) with a τ transition. From [10] and from the fact that Extra_{LU} and Extra^+_{LU} are sound and complete [2] we get a non-Zeno run of $\mathcal{A}$.

Notice that it is sufficient to simulate $N \times (|X| + 1)$ transitions since we can avoid visiting a node (q', Z', Y') twice in π. □

The abstraction Extra_{LU} does not weakly preserve orders in zones due to relevant clocks with $L(x) = -\infty$ and $U(x) \geq 0$. We show that this is the only reason for NP-hardness. We slightly modify Extra_{LU} to get an abstraction $\mathsf{Extra}_{\overline{L}U}$ that is coarser than Extra_M, but it still weakly preserves orders.

Definition 4 (Weak L bounds). *Let $\mathcal{A}$ be a timed automaton. Given the bounds $L(x)$ and $U(x)$ for every clock $x \in X$, the* weak lower bound $\overline{L}$ *is given by: $\overline{L}(x) = 0$ if $x \in \mathrm{Rl}(\mathcal{A})$, $L(x) = -\infty$ and $U(x) \geq 0$, and $\overline{L}(x) = L(x)$ otherwise.*

We denote $\mathsf{Extra}_{\overline{L}U}$ the Extra_{LU} abstraction obtained by choosing $\overline{L}$ instead of L. Notice that $\mathsf{Extra}_{\overline{L}U}$ and Extra_{LU} coincide when zero-checks are written $x = 0$ instead of $x \leq 0$ in the automaton. By definition of Extra_{LU}, we get the following.

Lemma 4. *The abstraction* $\mathsf{Extra}_{\overline{L}U}$ *weakly preserves orders.*

$\mathsf{Extra}_{\overline{L}U}$ coincides with Extra_{LU} for a wide class of automata. For instance, when the automaton does not have a zero-check, $\mathsf{Extra}_{\overline{L}U}$ is exactly Extra_{LU}, and the existence of a non-Zeno run can be decided in polynomial time.

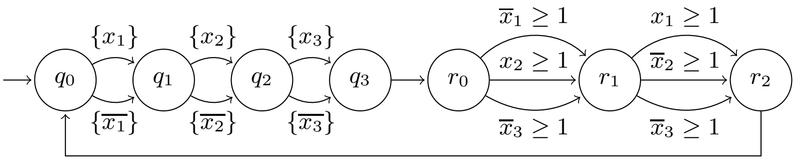


Fig. 2. $\mathcal{A}_\phi^Z$ for $\phi = (p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$

5 The Zenoness Problem

In this section we consider the Zenoness problem ($\mathsf{ZP}^{\mathfrak{a}}$):

> Given an automaton $\mathcal{A}$ and its abstract zone graph $ZG^{\mathfrak{a}}(\mathcal{A})$, decide if $\mathcal{A}$ has a Zeno run.

As in the case of non-Zenoness, this problem turns out to be NP-complete when the abstraction operator $\mathfrak{a}$ is Extra_{LU}. We subsequently give the hardness proof by providing a reduction from 3SAT.

5.1 Reducing 3SAT to $\mathsf{ZP}^{\mathfrak{a}}$ with Abstraction Extra_{LU}

Let $P = \{p_1, \dots, p_k\}$ be a set of propositional variables. Let $\phi = C_1 \wedge \dots \wedge C_n$ be a 3CNF formula with n clauses. Each clause C_m, $m = 1, 2, \dots, n$ is a disjunction of three literals λ_1^m, λ_2^m and λ_3^m. We construct in polynomial time an automaton $\mathcal{A}_\phi^Z$ and its zone graph $ZG^{LU}(\mathcal{A}_\phi^Z)$ such that $\mathcal{A}_\phi^Z$ has a Zeno run iff ϕ is satisfiable, thus proving the NP-hardness.

The automaton $\mathcal{A}_\phi^Z$ has clocks $\{x_1, \overline{x_1}, \dots, x_k, \overline{x_k}\}$ with x_i and $\overline{x_i}$ corresponding to the literals p_i and $\neg p_i$ respectively. We denote the clock associated to a literal λ by $cl(\lambda)$. The set of states of $\mathcal{A}_\phi^Z$ is given by $\{q_0, q_1, \dots, q_k\} \cup \{r_0, r_1, r_2, \dots, r_n\}$ with q_0 being the initial state. The transitions are as follows:

- transitions $q_{i-1} \xrightarrow{\{x_i\}} q_i$ and $q_{i-1} \xrightarrow{\{\overline{x_i}\}} q_i$ for $i = 1, 2, \dots, k$,
- a transition $q_k \rightarrow r_0$ with no guards and resets,
- for each clause C_m there are three transitions $r_{m-1} \xrightarrow{cl(\overline{\lambda})\ \geq 1} r_m$ where $\lambda = \{\lambda_1^m, \lambda_2^m, \lambda_3^m\}$,
- a transition $r_n \rightarrow q_0$ with no guards and resets. This transition creates a cycle in $\mathcal{A}_\phi^Z$.

As an example, Figure 2 shows the automaton for the formula $(p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$. Clearly, $\mathcal{A}_\phi^Z$ can be constructed from ϕ in $\mathcal{O}(|\phi|)$ time. It remains to show that $ZG^{LU}(\mathcal{A}_\phi^Z)$ can also be calculated in polynomial time from $\mathcal{A}_\phi^Z$ and to show that ϕ is satisfiable iff $\mathcal{A}_\phi^Z$ has a Zeno run.

Lemma 5. *A 3CNF formula ϕ is satisfiable iff $\mathcal{A}_\phi^Z$ has a Zeno run.*

The proof of Lemma 5 is given in [9]. We note that the size of the $ZG^{LU}(\mathcal{A})$ is the same as that of the automaton.

Theorem 7. *The zone graph $ZG^{LU}(\mathcal{A}_\phi^Z)$ is isomorphic to $\mathcal{A}_\phi^Z$. The Zenoness problem is* NP*-complete for* Extra_{LU} *and* Extra_{LU}^+.

Proof. By looking at the guards in the transitions, we get that for each clock x, $L(x) = 1$ and $U(x) = -\infty$. The initial node of the zone graph $ZG^{LU}(\mathcal{A}_\phi^Z)$ is $(q_0, \mathsf{Extra}_{LU}(Z_0))$ where Z_0 is the set of valuations given by $(x_1 \geq 0) \wedge (x_1 = \overline{x_1} = \cdots = x_k = \overline{x_k})$. By definition, since for each clock x, $U(x) = -\infty$, we have $\mathsf{Extra}_{LU}(Z_0) = \mathbb{R}^X_{\geq 0}$, the non-negative half-space.

On taking a transition with a guard $x \geq 1$ from $\mathbb{R}^X_{\geq 0}$, we come to a zone $\mathbb{R}^X_{\geq 0} \wedge x \geq 1$. However, since $U(x) = -\infty$, $\mathsf{Extra}_{LU}(\mathbb{R}^X_{\geq 0} \wedge x \geq 1)$ gives back $\mathbb{R}^X_{\geq 0}$. Same for transitions that reset a clock. It follows that $ZG^{LU}(\mathcal{A}_\phi^Z)$ is isomorphic to $\mathcal{A}_\phi^Z$. This extends to Extra_{LU}^+ by Theorem 1. NP-hardness then comes from Lemma 5. NP-membership is proved in Lemma 7. □

In the next section, we provide an algorithm for the zenoness problem $\mathsf{ZP}^{\mathfrak{a}}$ and give conditions on abstraction $\mathfrak{a}$ for the solution to be polynomial.

5.2 Finding Zeno Paths

We say that a transition is *lifting* if it has a guard that implies $x \geq 1$ for some clock x. The idea is to find if there exists a run of an automaton $\mathcal{A}$ in which every clock x that is reset infinitely often is lifted only finitely many times, ensuring that the run is Zeno. This amounts to checking if there exists a cycle in $ZG(\mathcal{A})$ where every clock that is reset is not lifted. Observe that when $(q, Z) \overset{x \geq c}{\Longrightarrow} (q', Z')$ is a transition of $ZG(\mathcal{A})$, then Z' entails that $x \geq c$. Therefore, if a node (q, Z) is part of a cycle in the required form, then in particular, all the clocks that are greater than 1 in Z should not be reset in the cycle.

Based on the above intuition, our solution begins with computing the zone graph on-the-fly. At some node (q, Z) the algorithm non-deterministically guesses that this node is part of a cycle that yields a zeno run. This node transits to what we call the *slow mode*. In this mode, a reset of x in a transition is allowed from (q', Z') only if Z' is consistent with $x < 1$.

Before we define our construction formally, recall that we would be working with the abstract zone graph $ZG^{\mathfrak{a}}(\mathcal{A})$ and not $ZG(\mathcal{A})$. Therefore for our solution to work, the abstraction operator $\mathfrak{a}$ should remember the fact that a clock has a value greater than 1. For an automaton $\mathcal{A}$ over the set of clocks X, let $\mathrm{Lf}(\mathcal{A})$ denote the set of clocks that appear in a lifting transition of $\mathcal{A}$.

Definition 5 (Lift-safe abstractions). *An abstraction $\mathfrak{a}$ is called* lift-safe *if for every zone Z and for every clock $x \in \mathrm{Lf}(\mathcal{A})$, if $Z \vDash x \geq 1$ then $\mathfrak{a}(Z) \vDash x \geq 1$.*

We are now in a position to define our *slow zone graph* construction to decide if an automaton has a Zeno run.

Definition 6 (Slow zone graph). *Let $\mathcal{A}$ be a timed automaton over the set of clocks X. Let $\mathfrak{a}$ be a lift-safe abstraction. The* slow zone graph $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ *has nodes of the form (q, Z, l) where $l = \{\text{free}, \text{slow}\}$. The initial node is (q_0, Z_0, free) where (q_0, Z_0) is the initial node of $ZG^{\mathfrak{a}}(\mathcal{A})$. For every transition $(q, Z) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z')$ in $ZG^{\mathfrak{a}}(\mathcal{A})$ with $t = (q, g, R, q')$, we have the following transitions in $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$:*

- *a transition $(q, Z, \text{free}) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z', \text{free})$,*
- *a transition $(q, Z, \text{slow}) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z', \text{slow})$ if for all clocks $x \in R$, $Z \wedge g$ is consistent with $x < 1$,*

A new letter τ is introduced that adds transitions $(q, Z, \text{free}) \overset{\tau}{\Rightarrow}_{\mathfrak{a}} (q, Z, \text{slow})$.

A node of the form (q, Z, slow) is said to be a *slow* node. A path of $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ is said to be *slow* if it has a suffix consisting entirely of slow nodes. The τ-transitions take a node (q, Z) from the *free* mode to the slow mode. Note that the transitions of the slow mode are constrained further. The correctness follows from the fact that there is a cycle in $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ consisting entirely of slow nodes iff $\mathcal{A}$ has a Zeno run, proof of which can be found in [9].

From the definition of $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ it follows clearly that for each node (q, Z) of the zone graph there are two nodes in $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$: (q, Z, free) and (q, Z, slow). We thus get the following theorem.

Theorem 8. *Let $\mathfrak{a}$ be a lift-safe abstraction. The automaton $\mathcal{A}$ has a Zeno run iff $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ has an infinite slow path. The number of reachable nodes of $\mathcal{S}ZG^{\mathfrak{a}}(\mathcal{A})$ is atmost twice the number of reachable nodes in $ZG^{\mathfrak{a}}(\mathcal{A})$.*

We now turn our attention towards some of the abstractions existing in the literature. We observe that both Extra_M and Extra_M^+ are lift-safe and hence the Zenoness problem can be solved using the slow zone graph construction. However, in accordance to the NP-hardness of the problem for Extra_{LU}, we get that Extra_{LU} is not lift-safe.

Lemma 6. *The abstractions Extra_M and Extra_M^+ are lift-safe.*

Proof. Observe that for every clock that is lifted, the bound M is at least 1. It is now direct from the definitions that Extra_M and Extra_M^+ are lift-safe. □

Lemma 7. *The abstractions Extra_{LU} and Extra_{LU}^+ are not lift-safe. The Zenoness problem for Extra_{LU} and Extra_{LU}^+ is in* NP*.*

Proof. That Extra_{LU} and Extra_{LU}^+ are not lift-safe follows from the proof of Theorem 7. We show the NP-membership using a technique similar to the slow zone graph construction. Since Extra_{LU} is not lift-safe, the reachable zones in $ZG^{LU}(\mathcal{A})$ do not maintain the information about the clocks that have been lifted. Therefore, at some reachable zone (q, Z) we non-deterministically guess the set of clocks W that are allowed to be lifted in the future and go to a node (q, Z, W). From now on, there are transitions $(q, Z, W) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z', W)$ when:

- $(q, Z) \overset{t}{\Rightarrow}_{\mathfrak{a}} (q', Z')$ is a transition in $ZG^{LU}(\mathcal{A})$,
- if t contains a guard $x \geq c$ with $c \geq 1$, then $x \in W$,
- if t resets a clock x, then $x \notin W$

If a cycle is obtained that contains (q, Z, W), then the clocks that are reset and lifted in this cycle are disjoint and hence $\mathcal{A}$ has a Zeno run.

This shows that if $\mathcal{A}$ has a Zeno run we can non-deterministically choose a path of the above form and the length of this path is bounded by twice the number of zones in $ZG^{LU}(\mathcal{A})$ (which is our other input). This proves the NP-membership. □

5.3 Weakening the U Bounds

We saw in Lemma 7 that the extrapolation Extra_{LU} is not lift-safe. This is due to clocks x that are lifted but have $U(x) = -\infty$. These are exactly the clocks x with $L(x) \geq 1$ and $U(x) = -\infty$. We propose to weaken the U bounds so that the information about a clock being lifted is remembered in the abstracted zone.

Definition 7 (Weak U bounds). *Given the bounds $L(x)$ and $U(x)$ for each clock $x \in X$, the* weak upper bound *$\overline{U}(x)$ is given by: $\overline{U}(x) = 1$ if $L(x) \geq 1$ and $U(x) = -\infty$, and $\overline{U}(x) = U(x)$ otherwise.*

Let $\mathsf{Extra}_{L\overline{U}}$ denote the Extra_{LU} abstraction, but with $\overline{U}$ bound for each clock instead of U. This definition ensures that for all lifted clocks, that is, for all $x \in \mathrm{Lf}(\mathcal{A})$, if a zone entails that $x \geq 1$ then $\mathsf{Extra}_{L\overline{U}}(Z)$ also entails that $x \geq 1$. This is summarized by the following lemma, the proof of which follows by definitions.

Lemma 8. *For all zones Z, $\mathsf{Extra}_{L\overline{U}}$ is lift-safe.*

From Theorem 8, we get that the Zenoness problem is polynomial for $\mathsf{Extra}_{L\overline{U}}$. However, there is a price to pay. Weakening the U bounds leads to zone graphs exponentially bigger in some cases. For example, for the automaton $\mathcal{A}^Z_\phi$ that was used to prove the NP-completeness of the Zenoness problem with Extra_{LU}, note that the zone graph $ZG^{L\overline{U}}(\mathcal{A}^Z_\phi)$ obtained by applying $\mathsf{Extra}_{L\overline{U}}$ is exponentially bigger than $ZG^{LU}(\mathcal{A}^Z_\phi)$. This leads to a slow zone graph $\mathcal{S}ZG^{L\overline{U}}(\mathcal{A}^Z_\phi)$ with size polynomial in $ZG^{L\overline{U}}(\mathcal{A}^Z_\phi)$.

6 Conclusion

We have shown a surprising fact that the problem of deciding existence of Zeno or non-Zeno behaviours from abstract zone graphs depends heavily on the abstractions, to the extent that the problem changes from being polynomial to becoming NP-complete as the abstractions get coarser. We have proved NP-completeness for the coarse abstractions Extra_{LU} and Extra^+_{LU}. In contrast, the fundamental notions of reachability and Büchi emptiness over abstract zone graphs have a mere linear complexity, independent of the abstraction.

On the positive side, from our study on the conditions for an abstraction to give a polynomial solution, we see that a small modification of the LU-extrapolation works. We have defined two weaker abstractions: $\mathsf{Extra}_{\overline{L}U}$ for detecting non-Zeno runs and $\mathsf{Extra}_{L\overline{U}}$ for detecting Zeno runs. The weak bounds $\overline{L}$ and $\overline{U}$ can also be used with Extra^{+}_{LU} to achieve similar results. Despite leading to a polynomial solution for checking Zeno or non-Zeno behaviours from abstract zone graphs, these abstractions transfer the complexity to the input: they could lead to exponentially bigger abstract zone graphs themselves.

While working with abstract zone graphs, coarse abstractions (and hence small abstract zone graphs) are essential to handle big models of timed automata. These, as we have seen, work against the Zenoness questions. Our results therefore provide a theoretical motivation to look for cheaper substitutes to the notion of Zenoness.

Acknowledgements. We would like to thank Igor Walukiewicz for his insightful ideas on our problems and his helpful comments during the preparation of the manuscript. We thank Laurent Fribourg for pointing us to the problem of detecting Zeno runs.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Behrmann, G., Bouyer, P., Larsen, K.G., Pelanek, R.: Lower and upper bounds in zone-based abstractions of timed automata. Int. Journal on Software Tools for Technology Transfer 8(3), 204–215 (2006)
3. Behrmann, G., David, A., Larsen, K.G., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society, Los Alamitos (2006)
4. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
5. Bowman, H., Gómez, R.: How to stop time stopping. Formal Aspects of Computing 18(4), 459–493 (2006)
6. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
7. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) AVMFSS 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
8. Gómez, R., Bowman, H.: Efficient detection of zeno runs in timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 195–210. Springer, Heidelberg (2007)
9. Herbreteau, F., Srivathsan, B.: Coarse abstractions make zeno behaviours difficult to detect. CoRR abs/1106.1850 (2011); Extended version with proofs
10. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient Emptiness Check for Timed Büchi Automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)

11. Li, G.: Checking timed Büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009)
12. Tripakis, S.: Verifying progress in timed systems. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
13. Tripakis, S.: Checking timed Büchi emptiness on simulation graphs. ACM Trans. on Computational Logic 10(3) (2009)
14. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. Formal Methods in System Design 26(3), 267–292 (2005)
15. Wang, F.: Redlib for the formal verification of embedded systems. In: ISoLA, pp. 341–346. IEEE, Los Alamitos (2006)

Bisimulations Meet PCTL Equivalences for Probabilistic Automata*

Lei Song[1], Lijun Zhang[2], and Jens Chr. Godskesen[1]

[1] IT University of Copenhagen, Denmark
[2] DTU Informatics, Technical University of Denmark

Abstract. Probabilistic automata (PA) [20] have been successfully applied in the formal verification of concurrent and stochastic systems. Efficient model checking algorithms have been studied, where the most often used logics for expressing properties are based on PCTL [11] and its extension PCTL* [4]. Various behavioral equivalences are proposed for PAs, as a powerful tool for abstraction and compositional minimization for PAs. Unfortunately, the behavioral equivalences are well-known to be strictly stronger than the logical equivalences induced by PCTL or PCTL*. This paper introduces novel notions of strong bisimulation relations, which characterizes PCTL and PCTL* exactly. We also extend weak bisimulations characterizing PCTL and PCTL* without next operator, respectively. Thus, our paper bridges the gap between logical and behavioral equivalences in this setting.

1 Introduction

Probabilistic automata (PA) [20] have been successfully applied in the formal verification of concurrent and stochastic systems. Efficient model checking algorithms have been studied, where properties are mostly expressed in the logic PCTL, introduced in [11] for Markov chains, and later extended in [4] for Markov decision processes, where PCTL is also extended to PCTL*.

To combat the infamous state space problem in model checking, various behavioral equivalences, including strong and weak bisimulations, are proposed for PAs. Indeed, they turn out to be a powerful tool for abstraction for PAs, since bisimilar states implies that they satisfy exactly the same PCTL formulae. Thus, bisimilar states can be grouped together, allowing one to construct smaller quotient automata before analyzing the model. Moreover, the nice compositional theory for PAs is exploited for compositional minimization [5], namely minimizing the automata before composing the components together.

For Markov chains, i.e., PAs without nondeterministic choices, the logical equivalence implies also bisimilarity, as shown in [3]. Unfortunately, it does not hold in general, namely PCTL equivalence is strictly coarser than bisimulation – and their extension probabilistic bisimulation – for PAs. Even there is such a gap between behavior and logical equivalences, bisimulation based minimization is extensively studied in the literatures to leverage the state space explosion, for instance see [6,1,15].

* Supported by the VKR Centre of Excellence MT-LAB.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 108–123, 2011.

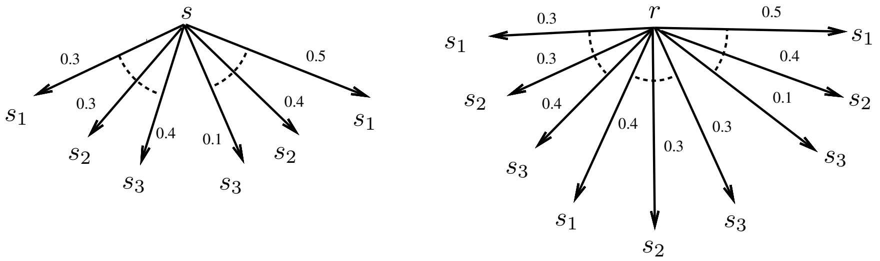


Fig. 1. Counter Example of Strong Probabilistic Bisimulation

The main reason for the gap can be illustrated by the following example. Consider the PAs in Fig.1 where assuming that s_1, s_2, s_3 are three absorbing states with different state properties. It is easy to see that s and r are PCTL equivalent: the additional middle transition out of r does not change the extreme probabilities. Existing bisimulations differentiate s and r, mainly because the middle transition out of r cannot be matched by any transition (or combined transition) of s. Bisimulation requires that the complete distribution of a transition must be matched, which is in this case too strong, as it differentiates states satisfying the same PCTL formulae.

In this paper we will bridge this gap. We introduce novel notions of behavioral equivalences which characterize (both soundly and completely) PCTL, PCTL* and their sublogics. Summarizing, our contributions are:

- A new bisimulation characterizing PCTL* soundly and completely. The bisimulation arises from a converging sequence of equivalence relations, each of which characterizes bounded PCTL*.
- Branching bisimulations which correspond to PCTL and bounded PCTL equivalences.
- We then extend our definitions to weak bisimulations, which characterize sublogics of PCTL and PCTL* with only unbounded path formulae.

Full proofs are given in [22].

Organization of the Paper. Section 2 introduces some notations. In Section 3 we recall definitions of probabilistic automata, bisimulation relations by Segala [19]. We also recall the logic PCTL* and its sublogics. Section 4 introduces the novel strong and strong branching bisimulations, and proves that they agree with PCTL* and PCTL equivalences, respectively. Section 5 extends them to weak (branching) bisimulations. In Section 6 we discuss related work, and Section 7 concludes the paper.

2 Preliminaries

Probability space. A (discrete) probability space is a tuple $\mathcal{P} = (\Omega, F, \eta)$ where Ω is a countable set, $F = 2^{\Omega}$ is the power set, and $\eta : F \to [0, 1]$ is a probability function which is countable additive. We skip F whenever convenient. Given probability spaces $\{\mathcal{P}_i = (\Omega_i, \eta_i)\}_{i \in I}$ and weights $w_i > 0$ for each i such that $\sum_{i \in I} w_i = 1$, the *convex combination* $\sum_{i \in I} w_i \mathcal{P}_i$ is defined as the probability space (Ω, η) such that $\Omega = \bigcup_{i \in I} \Omega_i$ and for each set $Y \subseteq \Omega$, $\eta(Y) = \sum_{i \in I} w_i \eta_i(Y \cap \Omega_i)$.

Distributions. We denote by $\mathit{Dist}(S)$ the set of discrete probability spaces over S. We shall use $s, r, t, \ldots$ and $\mu, \nu \ldots$ to range over S and $\mathit{Dist}(S)$, respectively. The support of μ is defined by $\mathit{supp}(\mu) := \{s \in S \mid \mu(s) > 0\}$. For an equivalence relation $\mathcal{R}$, we write $\mu \; \mathcal{R} \; \nu$ if it holds that $\mu(C) = \nu(C)$ for all equivalence classes $C \in S/\mathcal{R}$. A distribution μ is called *Dirac* if $|\mathit{supp}(\mu)| = 1$, and we let $\mathcal{D}_s$ denote the Dirac distribution with $\mathcal{D}_s(s) = 1$.

Upward Closure. Below we define the upward closure of a subset of states.

Definition 1. *For pre-order $\mathcal{R}$ over S and $C \subseteq S$, define $\overline{C_{\mathcal{R}}} = \{s' \mid s \; \mathcal{R} \; s' \wedge s \in C\}$. We say C is* $\mathcal{R}$-upward-closed *iff $C = \overline{C_{\mathcal{R}}}$.*

We use $\overline{s_{\mathcal{R}}}$ as the shorthand of $\overline{\{s\}_{\mathcal{R}}}$, and $\overline{\mathcal{R}} = \{\overline{C_{\mathcal{R}}} \mid C \subseteq S\}$ denotes the set of all $\mathcal{R}$-upward-closed sets.

3 Probabilistic Automaton, PCTL* and Bisimulations

Definition 2. *A* probabilistic automaton[1] *is a tuple $\mathcal{P} = (S, \rightarrow, \mathit{IS}, \mathit{AP}, L)$ where S is a finite set of states, $\rightarrow \; \subseteq S \times \mathit{Dist}(S)$ is a transition relation, $\mathit{IS} \subseteq S$ is a set of initial states, AP is a set of atomic propositions, and $L : S \rightarrow 2^{\mathit{AP}}$ is a labeling function.*

As usual we only consider image-finite PAs, i.e. $\{(r, \mu) \in \rightarrow \mid r = s\}$ is finite for each $s \in S$. A transition $(s, \mu) \in \rightarrow$ is denoted by $s \rightarrow \mu$. Moreover, we write $\mu \rightarrow \mu'$ iff for each $s \in \mathit{supp}(\mu)$ there exists $s \rightarrow \mu_s$ such that $\mu'(r) = \sum\limits_{s \in \mathit{supp}(\mu)} \mu(s) \cdot \mu_s(r)$.

A *path* is a finite or infinite sequence $\omega = s_0 s_1 s_2 \ldots$ of states. For each $i \geq 0$ there exists a distribution μ such that $s_i \rightarrow \mu$ and $\mu(s_{i+1}) > 0$. We use $\mathit{lstate}(\omega)$ and $l(\omega)$ to denote the last state of ω and the length of ω respectively if ω is finite. The sets Path is the set of all paths, and $\mathit{Path}(s_0)$ are those starting from s_0. Similarly, Path^* is the set of finite paths, and $\mathit{Path}^*(s_0)$ are those starting from s_0. Also we use $\omega[i]$ to denote the $(i+1)$-th state for $i \geq 0$, $\omega|^i$ to denote the fragment of ω ending at $\omega[i]$, and $\omega|_i$ to denote the fragment of ω starting from $\omega[i]$.

We introduce the definition of *scheduler* to resolve nondeterminism. A scheduler is a function $\sigma : \mathit{Path}^* \rightarrow \mathit{Dist}(\rightarrow)$ such that $\sigma(\omega)(s, \mu) > 0$ implies $s = \mathit{lstate}(\omega)$. A scheduler σ is *deterministic* if it returns only Dirac distributions, that is, the next step is chosen deterministically. We use

$$\mathit{Path}(s_0, \sigma) = \{\omega \in \mathit{Path}(s_0) \mid \forall i \geq 0. \exists \mu. \sigma(\omega|^i)(\omega[i], \mu) > 0 \wedge \mu(\omega[i+1]) > 0\}$$

[1] In this paper we omit the set of actions, since they do not appear in the logic PCTL we shall consider later. Note that the bisimulation we shall introduce later can be extended to PA with actions directly.

to denote the set of paths starting from s_0 respecting σ. Similarly, $Path^*(s_0, \sigma)$ only contains finite paths.

The *cone* of a finite path ω, denoted by C_ω, is the set of paths having ω as their prefix, i.e., $C_\omega = \{\omega' \mid \omega \leq \omega'\}$ where $\omega' \leq \omega$ iff ω' is a prefix of ω. Fixing a starting state s_0 and a scheduler σ, the measure $Prob_{\sigma,s_0}$ of a cone C_ω, where $\omega = s_0 s_1 \ldots s_k$, is defined inductively as follows: $Prob_{\sigma,s_0}(C_\omega)$ equals 1 if $k = 0$, and for $k > 0$,

$$Prob_{\sigma,s_0}(C_\omega) = Prob_{\sigma,s_0}(C_{\omega|^{k-1}}) \cdot \left(\sum_{(s_{k-1},\mu') \in \rightarrow} \sigma(\omega|^{k-1})(s_{k-1}, \mu') \cdot \mu'(s_k) \right)$$

Let $\mathcal{B}$ be the smallest algebra that contains all the cones and is closed under complement and countable unions. [2] $Prob_{\sigma,s_0}$ can be extended to a unique measure on $\mathcal{B}$.

Given a pre-order $\mathcal{R}$ over S, $\overline{\mathcal{R}}^i$ is the set of $\mathcal{R}$*-upward-closed paths* of length i composed of $\mathcal{R}$-upward-closed sets, and is equal to the *Cartesian* product of $\overline{\mathcal{R}}$ with itself i times. Let $\overline{\mathcal{R}}^* = \cup_{i \geq 1} \overline{\mathcal{R}}^i$ be the set of $\mathcal{R}$*-upward-closed paths* of arbitrary length. Define $l(\Omega) = n$ for $\Omega \in \overline{\mathcal{R}}^n$. For $\Omega = C_0 C_1 \ldots C_n \in \overline{\mathcal{R}}^*$, the $\mathcal{R}$*-upward-closed cone* C_Ω is defined as $C_\Omega = \{C_\omega \mid \omega \in \Omega\}$, where $\omega \in \Omega$ iff $\omega[i] \in C_i$ for $0 \leq i \leq n$.

For distributions μ_1 and μ_2, we define $\mu_1 \times \mu_2$ by $(\mu_1 \times \mu_2)((s_1, s_2)) = \mu_1(s_1) \times \mu_2(s_2)$. Following [2] we also define the interleaving of PAs:

Definition 3. *Let* $\mathcal{P}_i = (S_i, \rightarrow_i, IS_i, AP_i, L_i)$ *be two PAs with* $i = 1, 2$. *The* interleave composition $\mathcal{P}_1 \,||\, \mathcal{P}_2$ *is defined by:*

$$\mathcal{P}_1 \,||\, \mathcal{P}_2 = (S_1 \times S_2, \rightarrow, IS_1 \times IS_2, AP_1 \cup AP_2, L)$$

where $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ *and* $(s_1, s_2) \rightarrow \mu$ *iff either* $s_1 \rightarrow \mu_1$ *and* $\mu = \mu_1 \times \mathcal{D}_{s_2}$, *or* $s_2 \rightarrow \mu_2$ *and* $\mu = \mathcal{D}_{s_1} \times \mu_2$.

3.1 PCTL* and Its Sublogics

We introduce the syntax of PCTL [11] and PCTL* [4] which are probabilistic extensions of CTL and CTL* respectively. PCTL* over the set AP of atomic propositions are formed according to the following grammar:

$$\begin{aligned} \varphi &::= a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbb{P}_{\bowtie q}(\psi) \\ \psi &::= \varphi \mid \psi_1 \wedge \psi_2 \mid \neg\psi \mid \mathsf{X}\,\psi \mid \psi_1 \mathsf{U} \psi_2 \end{aligned}$$

where $a \in AP$, $\bowtie \in \{<, >, \leq, \geq\}$, $q \in [0, 1]$. We refer to φ and ψ as (PCTL*) state and path formulae, respectively.

The satisfaction relation $s \models \varphi$ for state formulae is defined in a standard manner for boolean formulae. For probabilistic operator, it is defined by $s \models \mathbb{P}_{\bowtie q}(\psi)$ iff

[2] By standard measure theory this algebra is a *σ-algebra* and all its elements are the measurable sets of paths.

$\forall\sigma.Prob_{\sigma,s}(\{\omega \in Path(s) \mid \omega \models \psi\}) \bowtie q$. The satisfaction relation $\omega \models \psi$ for path formulae is defined exactly the same as for LTL formulae, for example $\omega \models \mathsf{X}\,\psi$ iff $\omega|_1 \models \psi$, and $\omega \models \psi_1 \mathsf{U} \psi_2$ iff there exists $j \geq 0$ such that $\omega|_j \models \psi_2$ and $\omega|_k \models \psi_1$ for all $0 \leq k < j$.

Sublogics. The depth of path formula ψ of PCTL* free of U operator, denoted by $Depth(\psi)$, is defined by the maximum number of embedded X operators appearing in ψ, that is, $Depth(\varphi) = 0$, $Depth(\psi_1 \wedge \psi_2) = \max\{Depth(\psi_1), Depth(\psi_2)\}$, $Depth(\neg\psi) = Depth(\psi)$ and $Depth(\mathsf{X}\,\psi) = 1 + Depth(\psi)$. Then, we let PCTL^{*-} be the sublogic of PCTL* without the until ($\psi_1 \mathsf{U} \psi_2$) operator. Moreover, PCTL$_i^{*-}$ is a sublogic of PCTL^{*-} where for each ψ we have $Depth(\psi) \leq i$.

The sublogic PCTL is obtained by restricting the path formulae to:

$$\psi ::= \mathsf{X}\,\varphi \mid \varphi_1 \mathsf{U} \varphi_2 \mid \varphi_1 \mathsf{U}^{\leq n} \varphi_2$$

Note the bounded until formula does not appear in PCTL* as it can be encoded by nested next operator. PCTL$^-$ is defined in a similar way as for PCTL^{*-}. Moreover we let PCTL$_i^-$ be the sublogic of PCTL$^-$ where only bounded until operator $\varphi_1 \mathsf{U}^{\leq j} \varphi_2$ with $j \leq i$ is allowed.

Logical equivalence. For a logic $\mathcal{L}$, we say that s and r are $\mathcal{L}$-equivalent, denoted by $s \sim_{\mathcal{L}} r$, if they satisfy the same set of formulae of $\mathcal{L}$, that is $s \models \varphi$ iff $r \models \varphi$ for all formulae φ in $\mathcal{L}$. The logic $\mathcal{L}$ can be PCTL* or one of its sublogics.

3.2 Strong Probabilistic Bisimulation

In this section we introduce the definition of strong probabilistic bisimulation [20]. Let $\{s \to \mu_i\}_{i \in I}$ be a collection of transitions of $\mathcal{P}$, and let $\{p_i\}_{i \in I}$ be a collection of probabilities with $\sum_{i \in I} p_i = 1$. Then $(s, \sum_{i \in I} p_i \mu_i)$ is called a *combined transition* and is denoted by $s \to_{\mathrm{P}} \mu$ where $\mu = \sum_{i \in I} p_i \mu_i$.

Definition 4. *An equivalence relation $\mathcal{R} \subseteq S \times S$ is a strong probabilistic bisimulation iff $s \mathrel{\mathcal{R}} r$ implies that $L(s) = L(r)$ and for each $s \to \mu$, there exists a combined transition $r \to_P \mu'$ such that $\mu \mathrel{\mathcal{R}} \mu'$. We write $s \sim_P r$ whenever there is a strong probabilistic bisimulation $\mathcal{R}$ such that $s \mathrel{\mathcal{R}} r$.*

It was shown in [20] that $\sim_{\mathrm{P}}$ is preserved by $||$, that is, $s \sim_{\mathrm{P}} r$ implies $s \,||\, t \sim_{\mathrm{P}} r \,||\, t$ for any t. Also strong probabilistic bisimulation is sound for PCTL which means that if $s \sim_{\mathrm{P}} r$ then for any state formula φ of PCTL, $s \models \varphi$ iff $r \models \varphi$. But the other way around is not true, i.e. strong probabilistic bisimulation is not complete for PCTL, as illustrated by the following example.

Example 1. Consider again the two PAs in Fig. 1 and assume that $L(s) = L(r)$ and $L(s_1) \neq L(s_2) \neq L(s_3)$. In addition, s_1, s_2, and s_3 only have one transition to themselves with probability 1. The only difference between the left and right automata is that the right automaton has an extra step. It is not hard to see that $s \sim_{PCTL^*} r$. By Definition 4 $s \not\sim_{\mathrm{P}} r$ since the middle transition of r cannot be simulated by s even with combined transition. So we conclude that strong probabilistic bisimulation is not complete for PCTL* as well as for PCTL.

It should be noted that PCTL* distinguishes more states in a PA than PCTL. Refer to the following example.

Example 2. Suppose s and r are given by Fig. 1 where each of s_1, s_2, and s_3 is extended with a transition such that $s_1 \rightarrow \mu_1$ with $\mu_1(s_1) = 0.6$ and $\mu_1(s_4) = 0.4$, $s_2 \rightarrow \mu_2$ with $\mu_2(s_4) = 1$, and $s_3 \rightarrow \mu_3$ with $\mu_3(s_3) = 0.5$ and $\mu_3(s_4) = 0.5$. Here we assume that every state satisfies different atomic propositions except that $L(s) = L(r)$. Then it is not hard to see $s \sim_{PCTL} r$ while $s \nsim_{PCTL^*} r$. Consider the PCTL* formula $\varphi = \mathbb{P}_{\leq 0.38}(\mathsf{X}(L(s_1) \vee L(s_3)) \wedge \mathsf{X}\,\mathsf{X}(L(s_1) \vee L(s_3)))$: it holds $s \models \varphi$ but $r \not\models \varphi$. Note that φ is not a well-formed PCTL formula. Indeed, states s and r are PCTL-equivalent.

We have the following theorem:

Theorem 1. *1.* $\sim_{PCTL}$, $\sim_{PCTL^*}$, $\sim_{PCTL^-}$, $\sim_{PCTL_i^-}$, $\sim_{PCTL^{*-}}$, $\sim_{PCTL_i^{*-}}$, *and* $\sim_P$ *are equivalence relations for any* $i \geq 1$.

2. $\sim_P \;\subseteq\; \sim_{PCTL^*} \;\subseteq\; \sim_{PCTL}$.
3. $\sim_{PCTL^{*-}} \;\subseteq\; \sim_{PCTL^-}$.
4. $\sim_{PCTL_1^{*-}} \;=\; \sim_{PCTL_1^-}$.
5. $\sim_{PCTL_i^{*-}} \;\subseteq\; \sim_{PCTL_i^-}$ *for any* $i > 1$.
6. $\sim_{PCTL} \;\subseteq\; \sim_{PCTL^-} \;\subseteq\; \sim_{PCTL_{i+1}^-} \;\subseteq\; \sim_{PCTL_i^-}$ *for all* $i \geq 0$.
7. $\sim_{PCTL^*} \;\subseteq\; \sim_{PCTL^{*-}} \;\subseteq\; \sim_{PCTL_{i+1}^{*-}} \;\subseteq\; \sim_{PCTL_i^{*-}}$ *for all* $i \geq 0$.

4 A Novel Strong Bisimulation

This section presents our main contribution of the paper: we introduce a novel notion of strong bisimulation and strong branching bisimulation. We shall show that they agree with PCTL and PCTL* equivalences, respectively. As the preparation step we introduce the strong 1-depth bisimulation.

4.1 Strong 1-depth Bisimulation

Definition 5. *A pre-order* $\mathcal{R} \subseteq S \times S$ *is a strong* 1*-depth bisimulation if* $s \;\mathcal{R}\; r$ *implies that* $L(s) = L(r)$ *and for any* $\mathcal{R}$*-upward-closed set* C

1. *if* $s \rightarrow \mu$ *with* $\mu(C) > 0$*, there exists* $r \rightarrow \mu'$ *such that* $\mu'(C) \geq \mu(C)$*,*
2. *if* $r \rightarrow \mu$ *with* $\mu(C) > 0$*, there exists* $s \rightarrow \mu'$ *such that* $\mu'(C) \geq \mu(C)$*.*

We write $s \sim_1 r$ *whenever there is a strong* 1*-depth bisimulation* $\mathcal{R}$ *such that* $s \;\mathcal{R}\; r$*.*

The – though very simple – definition requires only one step matching of the distributions out of s and r. The essential difference to the standard definition is: the quantification of the upward-closed set comes before the transition $s \rightarrow \mu$. This is indeed the key of the new definition of bisimulations. The following theorem shows that $\sim_1$ agrees with $\sim_{PCTL_1^-}$ and $\sim_{PCTL_1^{*-}}$ which is also an equivalence relation:

Lemma 1. $\sim_{PCTL_1^-} \; = \; \sim_1 \; = \; \sim_{PCTL_1^{*-}}$.

Note that in Definition 5 we consider all the $\mathcal{R}$-upward-closed sets since it is not enough to only consider the $\mathcal{R}$-upward-closed sets in $\{\overline{s_\mathcal{R}} \mid s \in S\}$, refer to the following counterexample.

Counterexample 1. *Suppose that there are four absorbing states* s_1, s_2, s_3, *and* s_4 *which are assigned with different atomic propositions. Suppose we have two processes* s *and* r *such that* $L(s) = L(r)$, *and* $s \rightarrow \mu_1$, $s \rightarrow \mu_2$, $r \rightarrow \nu_1$, $r \rightarrow \nu_2$ *where* $\mu_1(s_1) = 0.5$, $\mu_1(s_2) = 0.5$, $\mu_2(s_3) = 0.5$, $\mu_2(s_4) = 0.5$, $\nu_1(s_1) = 0.5$, $\nu_1(s_3) = 0.5$, $\nu_2(s_2) = 0.5$, $\nu_2(s_4) = 0.5$. *If we only consider the* $\mathcal{R}$*-upward-closed sets in* $\{\overline{s_\mathcal{R}} \mid s \in S\}$ *where* $S = \{s, r, s_1, s_2, s_3, s_4\}$, *then we will conclude that* $s \sim_1 r$, *but* $r \models \varphi$ *while* $s \not\models \varphi$ *where* $\varphi = \mathbb{P}_{\geq 0.5}(\mathsf{X}(L(s_1) \vee L(s_2)))$.

It turns out that $\sim_1$ is preserved by $||$, implying that $\sim_{PCTL_1^-}$ and $\sim_{PCTL_1^{*-}}$ are preserved by $||$ as well.

Theorem 2. $s \sim_1 r$ *implies that* $s \,||\, t \sim_1 r \,||\, t$ *for any* t.

Remark 1. We note that for Kripke structure (PA with only Dirac distributions) $\sim_1$ agrees with the usual strong bisimulation by Milner [17].

4.2 Strong Branching Bisimulation

Now we extend the relation $\sim_1$ to strong i-step bisimulations. Then, the intersection of all of these relations gives us the new notion of strong branching bisimulation, which we show to be the same as $\sim_{PCTL}$. Recall Theorem 1 states that $\sim_{PCTL}$ is strictly coarser than $\sim_{PCTL^*}$, which we shall consider in the next section.

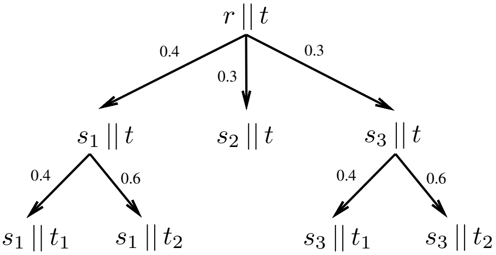


Fig. 2. $\sim_i^b$ is not compositional when $i > 1$

Following the way in [23] we define $Prob_{\sigma,s}(C, C', n, \omega)$ which denotes the probability from s to states in C' via states in C possibly in at most n steps under scheduler σ, where ω is used to keep track of the path and only deterministic schedulers are considered in the following. Formally, $Prob_{\sigma,s}(C, C', n, \omega)$ equals 1 if $s \in C'$, and else if $n > 0 \wedge (s \in C \setminus C')$, then

$$Prob_{\sigma,s}(C, C', n, \omega) = \sum_{r \in supp(\mu')} \mu'(r) \cdot Prob_{\sigma,r}(C, C', n-1, \omega r). \qquad (1)$$

where $\sigma(\omega)(s, \mu') = 1$, otherwise equals 0.

Strong i-depth branching bisimulation is a straightforward extension of strong 1-depth bisimulation, where instead of considering only one immediate step, we consider up to i steps. We let $\sim_1^b \; = \; \sim_1$ in the following.

Definition 6. *A pre-order $\mathcal{R} \subseteq S \times S$ is a strong i-depth branching bisimulation if $i > 1$ and $s \, \mathcal{R} \, r$ implies that $s \sim_{i-1}^{b} r$ and for any $\mathcal{R}$ upward-closed sets C, C',*

1. *if $Prob_{\sigma,s}(C, C', i, s) > 0$ for a scheduler σ, then there exists a scheduler σ' such that $Prob_{\sigma',r}(C, C', i, r) \geq Prob_{\sigma,s}(C, C', i, s)$,*
2. *if $Prob_{\sigma,r}(C, C', i, r) > 0$ for a scheduler σ, then there exists a scheduler σ' such that $Prob_{\sigma',s}(C, C', i, s) \geq Prob_{\sigma,r}(C, C', i, r)$.*

We write $s \sim_i^b r$ whenever there is a strong i-depth branching bisimulation $\mathcal{R}$ such that $s \, \mathcal{R} \, r$. The strong branching bisimulation $\sim^b$ is defined as $\sim^b = \bigcap_{i \geq 1} \sim_i^b$.

The following lemma shows that $\sim_i^b$ is an equivalence relation, and moreover, $\sim_i^b$ decreases until a fixed point is reached.

Lemma 2. *1. $\sim^b$ and $\sim_i^b$ are equivalence relations for any $i > 1$.*
2. $\sim_j^b \subseteq \sim_i^b$ provided that $1 \leq i \leq j$.
3. There exists $i \geq 1$ such that $\sim_j^b = \sim_k^b$ for any $j, k \geq i$.

It is not hard to show that $\sim_i^b$ characterizes PCTL$_i^-$. Moreover, we show that $\sim^b$ agrees with PCTL equivalence.

Theorem 3. $\sim_{PCTL_i^-} = \sim_i^b$ *for any $i \geq 1$, and moreover $\sim_{PCTL} = \sim^b$.*

Intuitively, since $\sim_{PCTL_i^-} = \sim_i^b$ decreases with i, for any PA $\sim_i^b$ will eventually converge to PCTL equivalence.

Recall $\sim_1^b$ is compositional by Theorem 2, which unfortunately is not the case for $\sim_i^b$ with $i > 1$. This is illustrated by the following example:

Counterexample 2. *$s \sim_i^b r$ does not imply $s \,||\, t \sim_i^b r \,||\, t$ for any t generally if $i > 1$.*

We have shown in Example 1 that $s \sim_{PCTL} r$. If we compose s and r with t where t only has a transition to μ such that $\mu(t_1) = 0.4$ and $\mu(t_2) = 0.6$, then it turns out that $s \,||\, t \not\sim_{PCTL} r \,||\, t$. Since there exists

$$\varphi = \mathbb{P}_{\leq 0.34}(true \mathsf{U}^{\leq 2}(L(s_1 \,||\, t_2) \vee L(s_3 \,||\, t_1)))$$

such that $s \,||\, t \models \varphi$ but $r \,||\, t \not\models \varphi$, as there exists a scheduler σ such that the probability of paths satisfying ψ in $Prob_{\sigma,r}$ equals 0.36 where $\psi = (true \mathsf{U}^{\leq 2}(L(s_1 \,||\, t_2) \vee L(s_3 \,||\, t_1)))$. Fig. 2 shows the execution of r guided by the scheduler σ, and we assume all the states in Fig. 2 have different atomic propositions except that $L(s \,||\, t) = L(r \,||\, t)$. It is similar for $\sim_{PCTL^*}$.

Note that φ is also a well-formed state formula of PCTL$_2^-$, so $\sim_{PCTL_i^-}$ as well as $\sim_i^b$ are not compositional if $i \geq 2$.

4.3 Strong Bisimulation

In this section we introduce a new notion of strong bisimulation and show that it characterizes $\sim_{PCTL^*}$. Given a pre-order $\mathcal{R}$, a $\mathcal{R}$-upward-closed cone C_Ω and a measure $Prob$, the value of $Prob(C_\Omega)$ can be computed by summing up the values of all

$Prob(C_\omega)$ with $\omega \in \Omega$. We let $\tilde{\Omega} \subseteq \overline{\mathcal{R}}^*$ be a set of $\mathcal{R}$-upward-closed paths, then $C_{\tilde{\Omega}}$ is the corresponding set of $\mathcal{R}$-upward-closed cones, that is, $C_{\tilde{\Omega}} = \cup_{\Omega \in \tilde{\Omega}} C_\Omega$. Define $l(\tilde{\Omega}) = Max\{l(\Omega) \mid \Omega \in \tilde{\Omega}\}$ as the maximum length of Ω in $\tilde{\Omega}$. To compute $Prob(C_{\tilde{\Omega}})$, we cannot sum up the value of each $Prob(C_\Omega)$ such that $\Omega \in \tilde{\Omega}$ as before since we may have a path ω such that $\omega \in \Omega_1$ and $\omega \in \Omega_2$ where $\Omega_1, \Omega_2 \in \tilde{\Omega}$, so we have to remove these duplicate paths and make sure each path is considered once and only once as follows where we abuse the notation and write $\omega \in \tilde{\Omega}$ iff $\exists \Omega.(\Omega \in \tilde{\Omega} \wedge \omega \in \Omega)$:

$$Prob(C_{\tilde{\Omega}}) = \sum_{\omega \in \tilde{\Omega} \wedge \nexists \omega' \in \tilde{\Omega}. \omega' \leq \omega} Prob(C_\omega) \tag{2}$$

Note Equation 2 can be extended to compute the probability of any set of cones in a given measure.

The definition of strong i-depth bisimulation is as follows:

Definition 7. *A pre-order $\mathcal{R} \subseteq S \times S$ is a strong i-depth bisimulation if $i > 1$ and $s \mathrel{\mathcal{R}} r$ implies that $s \sim_{i-1} r$ and for any $\tilde{\Omega} \subseteq \overline{\mathcal{R}}^*$ with $l(\tilde{\Omega}) = i$*

1. *if $Prob_{\sigma,s}(C_{\tilde{\Omega}}) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',r}(C_{\tilde{\Omega}}) \geq Prob_{\sigma,s}(C_{\tilde{\Omega}})$,*
2. *if $Prob_{\sigma,r}(C_{\tilde{\Omega}}) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',s}(C_{\tilde{\Omega}}) \geq Prob_{\sigma,r}(C_{\tilde{\Omega}})$.*

We write $s \sim_i r$ whenever there is a i-depth strong bisimulation $\mathcal{R}$ such that $s \mathrel{\mathcal{R}} r$. The strong bisimulation $\sim$ is defined as $\sim = \cap_{i \geq 1} \sim_i$.

Similar to $\sim_i^b$, the relation $\sim_i$ forms a chain of equivalence relations where the strictness of $\sim_i$ increases as i increases, and $\sim_i$ will converge finally in a PA.

Lemma 3. *1. $\sim_i$ is an equivalence relation for any $i > 1$.*
2. $\sim_j \subseteq \sim_i$ provided that $1 \leq i \leq j$.
3. There exists $i \geq 1$ such that $\sim_j = \sim_k$ for any $j, k \geq i$.

Below we show that $\sim_i$ characterizes $\sim_{PCTL_i^{*-}}$ for all $i \geq 1$, and $\sim$ agrees with PCTL* equivalence:

Theorem 4. *$\sim_{PCTL_i^{*-}} = \sim_i$ for any $i \geq 1$, and moreover, $\sim_{PCTL^*} = \sim$.*

Recall by Lemma 3, there exists $i > 0$ such that $\sim_{PCTL^*} = \sim_i$. For the same reason as strong i-depth branching bisimulation, $\sim_i$ is not preserved by $||$ when $i > 1$.

Counterexample 3. *$s \sim_i r$ does not imply $s \,||\, t \sim_i r \,||\, t$ for any t generally if $i > 1$. This can be shown by using the same arguments as in Counterexample 2.*

4.4 Taxonomy for Strong Bisimulations

Fig. 3 summaries the relationship among all these bisimulations and logical equivalences. The arrow → denotes $\subseteq$ and $\nrightarrow$ denotes $\nsubseteq$. We also abbreviate $\sim_{PCTL}$ as PCTL,

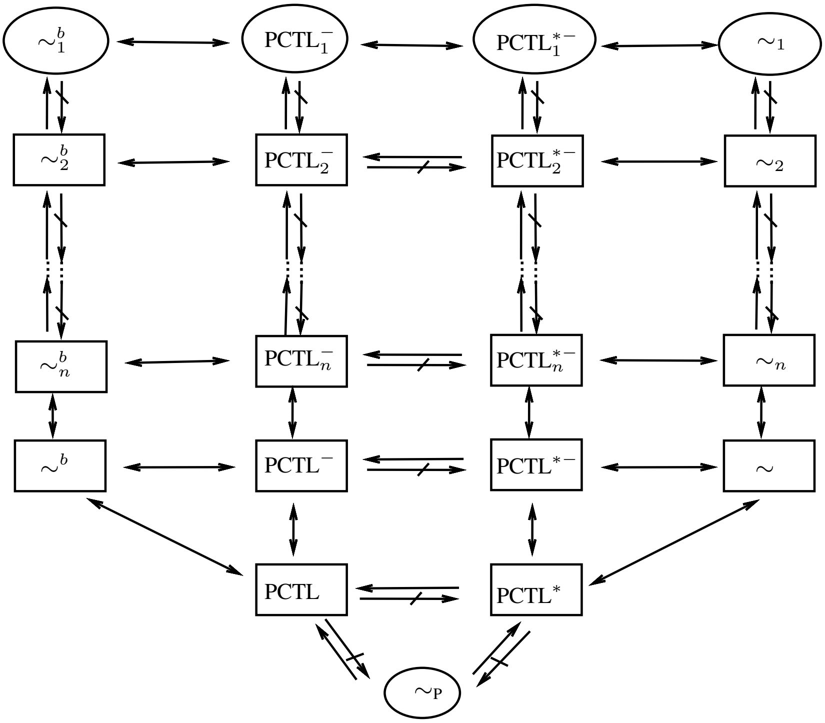


Fig. 3. Relationship of Different Equivalences in Strong Scenario

and it is similar for other logical equivalences. Congruent relations with respect to $||$ operator are shown in circles, and non-congruent in boxes. Segala has considered another strong bisimulation in [20], which can be defined by replacing the $r \rightarrow_\text{P} \mu'$ with $r \rightarrow \mu'$ and thus is strictly stronger than $\sim_\text{P}$. It is also worth mentioning that all the bisimulations shown in Fig.3 coincide with the strong bisimulation defined in [3] in the DTMC setting which can be seen as a special case of PA (i.e., deterministic probabilistic automata).

5 Weak Bisimulations

As in [3] we use $\text{PCTL}_{\setminus \mathsf{X}}$ to denote the subset of PCTL without next operator $\mathsf{X}\,\varphi$ and bounded until $\varphi_1 \mathsf{U}^{\leq n} \varphi_2$. Similarly, $\text{PCTL}^*_{\setminus \mathsf{X}}$ is used to denote the subset of PCTL^* without next operator $\mathsf{X}\,\psi$. In this section we shall introduce weak bisimulations and study the relation to $\sim_{PCTL_{\setminus \mathsf{X}}}$ and $\sim_{PCTL^*_{\setminus \mathsf{X}}}$, respectively. Before this we should point out that $\sim_{PCTL^*_{\setminus \mathsf{X}}}$ implies $\sim_{PCTL_{\setminus \mathsf{X}}}$ but the other direction does not hold. Refer to the following example.

Example 3. Suppose s and r are given by Fig. 1 where each of s_1 and s_3 is attached with one transition respectively, that is, $s_1 \rightarrow \mu_1$ such that $\mu_1(s_4) = 0.4$ and $\mu_1(s_5) = 0.6$,

$s_3 \rightarrow \mu_3$ such that $\mu_3(s_4) = 0.4$ and $\mu_3(s_5) = 0.6$. In addition, s_2, s_4 and s_5 only have a transition with probability 1 to themselves, and all these states are assumed to have different atomic propositions. Then $s \sim_{PCTL_{\setminus \mathsf{X}}} r$ but $s \nsim_{PCTL^*_{\setminus \mathsf{X}}} r$, since we have a path formula $\psi = ((L(s) \vee L(s_1))\mathsf{U}L(s_5)) \vee ((L(s) \vee L(s_3))\mathsf{U}L(s_4))$ such that $s \models \mathbb{P}_{\leq 0.34}\psi$ but $r \not\models \mathbb{P}_{\leq 0.34}\psi$, since there exists a scheduler σ where the probability of path formulae satisfying ψ in $Prob_{\sigma,r}$ is equal to $Prob_{\sigma,r}(ss_1s_5) + Prob_{\sigma,r}(ss_3s_4) = 0.36$. Note ψ is not a well-formed path formula of PCTL$_{\setminus \mathsf{X}}$.

5.1 Branching Probabilistic Bisimulation by Segala

Before introducing our weak bisimulations, we give the classical definition of branching probabilistic bisimulation proposed in [20]. Given an equivalence relation $\mathcal{R}$, s can evolve into μ by a *branching transition*, written as $s \Rightarrow^{\mathcal{R}} \mu$, iff i) $\mu = \mathcal{D}_s$, or ii) $s \rightarrow \mu'$ and

$$\mu = \sum_{r \in (supp(\mu') \cap [s]) \wedge r \Rightarrow^{\mathcal{R}} \mu_r} \mu'(r) \cdot \mu_r + \sum_{r \in supp(\mu') \setminus [s]} \mu'(r) \cdot \mathcal{D}_r$$

where $[s]$ denotes the equivalence class containing s. Stated differently, $s \Rightarrow^{\mathcal{R}} \mu$ means that s can evolve into μ only via states in $[s]$. Accordingly, *branching combined transition* $s \Rightarrow^{\mathcal{R}}_{\mathsf{P}} \mu$ can be defined based on the branching transition, i.e. $s \Rightarrow^{\mathcal{R}}_{\mathsf{P}} \mu$ iff there exists a collection of branching transitions $\{s \Rightarrow^{\mathcal{R}} \mu_i\}_{i \in I}$, and a collection of probabilities $\{p_i\}_{i \in I}$ with $\sum_{i \in I} p_i = 1$ such that $\mu = \sum_{i \in I} p_i \mu_i$.

We give the definition branching probabilistic bisimulation as follows:

Definition 8. *An equivalence relation $\mathcal{R} \subseteq S \times S$ is a branching probabilistic bisimulation iff $s \mathrel{\mathcal{R}} r$ implies that $L(s) = L(r)$ and for each $s \rightarrow \mu$, there exists $r \Rightarrow^{\mathcal{R}}_{P} \mu'$ such that $\mu \mathrel{\mathcal{R}} \mu'$.*

We write $s \simeq_P r$ whenever there is a branching probabilistic bisimulation $\mathcal{R}$ such that $s \mathrel{\mathcal{R}} r$.

The following properties concerning branching probabilistic bisimulation are taken from [20]:

Lemma 4 ([20]).

1. $\simeq_P \ \subseteq \ \sim_{PCTL^*_{\setminus \mathsf{X}}} \ \subseteq \ \sim_{PCTL_{\setminus \mathsf{X}}}$.
2. *$\simeq_P$ is preserved by $||$.*

5.2 A Novel Weak Branching Bisimulation

Similar to the definition of bounded reachability $Prob_{\sigma,s}(C, C', n, \omega)$, we define the function $Prob_{\sigma,s}(C, C', \omega)$ which denotes the probability from s to states in C' possibly via states in C. Again ω is used to keep track of the path which has been visited. Formally, $Prob_{\sigma,s}(C, C', \omega)$ is equal to 1 if $s \in C'$, $Prob_{\sigma,s}(C, C', \omega)$ is equal to 0 if $s \notin C$, otherwise when $\sigma(\omega)(s, \mu') = 1$,

$$Prob_{\sigma,s}(C, C', \omega) = \sum_{r \in supp(\mu')} \mu'(r) \cdot Prob_{\sigma,r}(C, C', \omega r). \quad (3)$$

The definition of weak branching bisimulation is as follows:

Definition 9. *A pre-order $\mathcal{R} \subseteq S \times S$ is a weak branching bisimulation if $s \ \mathcal{R} \ r$ implies that $L(s) = L(r)$ and for any $\mathcal{R}$-upward closed sets C, C'*

1. *if $Prob_{\sigma,s}(C, C', s) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',r}(C, C', r) \geq Prob_{\sigma,s}(C, C', s)$,*
2. *if $Prob_{\sigma,r}(C, C', r) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',s}(C, C', s) \geq Prob_{\sigma,r}(C, C', r)$.*

We write $s \ \approx^b \ r$ whenever there is a weak branching bisimulation $\mathcal{R}$ such that $s \ \mathcal{R} \ r$.

The following theorem shows that $\approx^b$ is an equivalence relation. Also different from the strong cases where we use a series of equivalence relations to either characterize or approximate $\sim_{PCTL}$ and $\sim_{PCTL^*}$, in the weak scenario we show that $\approx^b$ itself is enough to characterize $\sim_{PCTL_{\setminus \mathsf{X}}}$. Intuitively because in $\sim_{PCTL_{\setminus \mathsf{X}}}$ only unbounded until operator is allowed in path formula which means we abstract from the number of steps to reach certain states.

Theorem 5. *1. $\approx^b$ is an equivalence relation.*
2. $\approx^b \ = \ \sim_{PCTL_{\setminus \mathsf{X}}}$.

As in the strong scenario, $\approx^b$ suffers from the same problem as $\sim_i^b$ and $\sim_i$ with $i > 1$, that is, it is not preserved by $||$.

Counterexample 4. *$s \ \approx^b \ r$ does not always imply $s \,||\, t \ \approx^b \ r \,||\, t$ for any t. This can be shown in a similar way as Counterexample 2 since the result will still hold even if we replace the bounded until formula with unbounded until formula in Counterexample 2.*

5.3 Weak Bisimulation

In order to define weak bisimulation we consider stuttering paths. Let Ω be a finite $\mathcal{R}$-upward-closed path, then

$$C_{\Omega_{st}} = \begin{cases} C_{\Omega} & l(\Omega) = 1 \\ \bigcup\limits_{\forall 0 \leq i < n. \forall k_i \geq 0} C_{(\Omega[0])^{k_0} \ldots (\Omega[n-2])^{k_{n-2}} \Omega[n-1]} & l(\Omega) = n \geq 2 \end{cases} \tag{4}$$

is the set of $\mathcal{R}$-upward-closed paths which contains all stuttering paths, where $\Omega[i]$ denotes the $(i+1)$-th element in Ω such that $0 \leq i < l(\Omega)$. Accordingly, $C_{\tilde{\Omega}_{st}} = \bigcup\limits_{\Omega \in \tilde{\Omega}} C_{\Omega_{st}}$ contains all the stuttering paths of each $\Omega \in \tilde{\Omega}$. Given a measure $Prob$, $Prob(\tilde{\Omega}_{st})$ can be computed by Equation (2).

Now we are ready to give the definition of weak bisimulation as follows:

Definition 10. *A pre-order $\mathcal{R} \subseteq S \times S$ is a weak bisimulation if $s \ \mathcal{R} \ r$ implies that $L(s) = L(r)$ and for any $\tilde{\Omega} \subseteq \overline{\mathcal{R}}^*$*

1. *if $Prob_{\sigma,s}(C_{\tilde{\Omega}_{st}}) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',r}(C_{\tilde{\Omega}_{st}}) \geq Prob_{\sigma,s}(C_{\tilde{\Omega}_{st}})$,*
2. *if $Prob_{\sigma,r}(C_{\tilde{\Omega}_{st}}) > 0$ for a scheduler σ, there exists σ' such that $Prob_{\sigma',s}(C_{\tilde{\Omega}_{st}}) \geq Prob_{\sigma,r}(C_{\tilde{\Omega}_{st}})$.*

We write $s \approx r$ whenever there is a weak bisimulation $\mathcal{R}$ such that $s \,\mathcal{R}\, r$.

The following theorem shows that $\approx$ is an equivalence relation. For the same reason as in Theorem 5, $\approx$ is enough to characterize $\sim_{PCTL^*_{\setminus \mathsf{X}}}$ which gives us the following theorem.

Theorem 6. *1. $\approx$ is an equivalence relation.*
2. $\approx \;=\; \sim_{PCTL^*_{\setminus \mathsf{X}}}$.

Not surprisingly $\approx$ is not preserved by $||$.

Counterexample 5. *$s \approx r$ does not always imply $s \,||\, t \approx r \,||\, t$ for any t. This can be shown by using the same arguments as in Counterexample 4.*

5.4 Taxonomy for Weak Bisimulations

As in the strong cases we summarize the relation of the equivalences in the weak scenario in Fig. 4 where all the denotations have the same meaning as Fig. 3. Compared to Fig. 3, Fig. 4 is much simpler because the step-indexed bisimulations are absent. As in strong cases, here we do not consider the standard definition of branching bisimulation which is a strict subset of $\simeq_{\mathrm{P}}$ and can be defined by replacing $\Rightarrow_{\mathrm{P}}^{\mathcal{R}}$ with $\Rightarrow^{\mathcal{R}}$ in Definition 8. Again not surprisingly all the relations shown in Fig. 4 coincide with the weak bisimulation defined in [3] in DTMC setting.

6 Related Work

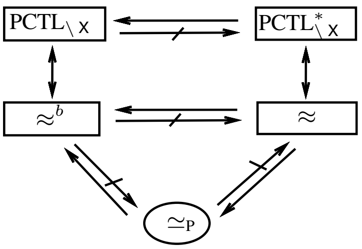

$\mathrm{PCTL}_{\setminus \mathsf{X}}$
$\mathrm{PCTL}^*_{\setminus \mathsf{X}}$
$\approx^b$
$\approx$
$\simeq_{\mathrm{P}}$

Fig. 4. Relationship of Different Equivalences in Weak Scenario

For Markov chains, i.e., deterministic probabilistic automata, the logic PCTL characterizes bisimulations, and PCTL without X operator characterizes weak bisimulations [10,3]. As pointed out in [20], probabilistic bisimulation is sound, but not complete for PCTL for PAs. In the literatures, various extensions of the Hennessy & Milner [12] are considered for characterizing bisimulations. Larsen and Skou [16] considered such an extension of Hennessy-Milner logic, which characterizes bisimulation for *alternating automaton* [16], or labeled Markov processes [8] (PAs but with continuous state space). For probabilistic automata, Jonsson *et al.* [14] considered a two-sorted logic in the Hennessy-Milner style to characterize strong bisimulations. In [13], the results are extended for characterizing also simulations.

Weak bisimulation was first defined in the context of PAs by Segala [20], and then formulated for alternating models by Philippou *et al.* [18]. The seemingly very related work is by Desharnais et al. [8], where it is shown that PCTL* is sound and complete with respect to weak bisimulation for alternating automata. The key difference is the

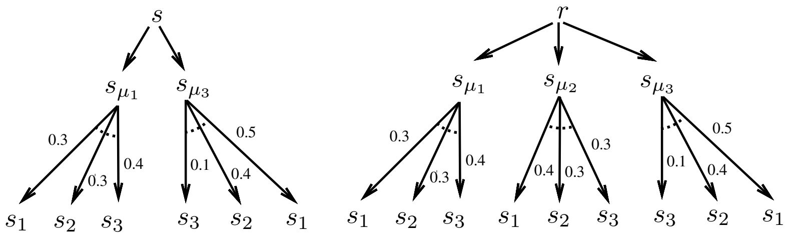


Fig. 5. Alternating Automata

model they have considered is not the same as probabilistic automata considered in this paper. Briefly, in alternating automata, states are either nondeterministic like in transition systems, or stochastic like in discrete-time Markov chains. As discussed in [21], a probabilistic automaton can be transformed to an alternating automaton by replacing each transition $s \to \mu$ by two consecutive transitions $s \to s'$ and $s' \to \mu$ where s' is the new inserted state. Surprisingly, for alternating automata, Desharnais et al. have shown that weak bisimulation – defined in the standard manner – characterizes PCTL* formulae. The following example illustrates why it works in that setting, but fails in probabilistic automata.

Example 4. Refer to Fig. 1, we need to add three additional states s_{μ_1}, s_{μ_2}, and s_{μ_3} in order to transform s and r to alternating automata. The resulting automata are shown in Fig. 5. Suppose that s_1, s_2, and s_3 are three absorbing states with different atomic propositions, so they are not (weak) bisimilar with each other, as result s_{μ_1}, s_{μ_2} and s_{μ_3} are not (weak) bisimilar with each other either since they can evolve into s_1, s_2, and s_3 with different probabilities. Therefore s and r are not (weak) bisimilar. Let $\varphi = \mathbb{P}_{\geq 0.4}(\mathsf{X}\, L(s_1)) \wedge \mathbb{P}_{\geq 0.3}(\mathsf{X}\, L(s_2)) \wedge \mathbb{P}_{\geq 0.3}(\mathsf{X}\, L(s_3))$, it is not hard to see that $s_{\mu_2} \models \varphi$ but $s_{\mu_1}, s_{\mu_3} \not\models \varphi$, so $s \models \mathbb{P}_{\leq 0}(\mathsf{X}\, \varphi)$ while $r \not\models \mathbb{P}_{\leq 0}(\mathsf{X}\, \varphi)$. If working with the probabilistic automata, s_{μ_1}, s_{μ_2}, and s_{μ_3} will not be considered as states, so we cannot use the above arguments for alternating automata anymore.

Finally, we want to mention some similarities of $\sim_1$ and notion of metrics studied in [9,7]. In the definition of $\sim_1$, we choose first the upward-closed set C before the successor distribution to be matched, which is the key for achieving our new notion of bisimulations. This is used in a similar way in defining metrics in [9,7].

7 Conclusion and Future Work

In this paper we have introduced novel notion of bisimulations for probabilistic automata. They are coarser than the existing bisimulations, and most importantly, we show that they agree with logical equivalences induced by PCTL* and its sublogics. Even in this paper we have not considered actions, it is worth noting that actions can be easily added, and all the results relating (weak) bisimulations hold straightforwardly. On the other side, they are then strictly finer than the logical equivalences, because of the presence of these actions.

As future work, we plan to study decision algorithms for our new (strong and weak) bisimulations, and also extend the work to countable state space.

References

1. Baier, C., Engelen, B., Majster-Cederbaum, M.E.: Deciding bisimilarity and similarity for probabilistic processes. J. Comput. Syst. Sci. 60(1), 187–231 (2000)
2. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press, Cambridge (2008)
3. Baier, C., Katoen, J.-P., Hermanns, H., Wolf, V.: Comparative branching-time semantics for markov chains. Inf. Comput. 200(2), 149–214 (2005)
4. Bianco, A., De Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
5. Boudali, H., Crouzen, P., Stoelinga, M.: A rigorous, compositional, and extensible framework for dynamic fault tree analysis. IEEE Transactions on Dependable and Secure Computing 99(1) (2009)
6. Cattani, S., Segala, R.: Decision algorithms for probabilistic bisimulation. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 371–385. Springer, Heidelberg (2002)
7. de Alfaro, L., Majumdar, R., Raman, V., Stoelinga, M.: Game relations and metrics. In: LICS, pp. 99–108 (2007)
8. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Weak bisimulation is sound and complete for pctl*. Inf. Comput. 208(2), 203–219 (2010)
9. Desharnais, J., Tracol, M., Zhioua, A.: Computing distances between probabilistic automata. In: QAPL (to appear, 2011)
10. Hansson, H., Jonsson, B.: A Calculus for Communicating Systems with Time and Probabitilies. In: IEEE Real-Time Systems Symposium, pp. 278–287 (1990)
11. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
12. Hennessy, M., Milner, R.: Algebraic Laws for Nondeterminism and Concurrency. J. ACM 32(1), 137–161 (1985)
13. Hermanns, H., Parma, A., Segala, R., Wachter, B., Zhang, L.: Probabilistic logical characterization. Inf. Comput. 209(2), 154–172 (2011)
14. Jonsson, B., Larsen, K., Wang, Y.: Probabilistic extensions of process algebras. In: Bergstra, J., Ponse, A., Smolka, S. (eds.) Handbook of Process Algebra, pp. 685–710. Elsevier, Amsterdam (2001)
15. Katoen, J.-P., Kemna, T., Zapreev, I.S., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007)
16. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94(1), 1–28 (1991)
17. Milner, R.: Communication and concurrency. Prentice Hall International Series in Computer Science (1989)
18. Philippou, A., Lee, I., Sokolsky, O.: Weak Bisimulation for Probabilistic Systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 334–349. Springer, Heidelberg (2000)
19. Segala, R.: Modeling and Verification of Randomized Distributed Realtime Systems. PhD thesis, MIT (1995)

20. Segala, R., Lynch, N.A.: Probabilistic Simulations for Probabilistic Processes. Nord. J. Comput. 2(2), 250–273 (1995)
21. Segala, R., Turrini, A.: Comparative analysis of bisimulation relations on alternating and non-alternating probabilistic models. In: QEST, pp. 44–53 (2005)
22. Song, L., Zhang, L., Godskesen, J.C.: Bisimulations Meet PCTL Equivalences for Probabilistic Automata (June 2011), http://arxiv.org/abs/1106.2181
23. van Glabbeek, R., Weijland, W.: Branching time and abstraction in bisimulation semantics. Journal of the ACM (JACM) 43(3), 555–600 (1996)

A Spectrum of Behavioral Relations over LTSs on Probability Distributions

Silvia Crafa and Francesco Ranzato

University of Padova, Italy

Abstract. Probabilistic nondeterministic processes are commonly modeled as probabilistic LTSs (PLTSs, a.k.a. probabilistic automata). A number of logical characterizations of the main behavioral relations on PLTSs have been studied. In particular, Parma and Segala [2007] define a probabilistic Hennessy-Milner logic interpreted over distributions, whose logical equivalence/preorder when restricted to Dirac distributions coincide with standard bisimulation/simulation between the states of a PLTS. This result is here extended by studying the full logical equivalence/preorder between distributions in terms of a notion of bisimulation/simulation defined on a LTS of probability distributions (DLTS). We show that the standard spectrum of behavioral relations on nonprobabilistic LTSs as well as its logical characterization in terms of Hennessy-Milner logic scales to the probabilistic setting when considering DLTSs.

1 Introduction

Formal methods for concurrent and distributed system specification and verification have been extended to encompass randomized phenomena exhibited by the behavior of probabilistic systems. In a standard nonprobabilistic setting, systems are commonly modeled as labeled transition systems (LTSs) and model checking techniques are based on two major tools: temporal logics and behavioral relations. Logics are used to specify the properties that systems have to satisfy, while behavioral equivalence and preorder relations are used as appropriate abstractions that reduce the state space. Precise relationships have been established between these two approaches: van Glabbeek [8] shows how a wide spectrum of observational equivalences for concurrent processes is logically characterized in terms of Hennessy-Milner-like modal logics (HML).

A number of probabilistic behavioral relations and probabilistic temporal logics have been proposed (see e.g. [4,9,10,11,12,13,15]). Probabilistic LTSs (PLTSs, a.k.a. probabilistic automata) are a prominent model for formalizing probabilistic systems since they allow to model both probabilistic and nondeterministic behaviors. In PLTSs, a state s evolves through a labeled transition to a state distribution d that defines the probabilities of reaching the possible successor states of s. Accordingly, the standard probabilistic extension [15] of the simulation relation requires that if a state s progresses to a distribution d, then a simulating state s' needs to mimic such a transition by moving to a distribution d' that is related to d through a so-called weight function. This definition is a conservative extension of the simulation relation on LTSs since a LTS can be viewed as a particular PLTS where the target of transitions are Dirac distributions, i.e., distributions δ_s such that $\delta_s(s) = 1$ and $\delta_s(t) = 0$ for any $t \neq s$.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 124–139, 2011.

A number of modal logics have been proposed in order to provide a logical characterization of probabilistic simulation and bisimulation. Larsen and Skou [12]'s logic as well as Hansson and Jonsson [10]'s PCTL logic are interpreted over states of probabilistic systems, such as reactive models and discrete time Markov chains, that do not express nondeterminism. On the other hand, Parma and Segala [13] show that richer probabilistic models that encode pure nondeterminism (besides probabilistic choice), such as PLTSs, call for a richer logic. They propose a probabilistic extension of HML whose formulae are interpreted over distributions rather than states, and they show that two states s and t are similar (the same holds for bisimilarity) if and only if their corresponding Dirac distributions δ_s and δ_t satisfy the same set of formulae. However, nothing is stated about logically equivalent distributions that are not Dirac distributions.

In this paper we study the full logical equivalence between (possibly non-Dirac) distributions that is induced by Parma and Segala [13]'s logic. We show that this logic actually characterizes a novel and natural notion of simulation (bisimulation) between distributions of a PLTS, so that the standard state simulation (bisimulation) on PLTSs can be indeed retrieved by a suitable restriction to Dirac distributions. Furthermore, the transition relation of a PLTS is lifted to a transition relation between distributions that gives rise to a corresponding LTS on distributions (called DLTS). This allows us to lift behavioral relations on PLTSs to corresponding behavioral relations on DLTSs. Such a move from PLTSs to DLTSs yields a number of byproducts:

- Parma and Segala [13]'s logic turns out to be equivalent to a logic $\mathcal{L}$ whose diamond operator is interpreted on the DLTS in accordance with its standard interpretation on LTSs. Hence, this logic best suits as probabilistic extension of HML.
- This logic $\mathcal{L}$ characterizes a (bi)simulation relation between distributions which is equivalent to that characterized by Parma and Segala's logic, but that naturally admits a game-theoretic characterization.
- A spectrum of behavioral relations can be defined on DLTSs along the lines of the standard approach on LTSs [9]. These preorders and equivalences between distributions can be then projected back to states, thus providing a spectrum of (probabilistic) preorders and equivalences between states of PLTSs.
- Shifting the problem from PLTSs to DLTSs opens the way to the reuse of efficient model checking techniques available for LTSs.

This approach is studied on a number of well known probabilistic relations appearing in literature, namely simulation, probabilistic simulation, failure simulation, and their corresponding bisimulations. A discussion about related approaches is the subject of the final section, that also hints at future work.

2 Probabilistic Simulation and Bisimulation

Given a set X and a relation $R \subseteq X \times X$, we write xRy for $(x, y) \in R$; if $x \in X$ and $Y \subseteq X$ then $R(x) \triangleq \{y \in X \mid xRy\}$ and $R(Y) \triangleq \cup_{x \in Y} R(x)$.

$\mathrm{Distr}(X)$ denotes the set of (stochastic) distributions on a set X, i.e., the set of functions $d : X \to [0, 1]$ such that $\sum_{x \in X} d(x) = 1$. The support of a distribution d is defined by $\mathrm{supp}(d) \triangleq \{x \in X \mid d(x) > 0\}$; moreover, if $Y \subseteq X$, then $d(Y) \triangleq$

$\sum_{y \in Y} d(y)$. The Dirac distribution on $x \in X$, denoted by δ_x, is the distribution that assigns probability 1 to x (and 0 otherwise).

A probabilistic LTS (PLTS) is a tuple $\mathcal{M} = \langle \Sigma, Act, \rightarrow \rangle$ where Σ is a (denumerable) set of states, Act is a (denumerable) set of actions, and $\rightarrow \subseteq \Sigma \times Act \times \mathrm{Distr}(\Sigma)$ is a transition relation, where $(s, a, d) \in \rightarrow$ is tipically denoted by $s \xrightarrow{a} d$. For any $a \in Act$, the predecessor operator $\mathrm{pre}_a : \wp(\mathrm{Distr}(\Sigma)) \rightarrow \wp(\Sigma)$ is defined by $\mathrm{pre}_a(D) \triangleq \{s \in \Sigma \mid \exists d \in D.\, s \xrightarrow{a} d\}$.

The definitions of probabilistic behavioral relations often rely on so-called weight functions [14], that are used to lift a relation between states to a relation between distributions. We do not recall here the definition of weight functions, as we will use the following equivalent characterizations (see [6,11,17]).

Definition 2.1 (Lifting). Let $R \subseteq X \times X$ be any relation. Then, the *lifting* of R to distributions is the relation $\sqsubseteq_R \subseteq \mathrm{Distr}(X) \times \mathrm{Distr}(X)$ that can be equivalently defined in one of the following ways:

- $d \sqsubseteq_R e$ iff there exists a weight function for (d, e) w.r.t. R;
- $d \sqsubseteq_R e$ iff $d(U) \leq e(R(U))$ for any set $U \subseteq \mathrm{supp}(d)$;
- when R is an equivalence on X, then $d \sqsubseteq_R e$ iff $d(B) = e(B)$ for any equivalence class B of R. □

It is easy to see that if $R \subseteq R'$ then $\sqsubseteq_R \subseteq \sqsubseteq_{R'}$; moreover, if R is symmetric then $\sqsubseteq_R$ is also a symmetric relation, that we denote with $\equiv_R$.

Definition 2.2 (Simulation). Given a PLTS $\mathcal{M}$, a relation $R \subseteq \Sigma \times \Sigma$ is a simulation on $\mathcal{M}$ if for all $s, t \in \Sigma$ such that sRt,

- if $s \xrightarrow{a} d$ then there exists $e \in \mathrm{Distr}(\Sigma)$ such that $t \xrightarrow{a} e$ and $d \sqsubseteq_R e$. □

Let $R_{\mathrm{sim}} \triangleq \cup\{R \subseteq \Sigma \times \Sigma \mid R$ is a simulation on $\mathcal{M}\}$. Then, R_{sim} turns out to be a preorder relation which is the greatest simulation on $\mathcal{M}$ and is called simulation preorder on $\mathcal{M}$. Simulation equivalence P_{sim} on $\mathcal{M}$ is defined as the kernel of the simulation preorder, i.e., $P_{\mathrm{sim}} \triangleq R_{\mathrm{sim}} \cap R_{\mathrm{sim}}^{-1}$.

Definition 2.3 (Bisimulation). A symmetric relation $S \subseteq \Sigma \times \Sigma$ is a bisimulation on $\mathcal{M}$ if for all $s, t \in \Sigma$ such that sSt,

- if $s \xrightarrow{a} d$ then there exists $e \in \mathrm{Distr}(\Sigma)$ such that $t \xrightarrow{a} e$ and $d \equiv_S e$. □

Let $P_{\mathrm{bis}} \triangleq \cup\{S \subseteq \Sigma \times \Sigma \mid S$ is a bisimulation on $\mathcal{M}\}$. Then, P_{bis} turns out to be an equivalence relation which is the greatest bisimulation on $\mathcal{M}$ and is called bisimilarity on $\mathcal{M}$.

3 An Operational View of Probabilistic HML

In order to logically characterize behavioral relations on probabilistic models that encode pure nondeterminism, such as PLTSs, Parma and Segala [13] put forward an extension of Hennessy-Milner logic whose formulae are interpreted over distributions on the states of a PLTS. They show that two states are bisimilar if and only if their corresponding Dirac distributions satisfy the same set of formulae. However, nothing is stated about logically equivalent distributions that are not Dirac distributions. In the following, we give a novel notion of simulation (and correspondingly bisimulation) between distributions which (i) characterizes the full logical equivalence of Parma and Segala's logic and (ii) boils down to standard simulation (and bisimulation) between the states of a PLTS when restricted to Dirac distributions.

Parma and Segala's logic [13] is syntactically defined as follows:

$$\phi ::= \top \mid \bigwedge_{i\in I} \phi_i \mid \neg\phi \mid \Diamond_a\phi \mid [\phi]_p$$

where I is a possibly infinite (denumerable) set of indices, $a \in Act$ and p is a rational number in $[0,1]$. Given a PLTS $\langle \Sigma, Act, \rightarrow\rangle$, the semantics of the formulae is inductively defined as follows: for any distribution $d \in \mathrm{Distr}(\Sigma)$,

$$\begin{array}{l}
d \models \top \\
d \models \bigwedge_I \phi_i \text{ iff for any } i \in I,\ d \models \phi_i \\
d \models \neg\phi \text{ iff } d \not\models \phi \\
d \models \Diamond_a\phi \text{ iff } \forall x \in \mathrm{supp}(d).\, \exists e \in \mathrm{Distr}(\Sigma).\ x \xrightarrow{a} e \text{ and } e \models \phi \\
d \models [\phi]_p \text{ iff } d(\{s \in \Sigma \mid \delta_s \models \phi\}) \geq p
\end{array}$$

The first three clauses are standard. The modal connective $\Diamond_a$ is a probabilistic counterpart of HML's diamond operator. $\Diamond_a\phi$ is satisfied by a distribution $d \in \mathrm{Distr}(\Sigma)$ whenever *any state* $x \in \mathrm{supp}(d)$ reaches through an a-labeled transition a distribution e that satisfies the formula ϕ. As the formulae $\Diamond_a\phi$ only deal with transitions of the PLTS, a further modal operator $[\cdot]_p$ needs to take into account the probabilities that distributions assign to sets of related states. More precisely, a distribution d satisfies a formula $[\phi]_p$ when d assigns a probability at least p to the set of states whose Dirac distributions satisfy the formula ϕ. This logic is here referred to as $\mathcal{L}_\forall$ in order to stress the *universal* nature of its diamond operator $\Diamond_a$.

Definition 3.1 (Logical equivalence and preorder). Two distributions $d, e \in \mathrm{Distr}(\Sigma)$ are logically equivalent for $\mathcal{L}_\forall$, written $d \equiv_{\mathcal{L}_\forall} e$, when, for any $\phi \in \mathcal{L}_\forall$, $d \models \phi$ iff $e \models \phi$. We write $d \leq_{\mathcal{L}_\forall} e$ for the corresponding logical preorder, i.e., when for any $\phi \in \mathcal{L}_\forall$, $d \models \phi$ implies $e \models \phi$. □

Let $\mathcal{L}_\forall^+$ be the negation-free and finitely disjunctive fragment of $\mathcal{L}_\forall$, that is:

$$\phi ::= \top \mid \bigwedge_{i\in I} \phi_i \mid \phi \vee \phi \mid \Diamond_a\phi \mid [\phi]_p$$

The following result by Parma and Segala [13] (see also [11]) shows that the logical equivalence induced by $\mathcal{L}_\forall$ and the logical preorder induced by $\mathcal{L}_\forall^+$, when restricted to Dirac distributions, correspond, respectively, to bisimulation and simulation. Notice that the simulation preorder is logically characterized by negation-free formulae, reflecting the fact that simulation, differently from bisimulation, is not a symmetric relation. However, the logic for simulation requires finite disjunction to characterize probabilistic choice.

Theorem 3.2 **([13]).** *Consider R_{sim} and P_{bis} on a given PLTS. Then, for all $s, t \in \Sigma$,*

- *$s\, R_{\mathrm{sim}}\, t$ if and only if $\delta_s \leq_{\mathcal{L}_\forall^+} \delta_t$;*
- *$s\, P_{\mathrm{bis}}\, t$ if and only if $\delta_s \equiv_{\mathcal{L}_\forall} \delta_t$.*

Our main goal is to define a notion of simulation and bisimulation between distributions that represents the operational match of the full logical preorder $\leq_{\mathcal{L}_\forall^+}$ and equivalence $\equiv_{\mathcal{L}_\forall}$ between distributions. Firstly, notice that any relation on distributions $\mathcal{R} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ embeds a corresponding relation on states that can be obtained by restricting $\mathcal{R}$ to Dirac distributions. This is formalized by a mapping $\Delta : \wp(\mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)) \to \wp(\Sigma \times \Sigma)$ defined as follows:

$$\Delta(\mathcal{R}) \triangleq \{(s,t) \in \Sigma \times \Sigma \mid (\delta_s, \delta_t) \in \mathcal{R}\}.$$

Note that if $\mathcal{R}$ is a symmetric/preorder/equivalence relation then $\Delta(\mathcal{R})$ is correspondingly a symmetric/preorder/equivalence relation on Σ.

Our definition of (bi)simulation between distributions (called d-(bi)simulation) is directly inspired by the logic $\mathcal{L}_\forall$. In particular, the two distinctive modal operators of $\mathcal{L}_\forall$ are mirrored in two defining conditions of (bi)simulation between distributions. More precisely, the semantics of the diamond operator suggests a kind of transfer property that (bi)similar distributions should respect (cf. condition (1)). On the other hand, a second condition, peculiar of the probabilistic setting, deals with the probabilities assigned by (bi)similar distributions to sets of related states (cf. condition (2)).

Definition 3.3 **($\forall$d-simulation).** A relation $\mathcal{R} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ is a $\forall$*d-simulation* on a PLTS if for all $d, e \in \mathrm{Distr}(\Sigma)$, if $d\, \mathcal{R}\, e$ then:

(1) for all $D \subseteq \mathrm{Distr}(\Sigma)$, if $\mathrm{supp}(d) \subseteq \mathrm{pre}_a(D)$ then $\mathrm{supp}(e) \subseteq \mathrm{pre}_a(\mathcal{R}(D))$;
(2) $d \sqsubseteq_{\Delta(\mathcal{R})} e$. □

Definition 3.4 **($\forall$d-bisimulation).** A symmetric relation $\mathcal{S} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ is a $\forall$*d-bisimulation* on a PLTS if for all $d, e \in \mathrm{Distr}(\Sigma)$, if $d\, \mathcal{S}\, e$ then:

(1) for all $D \subseteq \mathrm{Distr}(\Sigma)$, if $\mathrm{supp}(d) \subseteq \mathrm{pre}_a(D)$ then $\mathrm{supp}(e) \subseteq \mathrm{pre}_a(\mathcal{S}(D))$;
(2) $d \equiv_{\Delta(\mathcal{S})} e$. □

Given a PLTS $\mathcal{M}$, let $\mathcal{R}^\forall_{\mathrm{sim}} \triangleq \cup\{\mathcal{R} \mid \mathcal{R} \text{ is a } \forall\text{d-simulation on } \mathcal{M}\}$. Then, it turns out that $\mathcal{R}^\forall_{\mathrm{sim}}$ is the greatest $\forall$d-simulation on $\mathcal{M}$ and is a preorder, called the $\forall$d-simulation preorder on $\mathcal{M}$. Analogously, let $\mathcal{P}^\forall_{\mathrm{bis}} \triangleq \cup\{\mathcal{S} \mid \mathcal{S} \text{ is a } \forall\text{d-bisimulation on } \mathcal{M}\}$, so that $\mathcal{P}^\forall_{\mathrm{bis}}$ turns out to be the greatest $\forall$d-bisimulation on $\mathcal{M}$ and an equivalence relation, called the $\forall$d-bisimilarity on $\mathcal{M}$

It turns out that $\forall$d-simulation preorder fully captures the logical preorder induced by $\mathcal{L}_\forall^+$ while $\forall$d-bisimilarity fully captures the logical equivalence induced by $\mathcal{L}_\forall$.

Theorem 3.5. *For any $d, e \in \mathrm{Distr}(\Sigma)$,*

- *$d\, \mathcal{R}^\forall_{\mathrm{sim}}\, e$ if and only $d \leq_{\mathcal{L}_\forall^+} e$;*
- *$d\, \mathcal{P}^\forall_{\mathrm{bis}}\, e$ if and only $d \equiv_{\mathcal{L}_\forall} e$.*

A closer look at the semantics of the diamond operator of $\mathcal{L}_\forall$ points out a key difference with the semantics of the standard diamond operator in HML. In the case of LTSs, the diamond operator of HML induces the predecessor operator of the LTS. Similarly, the semantic definition of the diamond operator of $\mathcal{L}_\forall$ induces the following operator $\mathrm{ppre}_a^\forall$, that we call probabilistic predecessor operator:

$$\mathrm{ppre}_a^\forall : \wp(\mathrm{Distr}(\Sigma)) \to \wp(\mathrm{Distr}(\Sigma))$$
$$\mathrm{ppre}_a^\forall(D) \triangleq \{d \in \mathrm{Distr}(\Sigma) \mid \mathrm{supp}(d) \subseteq \mathrm{pre}_a(D)\}$$

where $\mathrm{pre}_a : \wp(\mathrm{Distr}(\Sigma)) \to \wp(\Sigma)$ is the PLTS predecessor operator. However, differently from the predecessor operators of LTSs and PLTSs, this probabilistic predecessor $\mathrm{ppre}_a^\forall$ *does not preserve set unions*, i.e., it is not true in general that, for any $D_1, D_2 \subseteq \mathrm{Distr}(\Sigma)$, $\mathrm{ppre}_a^\forall(D_1 \cup D_2) = \mathrm{ppre}_a^\forall(D_1) \cup \mathrm{ppre}_a^\forall(D_2)$. In fact, $\mathrm{supp}(d) \subseteq \mathrm{pre}_a(D_1 \cup D_2)$ does not imply $\mathrm{supp}(d) \subseteq \mathrm{pre}_a(D_1)$ nor $\mathrm{supp}(d) \subseteq \mathrm{pre}_a(D_2)$. It is here worth noting that, in general, an operator $f : \wp(X) \to \wp(X)$ defined on a powerset $\wp(X)$ preserves set unions if and only if there exists a relation $R \subseteq X \times X$ whose corresponding predecessor operator $\mathrm{pre}_R = \lambda Y.\{x \in X \mid \exists y \in Y. xRy\}$ coincides with f. As a consequence, *one cannot define* a transition relation between distributions whose corresponding predecessor operator coincides with $\mathrm{ppre}_a^\forall$. The lack of a transition relation between distributions is particularly troublesome when defining coinductive behavioral relations between distributions. Consider the transfer property of ∀d-simulations, namely condition (1) of Definition 3.3: this can be equivalently stated as

$$\text{if } d \in \mathrm{ppre}_a^\forall(D) \text{ then } e \in \mathrm{ppre}_a^\forall(\mathcal{R}(D)) \qquad (1)$$

Since $\mathrm{ppre}_a^\forall$ does not preserve set unions, the statement $d \in \mathrm{ppre}_a^\forall(D)$ is not equivalent to $\exists f \in D.\, d \in \mathrm{ppre}_a^\forall(f)$, so that the above condition (1) does not scale to the standard transfer property of (bi)simulations on LTSs that naturally admits a game characterization. It is therefore interesting to ask whether a suitable definition of an additive (i.e., union-preserving) probabilistic predecessor operator between distributions can be found. In the following, this question will be positively answered.

3.1 LTS on Distributions

Let us consider the following alternative definition of probabilistic predecessor operator:

$$\mathrm{ppre}_a : \wp(\mathrm{Distr}(\Sigma)) \to \wp(\mathrm{Distr}(\Sigma))$$
$$\mathrm{ppre}_a(D) \triangleq \{d \in \mathrm{Distr}(\Sigma) \mid \mathrm{supp}(d) \cap \mathrm{pre}_a(D) \neq \varnothing\}$$

This definition is much less restrictive than that of the above $\mathrm{ppre}_a^\forall$ operator: in order for a distribution d to be a probabilistic predecessor of a distribution e it is now sufficient that the support of d contains some state that reaches e. In this sense, ppre_a has an *existential* flavour as opposed to the *universal* flavour of $\mathrm{ppre}_a^\forall$. In the following, this observation will be also formalized by means of abstract interpretation [1,2].

Since the ppre_a operator actually preserves set unions, a corresponding transition relation between distributions can be defined as follows: $d \xrightarrow{a} e$ iff $d \in \mathrm{ppre}_a(\{e\})$, namely,

$$d \xrightarrow{a} e \text{ iff } \exists s \in \mathrm{supp}(d).\ s \xrightarrow{a} e \qquad (*)$$

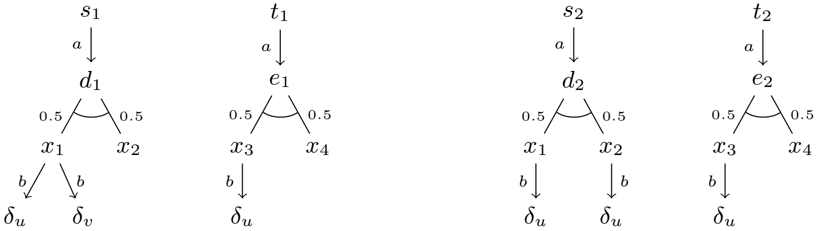


Fig. 1. A pair of PLTSs

This allows us to lift a PLTS to an LTS of distributions, that we call DLTS. Hence, the following notions of simulation/bisimulation based on the standard transfer property naturally arise.

Definition 3.6 (d-simulation). Given a PLTS $\mathcal{M}$, a relation $\mathcal{R} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ is a *d-simulation* on $\mathcal{M}$ if for all $d, e \in \mathrm{Distr}(\Sigma)$, if $d \, \mathcal{R} \, e$ then:

(1) if $d \xrightarrow{a} f$ then there exists $g \in \mathrm{Distr}(\Sigma)$ such that $e \xrightarrow{a} g$ and $f \, \mathcal{R} \, g$;
(2) $d \sqsubseteq_{\Delta(\mathcal{R})} e$. □

Definition 3.7 (d-bisimulation). Given a PLTS $\mathcal{M}$, a symmetric relation $\mathcal{S} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ is a *d-bisimulation* on $\mathcal{M}$ if for all $d, e \in \mathrm{Distr}(\Sigma)$, if $d \, \mathcal{S} \, e$ then:

(1) if $d \xrightarrow{a} f$ then there exists $g \in \mathrm{Distr}(\Sigma)$ such that $e \xrightarrow{a} g$ and $f \, \mathcal{S} \, g$;
(2) $d \equiv_{\Delta(\mathcal{S})} e$. □

Given a PLTS $\mathcal{M}$, let $\mathcal{R}_{\mathrm{sim}} \triangleq \cup \{\mathcal{R} \mid \mathcal{R}$ is a d-simulation on $\mathcal{M}\}$ and $\mathcal{P}_{\mathrm{bis}} \triangleq \cup \{\mathcal{S} \mid \mathcal{S}$ is a d-bisimulation on $\mathcal{M}\}$. Then, $\mathcal{R}_{\mathrm{sim}}$ turns out to be the greatest d-simulation on $\mathcal{M}$ and a preorder, called the d-simulation preorder on $\mathcal{M}$. Likewise, it turns out that $\mathcal{P}_{\mathrm{bis}}$ is the greatest d-bisimulation on $\mathcal{M}$ and an equivalence, called d-bisimilarity on $\mathcal{M}$

Interestingly, d-simulations (and analogously for d-bisimulations) enjoy a neat correspondence with state simulations in a PLTS. More precisely, the state simulation preorder R_{sim} can be recovered from the d-simulation preorder $\mathcal{R}_{\mathrm{sim}}$ by restricting $\mathcal{R}_{\mathrm{sim}}$ to Dirac distributions. On the other hand, the d-simulation preorder coincides with the lifting to distributions of the state simulation preorder.

Theorem 3.8.

- $\Delta(\mathcal{R}_{\mathrm{sim}}) = R_{\mathrm{sim}}$ *and* $\mathcal{R}_{\mathrm{sim}} = \sqsubseteq_{R_{\mathrm{sim}}}$.
- $\Delta(\mathcal{P}_{\mathrm{bis}}) = P_{\mathrm{bis}}$ *and* $\mathcal{P}_{\mathrm{bis}} = \equiv_{P_{\mathrm{bis}}}$.

It is worth noting that this result opens the way to define new model checking tools that compute (bi)simulations on PLTSs by adapting to DLTSs the standard (bi)simulation techniques/algorithms designed in the nonprobabilistic framework.

Example 3.9. Consider the leftmost PLTS depicted in Figure 1. The relation $\mathcal{R}_1 = \{(\delta_{s_1}, \delta_{t_1}), (d_1, e_1)\} \cup \{(d, d) \mid d \in \mathrm{Distr}(\Sigma)\}$ is not a d-simulation since $d_1 \xrightarrow{b} \delta_v$ but $e_1 \xrightarrow{b} \delta_u$ and $\delta_u \notin \mathcal{R}_1(\delta_v)$. Moreover, even if $\mathcal{R}_1$ respects the transfer property of $\forall$d-simulations (since there is no set $D \subseteq \mathrm{Distr}(\Sigma)$ such that $\mathrm{supp}(d_1) \subseteq \mathrm{pre}_a(D)$), $\mathcal{R}_1$ is not even a $\forall$d-simulation because $d_1 \not\sqsubseteq_{\Delta(\mathcal{R}_1)} e_1$, since, for instance, $0.5 = d(\{x_2\}) \not\leq e(\Delta(\mathcal{R}_1)(\{x_2\})) = e(\{x_2\}) = 0$. Nevertheless, s_1 and t_1 are bisimilar states since there exists a ($\forall$)d-bisimulation containing the pair $(\delta_{s_1}, \delta_{t_1})$. Let $\mathcal{R}$ be the equivalence relation corresponding to the partition $\{\{\delta_{s_1}, \delta_{t_1}\}, \{d_1, e_1\}, \{\delta_{x_1}, \delta_{x_3}\}, \{\delta_{x_2}, \delta_{x_4}, \delta_u, \delta_v\}\}$. It is not difficult to check that $\mathcal{R}$ is a ($\forall$)d-bisimulation: every pair in $\mathcal{R}$ respects the transfer property and is $\equiv_{\Delta(\mathcal{R})}$-equivalent, where $\Delta(\mathcal{R}) = \{\{s_1, t_1\}, \{x_1, x_3\}, \{u, v, x_2, x_4\}\}$.

As a further example, consider the rightmost PLTS in Figure 1. We have that s_2 simulates t_2 but t_2 does not simulate s_2. In fact, consider the relation

$$\mathcal{R}_2 = \{(\delta_{t_2}, \delta_{s_2}), (e_2, d_2), (\delta_u, \delta_u)\} \cup \{(\delta_{x_4}, \delta_{x_i})\}_{i=1,\ldots,4} \cup \{(\delta_{x_3}, \delta_{x_i})\}_{i=1,2,3}.$$

Then, $\mathcal{R}_2$ is a ($\forall$)d-simulation since every pair respects the transfer property and belongs to $\sqsubseteq_{\Delta(\mathcal{R}_2)}$. For instance, let us check that $e_2 \sqsubseteq_{\Delta(\mathcal{R}_2)} d_2$: by Definition 2.1, it is enough to check that for all $U \subseteq \mathrm{supp}(e_2)$, $e_2(U) \leq d_2(\Delta(\mathcal{R}_2)(U))$. The nonempty subsets of $\mathrm{supp}(e_2)$ are: $U_1 = \{x_3\}$, $U_2 = \{x_4\}$ and $U_3 = \{x_3, x_4\}$, so that we have

$$\begin{aligned}
0.5 = e_2(\{x_3\}) &\leq d_2(\Delta(\mathcal{R}_2)(\{x_3\})) = d_2(\{x_1, x_2, x_3\}) = 1\\
0.5 = e_2(\{x_4\}) &\leq d_2(\Delta(\mathcal{R}_2)(\{x_4\})) = d_2(\{x_1, x_2, x_3, x_4\}) = 1\\
1 = e_2(\{x_3, x_4\}) &\leq d_2(\Delta(\mathcal{R}_2)(\{x_3, x_4\})) = d_2(\{x_1, x_2, x_3, x_4\}) = 1
\end{aligned}$$

The fact that t_2 does not simulate s_2 depends on the fact that e_2 does not simulate d_2 since this would imply that there exists a ($\forall$)d-simulation $\mathcal{R}$ such that $d_2 \sqsubseteq_{\Delta(\mathcal{R})} e_2$. However, the latter statement implies that $1 = d_2(\{x_1, x_2\}) \leq e_2(\Delta(\mathcal{R})(\{x_1, x_2\}))$, which is true only if $\mathrm{supp}(e_2) = \{x_3, x_4\} \subseteq \Delta(\mathcal{R})(\{x_1, x_2\})$; hence, in particular, we would obtain $\delta_{x_4} \in \mathcal{R}(\{\delta_{x_1}, \delta_{x_2}\})$, which is a contradiction since x_4 cannot simulate a b-transition. □

Besides the above notions of d-simulation/d-bisimulation, the operator ppre_a allows us to provide a corresponding new interpretation for the diamond connective. Let $\mathcal{L}$ denote the logic whose syntax coincides with $\mathcal{L}_\forall$ and whose semantics is identical to that of $\mathcal{L}_\forall$ but for the diamond connective, which is interpreted as follows:

$$d \models \Diamond_a \phi \text{ iff } \exists e.\ d \xrightarrow{a} e \text{ and } e \models \phi$$

This is therefore the *standard interpretation* of the diamond operator on a DLTS, namely a LTS whose "states" are distributions and whose transitions are defined by $(*)$. In the following, we will argue that $\mathcal{L}$ is best suited as probabilistic extension of Hennessy-Milner logic. As a first result, it turns out that the preorder $\leq_{\mathcal{L}^+}$ and the equivalence $\equiv_{\mathcal{L}}$ logically characterize, respectively, d-simulations and d-bisimulations.

Theorem 3.10. *For any* $d, e \in \mathrm{Distr}(\Sigma)$*,*

- $d\,\mathcal{R}_{\mathrm{sim}}\,e$ *if and only* $d \leq_{\mathcal{L}^+} e$*;*
- $d\,\mathcal{P}_{\mathrm{bis}}\,e$ *if and only* $d \equiv_{\mathcal{L}} e$*.*

3.2 Comparing $\mathcal{L}_\forall$ and $\mathcal{L}$

It turns out that $\forall$d-(bi)simulations and d-(bi)simulations are equivalent notions. In spite of the fact they rely on quite different transfer properties (cf. condition (1) of Definitions 3.3 and 3.6), their second defining condition, peculiar to the probabilistic setting, is powerful enough to bridge their gap. More precisely, this depends on the following key property: if $d \sqsubseteq_R e$ then for any state s in the support of d there exists a state t in the support of e such that $t \in R(s)$, and viceversa, for any state t in the support of e there exists a state s in the support of d such that $t \in R(s)$.

Lemma 3.11. *Consider a PLTS $\mathcal{M}$ and a relation $\mathcal{R} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$. Then, $\mathcal{R}$ is a $\forall$d-(bi)simulation on $\mathcal{M}$ iff $\mathcal{R}$ is a d-(bi)simulation on $\mathcal{M}$.*

As a consequence, we have that $\mathcal{R}^\forall_{\mathrm{sim}} = \mathcal{R}_{\mathrm{sim}}$ and $\mathcal{P}^\forall_{\mathrm{bis}} = \mathcal{P}_{\mathrm{bis}}$, so that, by Theorems 3.5 and 3.10, the two modal logics $\mathcal{L}_\forall$ and $\mathcal{L}$ induce the same equivalence on distributions while $\mathcal{L}^+_\forall$ and $\mathcal{L}^+$ induce the same preorder on distributions. As far as their relative expressive powers are concerned, we have that $\mathcal{L}_\forall$ and $\mathcal{L}$ are equivalent, while this is not the case for their negation-free fragments.

Theorem 3.12.

- *$\mathcal{L}_\forall$ and $\mathcal{L}$ have the same expressive power (and therefore $\equiv_{\mathcal{L}_\forall} = \equiv_{\mathcal{L}}$).*
- *$\mathcal{L}^+_\forall$ is strictly less expressive than $\mathcal{L}^+$, although $\leq_{\mathcal{L}^+_\forall} = \leq_{\mathcal{L}^+}$.*

Let us observe that the equivalence between $\mathcal{L}_\forall$ and $\mathcal{L}$ depends on the fact that the semantics of the diamond operator of $\mathcal{L}_\forall$ can be encoded in $\mathcal{L}$ and viceversa. In particular, the semantics of the $\mathcal{L}_\forall$-formula $\Diamond_a\phi$, i.e. $[\![\Diamond_a\phi]\!]_{\mathcal{L}_\forall} = \{d \mid \mathrm{supp}(d) \subseteq \mathrm{pre}_a(\{e \mid e \models_{\mathcal{L}_\forall} \phi\})\}$, can be expressed in $\mathcal{L}$ by the formula $[\Diamond_a\phi]_1$, whose semantics is indeed $[\![[\Diamond_a\phi]_1]\!]_{\mathcal{L}} = \{d \mid d(\{x \mid \delta_x \models_{\mathcal{L}} \Diamond_a\phi\}) = 1\} = \{d \mid \mathrm{supp}(d) \subseteq \{x \mid \delta_x \models_{\mathcal{L}} \Diamond_a\phi\}\}$. On the other hand, the encoding of $\mathcal{L}$'s diamond as a $\mathcal{L}_\forall$-formula is more tricky. The semantics of $\Diamond_a\phi$ viewed as a $\mathcal{L}$-formula is given by all the distributions whose support contains at least a state that moves to a distribution that satisfies ϕ, i.e., $[\![\Diamond_a\phi]\!]_{\mathcal{L}} = \{d \mid d(\{x \mid \exists e.\, x \xrightarrow{a} e,\ e \models_{\mathcal{L}} \phi\}) > 0\}$. This semantics can be therefore expressed in $\mathcal{L}_\forall$ by requiring that $d \models_{\mathcal{L}_\forall} [\Diamond_a\phi]_p$ for some $p > 0$. However, in general the existence of a rational number $p > 0$ can be expressed as a logical formula only by means of an infinite (countable) disjunction, hence it is expressible in the full $\mathcal{L}_\forall$ logic, but not in its negation-free and finitely disjunctive fragment $\mathcal{L}^+_\forall$.

Let us describe an example showing that the logic $\mathcal{L}^+_\forall$ is strictly less expressive than $\mathcal{L}^+$. Consider a PLTS $\mathcal{M} = \langle \{x_1, x_2\}, \{a\}, \{x_1 \xrightarrow{a} d = (x_1/0.5, x_2/0.5)\}\rangle$ that contains two states x_1, x_2 and a single transition from x_1 to the distribution $d = (x_1/0.5, x_2/0.5)$. In the logic $\mathcal{L}$, we have that $[\![\Diamond_a\top]\!]_{\mathcal{L}} = \mathrm{Distr}(\Sigma) \smallsetminus \{\delta_{x_2}\}$, since any distribution different from δ_{x_2} contains x_1 in its support, and therefore has an outgoing a-transition. Let us show that there is no formula in $\mathcal{L}_\forall$ whose semantics is $\mathrm{Distr}(\Sigma) \smallsetminus \{\delta_{x_2}\}$. Consider the $\mathcal{L}_\forall$-formulae $\top$, $\Diamond_a\top$ and $[\Diamond_a\top]_p$, with $p > 0$, whose semantics are as follows: $[\![\top]\!]_{\mathcal{L}_\forall} = \mathrm{Distr}(\Sigma)$, $[\![\Diamond_a\top]\!]_{\mathcal{L}_\forall} = \{\delta_{x_1}\}$, $[\![[\Diamond_a\top]_p]\!]_{\mathcal{L}_\forall} = \{d \mid d(\{x_1\}) \geq p\}$. It is easily seen that the semantics of any other formula in $\mathcal{L}_\forall$ is in the set $\mathit{Sem}_{\mathcal{L}_\forall} = \{[\![\top]\!]_{\mathcal{L}_\forall}, [\![\Diamond_a\top]\!]_{\mathcal{L}_\forall}\} \cup \{[\![[\Diamond_a\top]_p]\!]_{\mathcal{L}_\forall} \mid p > 0\}$, which is indeed closed under infinite intersections, *finite* unions, probabilistic predecessor and the semantics of the operator $[\cdot]_p$. It is thus enough to observe that $\mathrm{Distr}(\Sigma) \smallsetminus \{\delta_{x_2}\} \notin \mathit{Sem}_{\mathcal{L}_\forall}$: actually, $\mathrm{Distr}(\Sigma) \smallsetminus \{\delta_{x_2}\}$ can only be expressed as the *infinite* union $\cup_{p>0} [\![[\Diamond_a \top]_p]\!]_{\mathcal{L}_\forall}$. □

Example 3.13. Consider again the rightmost PLTS in Figure 1. We have already observed that s_2 simulates t_2 whilst t_2 does not simulate s_2. The fact that t_2 does not simulate s_2 can be easily proved by exhibiting a formula that is satisfied by δ_{s_2} but not by δ_{t_2}. We provide both a formula in $\mathcal{L}_\forall$ and an equivalent formula in $\mathcal{L}$:

$$\text{(1) let } \phi \triangleq \Diamond_a \Diamond_b \top \in \mathcal{L}_\forall; \quad \text{then } \delta_{s_2} \models_{\mathcal{L}_\forall} \phi \text{ and } \delta_{t_2} \not\models_{\mathcal{L}_\forall} \phi$$

$$\text{(2) let } \phi' \triangleq \Diamond_a [\Diamond_b \top]_1 \in \mathcal{L}; \text{ then } \delta_{s_2} \models_{\mathcal{L}} \phi' \text{ and } \delta_{t_2} \not\models_{\mathcal{L}} \phi'$$

To see (1), observe that $\delta_{s_2} \models_{\mathcal{L}_\forall} \phi$ since $\mathrm{supp}(\delta_{s_2}) \subseteq \mathrm{pre}_a(d_2)$ and $\mathrm{supp}(d_2) \subseteq \mathrm{pre}_b(\delta_u)$ with $\delta_u \models_{\mathcal{L}_\forall} \top$. On the other hand, $\mathrm{supp}(\delta_{t_2}) \subseteq \mathrm{pre}_a(e_2)$ but $\mathrm{supp}(e_2) \not\subseteq \mathrm{pre}_b(f)$ for some distribution f such that $f \models_{\mathcal{L}_\forall} \top$. To show (2), notice that $\delta_{s_2} \models_{\mathcal{L}} \phi'$ since $\delta_{s_2} \xrightarrow{a} d_2$, and $d_2(\{x \mid \delta_x \models_{\mathcal{L}} \Diamond_b \top\}) = 1$ since for any $x \in \mathrm{supp}(d_2)$ it holds $\delta_x \xrightarrow{b} \delta_u$ with $\delta_u \models_{\mathcal{L}} \top$. On the other hand, $\delta_{t_2} \not\models_{\mathcal{L}} \phi'$ since $\delta_{t_2} \xrightarrow{a} e_2$ and $e_2(\{x \mid \delta_x \models_{\mathcal{L}} \Diamond_b \top\}) = 0.5 \not\geq 1$ since for $x_4 \in \mathrm{supp}(e_2)$ it holds $\delta_{x_4} \not\models_{\mathcal{L}} \Diamond_b \top$. □

4 States as Abstract Interpretation of Distributions

Differently from LTSs and their behavioral relations, whose definitions rely on a single notion of system state, PLTSs as well as their corresponding spectra of behavioral relations in some sense involve two notions of system state, namely a bare state and a probabilistic state modeled as a state distribution. We have shown above how PLTSs can be embedded into DLTSs, that is, LTSs of probabilistic states that involve a single (but richer) notion of system state, i.e. state distributions. We show in this section how to formalize a systematic embedding of states into distributions by viewing states as abstract interpretation of distributions.

Intuitively, Dirac distributions allow us to view states as an abstraction of distributions, namely the map $\delta : \Sigma \to \mathrm{Distr}(\Sigma)$ such that $\delta(x) \triangleq \delta_x$ may be viewed as a function that embeds states into distributions. The other way round, the support map $\mathrm{supp} : \mathrm{Distr}(\Sigma) \to \wp(\Sigma)$ can be viewed as a function that abstracts a distribution d as the set of states in its support.

Let us recall that in standard abstract interpretation [1,2], approximations of a concrete semantic domain are encoded by abstract domains that are specified by Galois insertions (GIs for short) or, equivalently, by adjunctions. Approximation on a concrete/abstract domain is encoded by a partial order where traditionally $x \leq y$ means that y is a concrete/abstract approximation of x. Concrete and abstract approximation orders, denoted by $\leq_C$ and $\leq_A$, must be related by a GI. Recall that a GI of an abstract domain $\langle A, \leq_A \rangle$ into a concrete domain $\langle C, \leq_C \rangle$ is determined by a surjective abstraction map $\alpha : C \to A$ and a 1-1 concretization map $\gamma : A \to C$ such that $\alpha(c) \leq_A a \;\Leftrightarrow\; c \leq_C \gamma(a)$ and is denoted by (α, C, A, γ). In a GI, intuitively $\alpha(c)$ provides the best approximation in A of a concrete value c while $\gamma(a)$ is the concrete value that a abstractly represents.

In our case, in order to cast δ as a concretization map in abstract interpretation, we need to lift its definition from sets of states to sets of distributions, namely we need to provide its so-called "collecting" version [1,2]. Observe that $\{\delta(x)\} = \{d \in \mathrm{Distr}(\Sigma) \mid \mathrm{supp}(d) \subseteq \{x\}\}$. This leads us to define the following concretization function $\gamma^\forall : \wp(\Sigma) \to \wp(\mathrm{Distr}(\Sigma))$:

$$\gamma^\forall(S) \triangleq \{d \in \mathrm{Distr}(\Sigma) \mid \mathrm{supp}(d) \subseteq S\}.$$

This is a *universal* concretization function, meaning that $d \in \gamma^\forall(S)$ iff all the states in $\mathrm{supp}(d)$ are contained into S. Hence, one can dually define an *existential* concretization map $\gamma^\exists : \wp(\Sigma) \to \wp(\mathrm{Distr}(\Sigma))$ as

$$\gamma^\exists(S) \triangleq \{d \in \mathrm{Distr}(\Sigma) \mid \mathrm{supp}(d) \cap S \neq \varnothing\},$$

where $d \in \gamma^\exists(S)$ if there exists some state in the support of d which is contained into S. Actually, these two mappings give rise to a pair of GIs (i.e., approximations in abstract interpretation) where $\wp(\mathrm{Distr}(\Sigma))$ and $\wp(\Sigma)$ play, respectively, the role of concrete and abstract domains. The approximation order is encoded by the subset relation (i.e., logical implication) in the case of $\gamma^\forall$ and by the superset relation (i.e., logical co-implication) in the case of $\gamma^\exists$. The dual maps, systematically obtained by adjunction from $\gamma^\forall$ and $\gamma^\exists$, are $\alpha^\forall, \alpha^\exists : \wp(\mathrm{Distr}(\Sigma)) \to \wp(\Sigma)$ defined as follows:

$$\begin{aligned} \alpha^\forall(X) &\triangleq \{s \in \Sigma \mid \exists d \in X.\ s \in \mathrm{supp}(d)\} \\ \alpha^\exists(X) &\triangleq \{s \in \Sigma \mid \forall d \in \mathrm{Distr}(\Sigma).\ s \in \mathrm{supp}(d) \Rightarrow d \in X\} \end{aligned}$$

Lemma 4.1. *$(\alpha^\forall, \wp(\mathrm{Distr}(\Sigma))_\subseteq, \wp(\Sigma))_\subseteq, \gamma^\forall)$ and $(\alpha^\exists, \wp(\mathrm{Distr}(\Sigma))_\supseteq, \wp(\Sigma))_\supseteq, \gamma^\exists)$ are GIs.*

Observe that $\alpha^\forall/\gamma^\forall$ and $\alpha^\exists/\gamma^\exists$ are dual abstractions, i.e.,

$$\alpha^\exists = \neg\, \alpha^\forall \neg \text{ and } \gamma^\exists = \neg\, \gamma^\forall \neg$$

where $\neg\, \alpha^\forall \neg(X) = \Sigma \smallsetminus \alpha^\forall(\mathrm{Distr}(\Sigma) \smallsetminus X)$ and $\neg\, \gamma^\forall \neg(S) = \mathrm{Distr}(\Sigma) \smallsetminus \gamma^\forall(\Sigma \smallsetminus S)$. Moreover, it is not hard to see that $\alpha^\forall$ is the additive extension of the supp function, while $\alpha^\exists$ is its co-additive extension, i.e.,

$$\alpha^\forall(X) = \cup_{d \in X}\, \mathrm{supp}(d) \text{ and } \alpha^\exists(\mathrm{Distr} \smallsetminus X) = \cap_{d \in X}\, \Sigma \smallsetminus \mathrm{supp}(d).$$

These two abstract domains thus provide dual universal/existential ways for logically approximating sets of distributions into sets of states. The interesting point in these formal abstractions lies in the fact that they allow us to systematically obtain the above probabilistic predecessor operators $\mathrm{ppre}_a^\forall$ and ppre_a in a DLTS from the predecessor operator pre_a of the corresponding PLTS. Recall that in a PLTS the predecessor operator $\mathrm{pre}_a : \wp(\mathrm{Distr}(\Sigma)) \to \wp(\Sigma)$ maps a set of distributions into a set of states. Here, $\wp(\Sigma)$ can therefore be viewed as a universal/existential abstraction of $\wp(\mathrm{Distr}(\Sigma))$, so that, correspondingly, pre_a can be viewed as an *abstract predecessor function*, since its co-domain actually is an abstract domain. Consequently, the output of this abstract function can be projected back to distributions using the corresponding concretization map. Interestingly, it turns out that the corresponding *concrete predecessor functions*, obtained by composing the operator pre_a with either $\gamma^\forall$ or $\gamma^\exists$, exactly coincide with the two probabilistic predecessors $\mathrm{ppre}_a^\forall$ and ppre_a.

Lemma 4.2. $\mathrm{ppre}^{\forall}_{a} = \gamma^{\forall} \circ \mathrm{pre}_a$ *and* $\mathrm{ppre}_a = \gamma^{\exists} \circ \mathrm{pre}_a$.

Thus, in equivalent terms, the predecessor operator pre_a is the best correct universal/existential approximation of the operators $\mathrm{ppre}^{\forall}_{a}$/$\mathrm{ppre}_a$, for the universal/existential abstractions $\alpha^{\forall}/\gamma^{\forall}$ and $\alpha^{\exists}/\gamma^{\exists}$.

5 A Spectrum of Probabilistic Relations over DLTSs

The approach developed above suggests a general methodology for defining behavioral relations between the states of a PLTS: first define a "lifted" behavioral relation between distributions of the corresponding DLTS and then restrict this definition to Dirac distributions. As discussed above, this approach works satisfactorily for simulation and bisimulation on PLTSs. In what follows, we show that this technique is indeed more general since it can be applied to a number of known probabilistic behavioral relations.

5.1 Probabilistic Simulation

Segala and Lynch [15] put forward a variant of simulation where a state transition $s \xrightarrow{a} d$ can be matched by a so-called combined transition from a state t, namely a convex combination of distributions reachable from t. We show that this same idea can be lifted to transitions in DLTSs.

Let $\mathcal{M} = \langle \Sigma, Act, \to \rangle$ be a PLTS, let $\{s \xrightarrow{a} d_i\}_{i \in I}$ be a (denumerable) family of transitions of $\mathcal{M}$ and let $\{p_i\}_{i \in I}$ be a corresponding family of probabilities in $[0, 1]$ such that $\sum_{i \in I} p_i = 1$. Let $d \in \mathrm{Distr}(\Sigma)$ be the convex combination $d = \sum_{i \in I} \; p_i d_i$. Then, $\{s, a, \sum_{i \in I} p_i d_i\}$, denoted by $s \overset{a}{\leadsto} d$, is called a *combined transition* in $\mathcal{M}$. This notion of combined transition can be lifted to distributions as follows.

Definition 5.1 (Combined d-transitions and Hyper transitions).

- Let $d, e \in \mathrm{Distr}(\Sigma)$. Then, $d \overset{a}{\leadsto} e$ if there exists $s \in \mathrm{supp}(d)$ such that $s \overset{a}{\leadsto} e$. $d \overset{a}{\leadsto} e$ is called a *combined d-transition*.
- Let $\{d \xrightarrow{a} d_i\}_{i \in I}$ be a family of transitions in a DLTS, and let $\{p_i\}_{i \in I}$ be a corresponding family of probabilities such that $\sum_{i \in I} p_i = 1$. Let $d = \sum_{i \in I} p_i d_i$. Then, the triple $\{d, a, \sum_{i \in I} p_i d_i\}$, compactly denoted by $d \overset{a}{\rightrightarrows} e$, is called a *hyper transition*. □

It is worth noting that the notion of hyper transition is "stronger" than that of combined d-transition, in that $d \overset{a}{\leadsto} e$ implies $d \overset{a}{\rightrightarrows} e$ but not viceversa. Moreover, our definition of hyper transition can be compared with analogous notions of hyper transition defined in [16] and [5]. In particular, it can be shown that a hyper transition in the sense of both Stoelinga [16] and Deng et al. [5] is a hyper transition in our sense, but not vice versa. Anyhow, the notion of combined d-transition is sufficient to lift probabilistic (bi)simulations of [15] to distributions.

In what follows, we focus on simulation relations only, since the same results scale to bisimulations. A probabilistic simulation is defined as a simulation in a PLTS apart from using combined transitions rather than standard transitions of a PLTS. Correspondingly, probabilistic d-simulation is defined as in Definition 3.6, but using combined d-transitions rather than transitions in a DLTS.

Let R_{psim} ($\mathcal{R}_{\mathrm{psim}}$) be the union of all the probabilistic (d-)simulations on $\mathcal{M}$. Then, all the results obtained in Section 3 also hold for probabilistic simulation, and they are collected in the following theorem. In particular, as before, the probabilistic simulation preorder between states can be recovered from the probabilistic d-simulation preorder by restricting to Dirac distributions. Dually, the probabilistic d-simulation preorder coincides with the lifting of the probabilistic simulation preorder. As far as the logic is concerned, Parma and Segala [13] show that the probabilistic relations between the states of a PLTS are logically characterized by the logical equivalence/preorder — restricted to Dirac distributions — of a modal logic that has the same syntax of $\mathcal{L}_\forall$ but whose diamond operator is defined in terms of combined transitions on the PLTS. Let $\mathcal{L}_P$ denote the logic $\mathcal{L}$ (which is equivalent to $\mathcal{L}_\forall$) where the semantics of the diamond operator is defined in terms of combined d-transitions. Then, as in Section 3, the result in [13] can be extended by showing that the full logical preorder of $\mathcal{L}_P^+$ coincides with the probabilistic d-simulation preorder.

Theorem 5.2.

- $\Delta(\mathcal{R}_{\mathrm{psim}}) = R_{\mathrm{psim}}$ *and* $\sqsubseteq_{R_{\mathrm{psim}}} = \mathcal{R}_{\mathrm{psim}}$.
- $\mathcal{R}_{\mathrm{psim}} = \leq_{\mathcal{L}_P^+}$.

5.2 Failure Simulation

One nice consequence of defining DLTSs as LTSs of distributions lies in the fact that the standard van Glabbeek's spectrum [8] of behavioral relations on LTSs can be reformulated in terms of transitions between distributions of a DLTS. This leads to a spectrum of d-relations between distributions of a DLTS, that can be projected into a spectrum of relations between states of a PLTS by restricting the d-relations to Dirac distributions. As an example we show how this approach works on failure simulation [8]. A formalization and generalization of such a "lifting schema" in a suitable framework like abstract interpretation or coalgebras is left as future work.

Definition 5.3 (Failure Simulation). A relation $R \subseteq \Sigma \times \Sigma$ is a *failure simulation* on a PLTS when for any $s, t \in \Sigma$, if sRt then:

- if $s \xrightarrow{a} d$ then there exists $e \in \mathrm{Distr}(\Sigma)$ such that $t \xrightarrow{a} e$ and $d \sqsubseteq_R e$;
- if $s \stackrel{A}{\not\rightarrow}$ then $t \stackrel{A}{\not\rightarrow}$ for any $A \subseteq Act$. □

Definition 5.4 (Failure d-Simulation). A relation $\mathcal{R} \subseteq \mathrm{Distr}(\Sigma) \times \mathrm{Distr}(\Sigma)$ is a *failure d-simulation* on a PLTS when for all $d, e \in \mathrm{Distr}(\Sigma)$, if $d \,\mathcal{R}\, e$ then:

(1) if $d \xrightarrow{a} f$ then there exists $g \in \mathrm{Distr}(\Sigma)$ such that $e \xrightarrow{a} g$ and $f \,\mathcal{R}\, g$;
(2) if $d \stackrel{A}{\not\rightarrow}$ then $e \stackrel{A}{\not\rightarrow}$ for any $A \subseteq Act$;
(3) $d \sqsubseteq_{\Delta(\mathcal{R})} e$. □

The lifting of a relation between states of a PLTS to a relation between distributions of the corresponding DLTS is obtained by resorting to the standard transfer property and by adding the condition (i.e., condition (3) in Definition 5.4) that deals with probabilities assigned to sets of related states. Let R_{fail} and $\mathcal{R}_{\mathrm{fail}}$ be, respectively, the failure simulation and d-simulation preorders on a PLTS $\mathcal{M}$. According to the LTS spectrum, failure simulation can be logically characterized through a modality that characterizes which transitions cannot be fired. We follow this same approch and we denote by $\mathcal{L}^+_F$ the logic obtained from $\mathcal{L}^+$ by adding a modality $\mathsf{ref}\langle A\rangle$, where $A \subseteq Act$, and whose semantics is defined as follows: for any $d \in \mathrm{Distr}(\Sigma)$, $d \models_{\mathcal{L}^+_F} \mathsf{ref}\langle A\rangle$ iff $d \overset{A}{\nrightarrow}$.

Theorem 5.5.

- $\Delta(\mathcal{R}_{\mathrm{fail}}) = R_{\mathrm{fail}}$ *and* $\sqsubseteq_{R_{\mathrm{fail}}} = \mathcal{R}_{\mathrm{fail}}$*;*
- $\mathcal{R}_{\mathrm{fail}} = \leq_{\mathcal{L}^+_F}$.

6 Related and Future work

Simulation and bisimulation relations on PLTSs have been introduced by Segala and Lynch [15] as two equivalences that preserve significant classes of temporal properties in the probabilistic logic PCTL [10]. Since then a number of works put forward probabilistic extensions of Hennessy-Milner logic in order to logically characterize these equivalences. Larsen and Skou [12] and Desharnais et al. [7] investigated a probabilistic diamond operator that enhances the diamond operator of HML with the probability bounds of transitions. However, these logics are adequate just for reactive and alternating systems, which are probabilistic models that are strictly less expressive than PLTSs. Two further probabilistic variants of HML are available [13,5]. The first one is that of Parma and Segala [13] (see also [11]), whose formulae are interpreted on sets of probability distributions over the states of a PLTS. One distinctive operator of this logic is a modal operator $[\phi]_p$, whose semantics is the set of distributions that assigns at least probability p to the set of states whose Dirac distributions satisfy ϕ. This paper has shown that such a logic admits an equivalent formulation that retains the probabilistic operator $[\phi]_p$ and retrieves the diamond operator of HML by lifting it to distributions. Deng et al. [5] follow a different approach. They propose a probabilistic variant of HML that is interpreted on sets of processes of the pCSP process calculus. In their logic the semantics of the diamond operator is defined in terms of hyper transitions between distributions: this notion of hyper transition is more complex than ours and has been compared with our notion of hyper transitions in Section 5. Moreover, Deng et al.'s logic features a probabilistic operator $\bigoplus_{i\in I} p_i\phi_i$ that is satisfied by processes that correspond to distributions that can be decomposed into convex combinations of distributions that satisfy ϕ_i. Besides (bi)simulation and probabilistic (bi)simulation, this logic is able to characterize two notions of failure and forward simulation that have been proved to agree with the testing preorders on pCSP processes (see [5]).

Deng et al. [5]'s definition of failure simulation is quite different from ours, that we directly derived from the standard LTS spectrum. One major difference is that we define a relation between states of a PLTS which is then lifted to a relation between distributions, whereas Deng et al. consider a relation between states and distributions.

A precise comparison between the spectrum of behavioral relations on DLTSs and the behavioral relations defined by Deng et al. [5] is left as subject for future work. We also plan to investigate weak transitions in DLTSs that abstract from internal, invisible, actions. Weak variants of simulation, probabilistic simulation, forward and failure simulation have been studied both in [5] and [13].

As a further avenue of future work we plan to study whether and how behavioral relations on PLTSs can be computed by resorting to standard algorithms for LTSs that compute the corresponding lifted relations on a DLTS. A first step in this direction has been taken in [3], where efficient algorithms to compute simulation and bisimulation on PLTSs have been derived by resorting to abstract interpretation techniques.

Acknowledgements. This work was partially supported by University of Padova under the projects “AVIAMO” and “BECOM”.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM POPL (1977)
2. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. 6th ACM POPL, pp. 269–282 (1979)
3. Crafa, S., Ranzato, F.: Probabilistic bisimulation and simulation algorithms by abstract interpretation. In: Aceto, L. (ed.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 295–306. Springer, Heidelberg (2011)
4. Deng, Y., van Glabbeek, R.: Characterising probabilistic processes logically. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 278–293. Springer, Heidelberg (2010)
5. Deng, Y., van Glabbeek, R., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4) (2008)
6. Desharnais, J.: Labelled Markov Processes. PhD thesis, McGill Univ. (1999)
7. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Weak bisimulation is sound and complete for PCTL*. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 355–370. Springer, Heidelberg (2002)
8. van Glabbeek, R.J.: The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In: Handbook of Process Algebra, ch. 1, pp. 3–99. Elsevier, Amsterdam (2001)
9. van Glabbeek, R.J., Smolka, S., Steffen, B., Tofts, C.: Reactive, generative and stratified models for probabilistic processes. In: Proc. IEEE LICS 1990, pp. 130–141 (1990)
10. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
11. Hermanns, H., Parma, A., Segala, R., Wachter, B., Zhang, L.: Probabilistic logical characterization. Information and Computation 209(2), 154–172 (2011)
12. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94(1), 1–28 (1991)
13. Parma, A., Segala, R.: Logical characterizations of bisimulations for discrete probabilistic systems. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 287–301. Springer, Heidelberg (2007)
14. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. PhD thesis, MIT (1995)

15. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic J. of Computing 2(2), 250–273 (1995)
16. Stoelinga, M.: Alea Jacta Est: Verification of Probabilistic, Real-Time and Parametric Systems. PhD thesis, University of Nijmegen, The Netherlands (2002)
17. Zhang, L., Hermanns, H., Eisenbrand, F., Jansen, D.N.: Flow faster: efficient decision algorithms for probabilistic simulations. Logical Methods in Computer Science 4(4) (2008)

Fixed-Delay Events in Generalized Semi-Markov Processes Revisited*

Tomáš Brázdil, Jan Krčál, Jan Křetínský**, and Vojtěch Řehák

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{brazdil, krcal, jan.kretinsky, rehak}@fi.muni.cz

Abstract. We study long run average behavior of generalized semi-Markov processes with both fixed-delay events as well as variable-delay events. We show that allowing two fixed-delay events and one variable-delay event may cause an unstable behavior of a GSMP. In particular, we show that a frequency of a given state may not be defined for almost all runs (or more generally, an invariant measure may not exist). We use this observation to disprove several results from literature. Next we study GSMP with at most one fixed-delay event combined with an arbitrary number of variable-delay events. We prove that such a GSMP always possesses an invariant measure which means that the frequencies of states are always well defined and we provide algorithms for approximation of these frequencies. Additionally, we show that the positive results remain valid even if we allow an arbitrary number of reasonably restricted fixed-delay events.

1 Introduction

Generalized semi-Markov processes (GSMP), introduced by Matthes in [23], are a standard model for discrete-event stochastic systems. Such a system operates in continuous time and reacts, by changing its state, to occurrences of events. Each event is assigned a random delay after which it occurs; state transitions may be randomized as well. Whenever the system reacts to an event, new events may be scheduled and pending events may be discarded. To get some intuition, imagine a simple communication model in which a server sends messages to several clients asking them to reply. The reaction of each client may be randomly delayed, e.g., due to latency of communication links. Whenever a reply comes from a client, the server changes its state (e.g., by updating its database of alive clients or by sending another message to the client) and then waits for the rest of the replies. Such a model is usually extended by allowing the server to time-out and to take an appropriate action, e.g., demand replies from the remaining clients in a more urgent way. The time-out can be seen as another event which has a fixed delay.

More formally, a GSMP consists of a set S of states and a set $\mathcal{E}$ of events. Each state s is assigned a set $\mathbf{E}(s)$ of events *scheduled* in s. Intuitively, each event in $\mathbf{E}(s)$ is assigned a positive real number representing the amount of time which elapses before

* The authors are supported by the Institute for Theoretical Computer Science, project No. 1M0545, the Czech Science Foundation, grant No. P202/10/1469 (T. Brázdil, V. Řehák) and No. 102/09/H042 (J. Krčál), and Brno PhD Talent Financial Aid (J. Křetínský).

** On leave at TU München, Boltzmannstr. 3, Garching, Germany.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 140–155, 2011.

the event occurs. Note that several events may occur at the same time. Once a set of events $E \subseteq \mathbf{E}(s)$ occurs, the system makes a *transition* to a new state s'. The state s' is randomly chosen according to a fixed distribution which depends only on the state s and the set E. In s', the *old* events of $\mathbf{E}(s) \setminus \mathbf{E}(s')$ are discarded, each *inherited* event of $(\mathbf{E}(s') \cap \mathbf{E}(s)) \setminus E$ remains scheduled to the same point in the future, and each *new* event of $(\mathbf{E}(s') \setminus \mathbf{E}(s)) \cup (\mathbf{E}(s') \cap E)$ is newly scheduled according to its given probability distribution.

In order to deal with GSMP in a rigorous way, one has to impose some restrictions on the distributions of delays. Standard mathematical literature, such as [15,16], usually considers GSMP with continuously distributed delays. This is certainly a limitation, as some systems with fixed time delays (such as time-outs or processor ticks) cannot be faithfully modeled using only continuously distributed delays. We show some examples where fixed delays exhibit qualitatively different behavior than any continuously distributed approximation. In this paper we consider the following two types of events:

- *variable-delay*: the delay of the event is randomly distributed according to a probability density function which is continuous and positive either on a bounded interval $[\ell, u]$ or on an unbounded interval $[\ell, \infty)$;
- *fixed-delay*: the delay is set to a fixed value with probability one.

The desired behavior of systems modeled using GSMP can be specified by various means. One is often interested in long-run behavior such as mean response time, frequency of errors, etc. (see, e.g., [1]). For example, in the above communication model, one may be interested in average response time of clients or in average time in which all clients eventually reply. Several model independent formalisms have been devised for expressing such properties of continuous time systems. For example, a well known temporal logic CSL contains a steady state operator expressing frequency of states satisfying a given subformula. In [9], we proposed to specify long-run behavior of a continuous-time process using a timed automaton which observes runs of the process, and measure the frequency of locations of the automaton.

In this paper we consider a standard performance measure, the frequency of states of the GSMP. To be more specific, let us fix a state $\mathring{s} \in S$. We define a random variable $\mathbf{d}$ which to every run assigns the (discrete) frequency of visits to $\mathring{s}$ on the run, i.e. the ratio of the number of transitions entering $\mathring{s}$ to the number of all transitions. We also define a random variable $\mathbf{c}$ which gives timed frequency of $\mathring{s}$, i.e. the ratio of the amount of time spent in $\mathring{s}$ to the amount of time spent in all states. Technically, both variables $\mathbf{d}$ and $\mathbf{c}$ are defined as limits of the corresponding ratios on prefixes of the run that are prolonged ad infinitum. Note that the limits may not be defined for some runs. For example, consider a run which alternates between $\mathring{s}$ and another state s; it spends 2 time unit in $\mathring{s}$, then 4 in s, then 8 in $\mathring{s}$, then 16 in s, etc. Such a run does not have a limit ratio between time spent in $\mathring{s}$ and in s. We say that $\mathbf{d}$ (or $\mathbf{c}$) is well-defined for a run if the limit ratios exist for this run. Our goal is to characterize stable systems that have the variables $\mathbf{d}$ and $\mathbf{c}$ well-defined for almost all runs, and to analyze the probability distributions of $\mathbf{d}$ and $\mathbf{c}$ on these stable systems.

As a working example of GSMP with fixed-delay events, we present a simplified protocol for time synchronization. Using the variable $\mathbf{c}$, we show how to measure reliability of the protocol. Via message exchange, the protocol sets and keeps a client clock

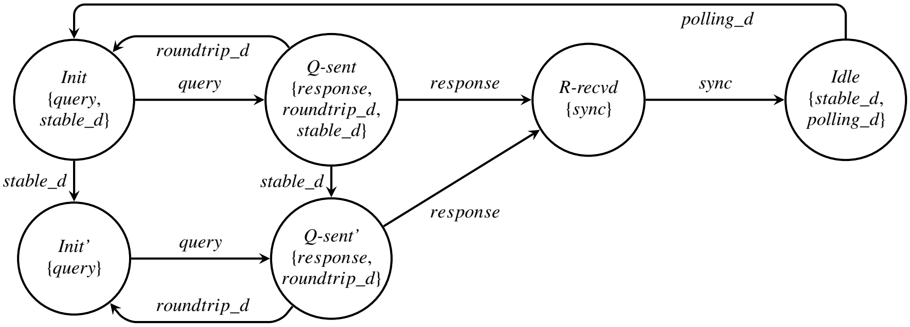


Fig. 1. A GSMP model of a clock synchronization protocol. Below each state label, we list the set of scheduled events. We only display transitions that can take place with non-zero probability.

sufficiently close to a server clock. Each message exchange is initialized by the client asking the server for the current time, i.e. sending a *query* message. The server adds a timestamp into the message and sends it back as a *response*. This query-response exchange provides a reliable data for *synchronization* action if it is realized within a given *round-trip delay*. Otherwise, the client has to repeat the procedure. After a success, the client is considered to be synchronized until a given *stable-time delay* elapses. Since the aim is to keep the clocks synchronized all the time, the client restarts the synchronization process sooner, i.e. after a given *polling delay* that is shorter than the stable-time delay. Notice that the client gets desynchronized whenever several unsuccessful synchronizations occur in a row. Our goal is to measure the portion of the time when the client clock is not synchronized.

Figure 1 shows a GSMP model of this protocol. The delays specified in the protocol are modeled using fixed-delay events *roundtrip_d*, *stable_d*, and *polling_d* while actions are modeled by variable-delay events *query*, *response*, and *sync*. Note that if the stable-time runs out before a fast enough response arrives, the systems moves into primed states denoting it is not synchronized at the moment. Thus, $\mathbf{c}(\mathit{Init'}) + \mathbf{c}(\mathit{Q\text{-}sent'})$ expresses the portion of the time when the client clock is not synchronized.

Our Contribution. So far, GSMP were mostly studied with variable-delay events only. There are a few exceptions such as [4,3,8,2] but they often contain erroneous statements due to presence of fixed-delay events. Our goal is to study the effect of mixing a number of fixed-delay events with an arbitrary amount of variable-delay events.

At the beginning we give an example of a GSMP with two fixed-delay events for which it is *not true* that the variables **d** and **c** are well-defined for almost all runs. We also disprove some crucial statements of [3,4]. In particular, we show an example of a GSMP which reaches one of its states with probability less than one even though the algorithms of [3,4] return the probability one. The mistake of these algorithms is fundamental as they neglect the possibility of unstable behavior of GSMP.

Concerning positive results, we show that if there is at most one fixed-delay event, then both **d** and **c** are almost surely well-defined. This is true even if we allow an arbitrary number of reasonably restricted fixed-delay events. We also show how to approximate distribution functions of $\mathbf{d}$ and $\mathbf{c}$. To be more specific, we show that for GSMP with at most one unrestricted and an arbitrary number of restricted fixed-delay events, both variables $\mathbf{d}$ and $\mathbf{c}$ have finite ranges $\{d_1, \ldots, d_n\}$ and $\{c_1, \ldots, c_n\}$. Moreover, all values d_i and c_i and probabilities $\mathcal{P}(\mathbf{d} = d_i)$ and $\mathcal{P}(\mathbf{c} = c_i)$ can be effectively approximated.

Related Work. There are two main approaches to the analysis of GSMP. One is to restrict the amount of events or types of their distributions and to solve the problems using symbolic methods [8,2,21]. The other is to estimate the values of interest using simulation [26,15,16]. Concerning the first approach, time-bounded reachability has been studied in [2] where the authors restricted the delays of events to so called expolynomial distributions. The same authors also studied reachability probabilities of GSMP where in each transition at most one event is inherited [8]. Further, the widely studied formalisms of semi-Markov processes (see, e.g., [20,9]) and continuous-time Markov chains (see, e.g., [6,7]) are both subclasses of GSMP.

As for the second approach, GSMP are studied by mathematicians as a standard model for discrete event simulation and Markov chains Monte Carlo (see, e.g., [14,17,25]). Our work is strongly related to [15,16] where the long-run average behavior of GSMP with variable-delay events is studied. Under relatively standard assumptions the stochastic process generated by a GSMP is shown to be irreducible and to possess an invariant measure. In such a case, the variables $\mathbf{d}$ and $\mathbf{c}$ are almost surely constant. Beside the theoretical results, there exist tools that employ simulation for model checking (see, e.g., [26,11]).

In addition, GSMP are a proper subset of stochastic automata, a model of concurrent systems (see, e.g., [12]). Further, as shown in [16], GSMP have the same modeling power as stochastic Petri nets [22]. The formalism of deterministic and stochastic Petri nets (DSPN) introduced by [21] adds deterministic transitions – a counterpart of fixed-delay events. The authors restricted the model to at most one deterministic transition enabled at a time and to exponentially distributed timed transitions. For this restricted model, the authors proved existence of a steady state distribution and provided an algorithm for its computation. However, the methods inherently rely on the properties of the exponential distribution and cannot be extended to our setting with general variable delays. DSPN have been extended by [13,19] to allow arbitrarily many deterministic transitions. The authors provide algorithms for steady-state analysis of DSPN that were implemented in the tool DSPNExpress [18], but do not discuss under which conditions the steady-state distributions exist.

2 Preliminaries

In this paper, the sets of all positive integers, non-negative integers, real numbers, positive real numbers, and non-negative real numbers are denoted by $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{R}$, $\mathbb{R}_{>0}$, and $\mathbb{R}_{\geq 0}$, respectively. For a real number $r \in \mathbb{R}$, $\mathrm{int}(r)$ denotes its integral part, i.e. the largest integer smaller than r, and $\mathrm{frac}(r)$ denotes its fractional part, i.e. $r - \mathrm{int}(r)$. Let A be a finite or countably infinite set. A *probability distribution* on A is a function $f : A \to \mathbb{R}_{\geq 0}$ such that $\sum_{a \in A} f(a) = 1$. The set of all distributions on A is denoted by $\mathcal{D}(A)$.

A *σ-field* over a set Ω is a set $\mathcal{F} \subseteq 2^{\Omega}$ that includes Ω and is closed under complement and countable union. A *measurable space* is a pair $(\Omega, \mathcal{F})$ where Ω is a set called *sample space* and $\mathcal{F}$ is a σ-field over Ω whose elements are called *measurable sets*. Given a measurable space $(\Omega, \mathcal{F})$, we say that a function $f : \Omega \to \mathbb{R}$ is a random variable if the inverse image of any real interval is a measurable set. A *probability measure* over a measurable space $(\Omega, \mathcal{F})$ is a function $\mathcal{P} : \mathcal{F} \to \mathbb{R}_{\geq 0}$ such that, for each countable collection $\{X_i\}_{i \in I}$ of pairwise disjoint elements of $\mathcal{F}$, we have $\mathcal{P}(\bigcup_{i \in I} X_i) = \sum_{i \in I} \mathcal{P}(X_i)$ and, moreover, $\mathcal{P}(\Omega) = 1$. A *probability space* is a triple $(\Omega, \mathcal{F}, \mathcal{P})$, where $(\Omega, \mathcal{F})$ is a measurable space and $\mathcal{P}$ is a probability measure over $(\Omega, \mathcal{F})$. We say that a property $A \subseteq \Omega$ holds for *almost all* elements of a measurable set Y if $\mathcal{P}(Y) > 0$, $A \cap Y \in \mathcal{F}$, and $\mathcal{P}(A \cap Y \mid Y) = 1$. Alternatively, we say that A holds *almost surely* for Y.

2.1 Generalized Semi-Markov Processes

Let $\mathcal{E}$ be a finite set of *events*. To every $e \in \mathcal{E}$ we associate the *lower bound* $\ell_e \in \mathbb{N}_0$ and the *upper bound* $u_e \in \mathbb{N} \cup \{\infty\}$ of its delay. We say that e is a *fixed-delay* event if $\ell_e = u_e$, and a *variable-delay* event if $\ell_e < u_e$. Furthermore, we say that a variable-delay event e is *bounded* if $u_e \neq \infty$, and *unbounded*, otherwise. To each variable-delay event e we assign a *density function* $f_e : \mathbb{R} \to \mathbb{R}$ such that $\int_{\ell_e}^{u_e} f_e(x)\,dx = 1$. We assume f_e to be positive and continuous on the whole $[\ell_e, u_e]$ or $[\ell_e, \infty)$ if e is bounded or unbounded, respectively, and zero elsewhere. We require that f_e have finite expected value, i.e. $\int_{\ell_e}^{u_e} x \cdot f_e(x)\,dx < \infty$.

Definition 1. *A* generalized semi-Markov process *is a tuple* $(S, \mathcal{E}, \mathbf{E}, \mathrm{Succ}, \alpha_0)$ *where*

- S *is a finite set of* states*,*
- $\mathcal{E}$ *is a finite set of* events*,*
- $\mathbf{E} : S \to 2^{\mathcal{E}}$ *assigns to each state s a set of events* $\mathbf{E}(s) \neq \emptyset$ scheduled *to occur in s,*
- $\mathrm{Succ} : S \times 2^{\mathcal{E}} \to \mathcal{D}(S)$ *is the* successor *function, i.e. assigns a probability distribution specifying the successor state to each state and set of events that occur simultaneously in this state, and*
- $\alpha_0 \in \mathcal{D}(S)$ *is the* initial distribution.

A *configuration* is a pair (s, ν) where $s \in S$ and ν is a *valuation* which assigns to every event $e \in \mathbf{E}(s)$ the amount of time that elapsed since the event e was scheduled.[1] For convenience, we define $\nu(e) = \perp$ whenever $e \notin \mathbf{E}(s)$, and we denote by $\nu(\triangle)$ the amount of time spent in the previous configuration (initially, we put $\nu(\triangle) = 0$). When a set of events E occurs and the process moves from s to a state s', the valuation of *old* events of $\mathbf{E}(s) \setminus \mathbf{E}(s')$ is discarded to $\perp$, the valuation of each *inherited* event of $(\mathbf{E}(s') \cap \mathbf{E}(s)) \setminus E$ is increased by the time spent in s, and the valuation of each *new* event of $(\mathbf{E}(s') \setminus \mathbf{E}(s)) \cup (\mathbf{E}(s') \cap E)$ is set to 0.

We illustrate the dynamics of GSMP on the example of Figure 1. Let the bounds of the fixed-delay events *roundtrip_d*, *polling_d*, and *stable_d* be 1,

[1] Usually, the valuation is defined to store the time left before the event appears. However, our definition is equivalent and more convenient for the general setting where both bounded and unbounded events appear.

90, and 100, respectively. We start in the state *Idle*, i.e. in the configuration $(Idle, ((polling\_d, 0), (stable\_d, 0), (\vartriangle, 0)))$ denoting that $\nu(polling\_d) = 0$, $\nu(stable\_d) = 0$, $\nu(\vartriangle) = 0$, and $\perp$ is assigned to all other events. After 90 time units, the event *polling_d* occurs and we move to $(Init, ((query, 0), (stable\_d, 90), (\vartriangle, 90)))$. Assume that the event *query* occurs in the state *Init* after 0.6 time units and we move to $(Q\text{-}sent, ((response, 0), (roundtrip\_d, 0), (stable\_d, 90.6), (\vartriangle, 0.6)))$ and so forth.

A formal semantics of GSMP is usually defined in terms of general state-space Markov chains (GSSMC, see, e.g., [24]). A GSSMC is a stochastic process Φ over a measurable state-space $(\Gamma, \mathcal{G})$ whose dynamics is determined by an initial measure μ on $(\Gamma, \mathcal{G})$ and a *transition kernel* P which specifies one-step transition probabilities.[2] A given GSMP induces a GSSMC whose state-space consists of all configurations, the initial measure μ is induced by α_0 in a natural way, and the transition kernel is determined by the dynamics of GSMP described above. Formally,

- Γ is the set of all configurations, and $\mathcal{G}$ is a σ-field over Γ induced by the discrete topology over S and the Borel σ-field over the set of all valuations;
- the initial measure μ allows to start in configurations with zero valuation only, i.e. for $A \in \mathcal{G}$ we have $\mu(A) = \sum_{s \in Zero(A)} \alpha_0(s)$ where $Zero(A) = \{s \in S \mid (s, \mathbf{0}) \in A\}$;
- the transition kernel $P(z, A)$ describing the probability to move in one step from a configuration $z = (s, \nu)$ to any configuration in a set A is defined as follows. It suffices to consider A of the form $\{s'\} \times X$ where X is a measurable set of valuations. Let V and F be the sets of variable-delay and fixed-delay events, respectively, that are scheduled in s. Let $F' \subseteq F$ be the set of fixed-delay events that can occur as first among the fixed-delay event enabled in z, i.e. that have in ν the minimal remaining time u. Note that two variable-delay events occur simultaneously with probability zero. Hence, we consider all combinations of $e \in V$ and $t \in \mathbb{R}_{\geq 0}$ stating that

$$P(z, A) = \begin{cases} \sum_{e \in V} \int_0^\infty \mathrm{Hit}(\{e\}, t) \cdot \mathrm{Win}(\{e\}, t)\, dt & \text{if } F = \emptyset \\ \sum_{e \in V} \int_0^u \mathrm{Hit}(\{e\}, t) \cdot \mathrm{Win}(\{e\}, t)\, dt + \mathrm{Hit}(F', u) \cdot \mathrm{Win}(F', u) & \text{otherwise,} \end{cases}$$

where the term $\mathrm{Hit}(E, t)$ denotes the conditional probability of hitting A under the condition that E occurs at time t and the term $\mathrm{Win}(E, t)$ denotes the probability (density) of E occurring at time t. Formally,

$$\mathrm{Hit}(E, t) = \mathrm{Succ}(s, E)(s') \cdot \mathbf{1}[\nu' \in X]$$

where $\mathbf{1}[\nu' \in X]$ is the indicator function and ν' is the valuation after the transition, i.e. $\nu'(e)$ is $\perp$, or $\nu(e) + t$, or 0 for each old, or inherited, or new event e, respectively; and $\nu'(\vartriangle) = t$. The most complicated part is the definition of $\mathrm{Win}(E, t)$ which intuitively corresponds to the probability that E is the set of events "winning" the competition among the events scheduled in s at time t. First, we define a "shifted" density function $f_{e|\nu(e)}$ that takes into account that the time $\nu(e)$ has already elapsed. Formally, for a variable-delay event e and any elapsed time $\nu(e) < u_e$, we define

$$f_{e|\nu(e)}(x) = \frac{f_e(x + \nu(e))}{\int_{\nu(e)}^\infty f_e(y)\, dy} \qquad \text{if } x \geq 0.$$

[2] Precisely, transition kernel is a function $P : \Gamma \times \mathcal{G} \to [0, 1]$ such that $P(z, \cdot)$ is a probability measure over $(\Gamma, \mathcal{G})$ for each $z \in \Gamma$; and $P(\cdot, A)$ is a measurable function for each $A \in \mathcal{G}$.

Otherwise, we define $f_{e|\nu(e)}(x) = 0$. The denominator scales the function so that $f_{e|\nu(e)}$ is again a density function. Finally,

$$\mathrm{Win}(E,t) = \begin{cases} f_{e|\nu(e)}(t) \cdot \prod_{c \in V \setminus E} \int_t^{\infty} f_{c|\nu(c)}(y)\, dy & \text{if } E = \{e\} \subseteq V \\ \prod_{c \in V} \int_t^{\infty} f_{c|\nu(c)}(y)\, dy & \text{if } E = F' \subseteq F \\ 0 & \text{otherwise.} \end{cases}$$

A *run* of the Markov chain is an infinite sequence $\sigma = z_0\ z_1\ z_2 \cdots$ of configurations. The Markov chain is defined on the probability space $(\Omega, \mathcal{F}, \mathcal{P})$ where Ω is the set of all runs, $\mathcal{F}$ is the product σ-field $\bigotimes_{i=0}^{\infty} \mathcal{G}$, and $\mathcal{P}$ is the unique probability measure such that for every finite sequence $A_0, \cdots, A_n \in \mathcal{G}$ we have that

$$\mathcal{P}(\Phi_0 \in A_0, \cdots, \Phi_n \in A_n) = \int\limits_{z_0 \in A_0} \cdots \int\limits_{z_{n-1} \in A_{n-1}} \mu(dz_0) \cdot P(z_0, dz_1) \cdots P(z_{n-1}, A_n)$$

where each Φ_i is the i-th projection of an element in Ω (the i-th configuration of a run).

Finally, we define an m-step transition kernel P^m inductively as $P^1(z,A) = P(z,A)$ and $P^{i+1}(z,A) = \int_{\Gamma} P(z,dy) \cdot P^i(y,A)$.

2.2 Frequency Measures

Our attention focuses on frequencies of a fixed state $\mathring{s} \in S$ in the runs of the Markov chain. Let $\sigma = (s_0, \nu_0)\ (s_1, \nu_1) \cdots$ be a run. We define

$$\mathbf{d}(\sigma) = \lim_{n \to \infty} \frac{\sum_{i=0}^{n} \delta(s_i)}{n} \qquad\qquad \mathbf{c}(\sigma) = \lim_{n \to \infty} \frac{\sum_{i=0}^{n} \delta(s_i) \cdot \nu_{i+1}(\triangle)}{\sum_{i=0}^{n} \nu_{i+1}(\triangle)}$$

where $\delta(s_i)$ is equal to 1 when $s_i = \mathring{s}$, and 0 otherwise. We recall that $\nu_{i+1}(\triangle)$ is the time spent in state s_i before moving to s_{i+1}. We say that the random variable $\mathbf{d}$ or $\mathbf{c}$ is *well-defined* for a run σ if the corresponding limit exists for σ. Then, $\mathbf{d}$ corresponds to the frequency of discrete visits to the state $\mathring{s}$ and $\mathbf{c}$ corresponds to the ratio of time spent in the state $\mathring{s}$.

2.3 Region Graph

In order to state the results in a simpler way, we introduce the *region graph*, a standard notion from the area of timed automata [5]. It is a finite partition of the uncountable set of configurations. First, we define the region relation $\sim$. For $a, b \in \mathbb{R}$, we say that a and b *agree on integral part* if $\mathrm{int}(a) = \mathrm{int}(b)$ and neither or both a, b are integers. Further, we set the bound $B = \max(\{\ell_e, u_e \mid e \in \mathcal{E}\} \setminus \{\infty\})$. Finally, we put $(s_1, \nu_1) \sim (s_2, \nu_2)$ if

- $s_1 = s_2$;
- for all $e \in \mathbf{E}(s_1)$ we have that $\nu_1(e)$ and $\nu_2(e)$ agree on integral parts or are both greater than B;
- for all $e, f \in \mathbf{E}(s_1)$ with $\nu_1(e) \leq B$ and $\nu_1(f) \leq B$ we have that $\mathrm{frac}(\nu_1(e)) \leq \mathrm{frac}(\nu_1(f))$ iff $\mathrm{frac}(\nu_2(e)) \leq \mathrm{frac}(\nu_2(f))$.

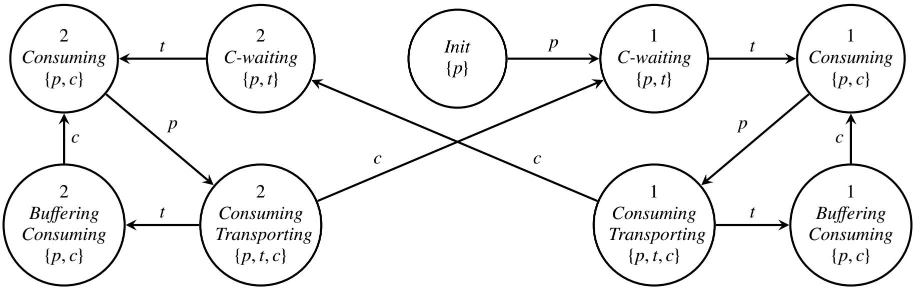


Fig. 2. A GSMP of a producer-consumer system. The events p, t, and c model that a packet production, transport, and consumption is finished, respectively. Below each state label, there is the set of scheduled events. The fixed-delay events p and c have $l_p = u_p = l_c = u_c = 1$ and the uniformly distributed variable-delay event t has $l_t = 0$ and $u_t = 1$.

Note that $\sim$ is an equivalence with finite index. The equivalence classes of $\sim$ are called *regions*. We define a finite *region graph* $G = (V, E)$ where the set of vertices V is the set of regions and for every pair of regions R, R' there is an edge $(R, R') \in E$ iff $P(z, R') > 0$ for some $z \in R$. The construction is correct because all states in the same region have the same one-step qualitative behavior (for details, see [10]).

3 Two Fixed-Delay Events

Now, we explain in more detail what problems can be caused by fixed-delay events. We start with an example of a GSMP with two fixed-delay events for which it is not true that the variables **d** and **c** are well-defined for almost all runs. Then we show some other examples of GSMP with fixed-delay events that disprove some results from literature. In the next section, we provide positive results when the number and type of fixed-delay events are limited.

When the Frequencies d and c are Not Well-Defined

In Figure 2, we show an example of a GSMP with two fixed-delay events and one variable-delay event for which it is not true that the variables **d** and **c** are well-defined for almost all runs. It models the following producer-consumer system. We use three components – a producer, a transporter and a consumer of packets. The components work in parallel but each component can process (i.e. produce, transport, or consume) at most one packet at a time.

Consider the following time requirements: each packet production takes *exactly* 1 time unit, each transport takes *at most* 1 time unit, and each consumption takes again *exactly* 1 time unit. As there are no limitations to block the producer, it is working for all the time and new packets are produced precisely each time unit. As the transport takes shorter time than the production, every new packet is immediately taken by the transporter and no buffer is needed at this place. When a packet arrives to the consumer, the

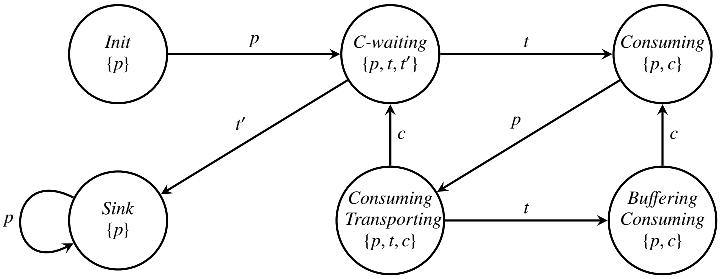


Fig. 3. A GSMP with two fixed-delay events p and c (with $l_p = u_p = l_c = u_c = 1$), a uniformly distributed variable-delay events t, t' (with $l_t = l_{t'} = 0$ and $u_t = u_{t'} = 1$)

consumption is started immediately if the consumer is waiting; otherwise, the packet is stored into a buffer. When the consumption is finished and the buffer is empty, the consumer waits; otherwise, a new consumption starts immediately.

In the GSMP in Figure 2, the consumer has two modules – one is in operation and the other idles at a time – when the consumer enters the waiting state, it switches the modules. The labels 1 and 2 denote which module of the consumer is in operation.

One can easily observe that the consumer enters the waiting state (and switches the modules) if and only if the current transport takes more time than it has ever taken. As the transport time is bounded by 1, it gets harder and harder to break the record. As a result, the system stays in the current module on average for longer time than in the previous module. Therefore, due to the successively prolonging stays in the modules, the frequencies for 1-states and 2-states oscillate. For precise computations, see [10]. We conclude the above observation by the following theorem.

Theorem 1. *There is a GSMP (with two fixed-delay events and one variable-delay event) for which it is* not *true that the variables* **c** *and* **d** *are almost surely well-defined.*

Counterexamples

In [3,4] there are algorithms for GSMP model checking based on the region construction. They rely on two crucial statements of the papers:

1. Almost all runs end in some of the bottom strongly connected components (BSCC) of the region graph.
2. Almost all runs entering a BSCC visit all regions of the component infinitely often.

Both of these statements are true for finite state Markov chains. In the following, we show that neither of them has to be valid for region graphs of GSMP.

Let us consider the GSMP depicted in Figure 3. This is a producer-consumer model similar to the previous example but we have only one module of the consumer here. Again, entering the state *C-waiting* indicates that the current transport takes more time than it has ever taken. In the state *C-waiting*, an additional event t' can occur and move the system into a state *Sink*. One can intuitively observe that we enter the state *C-waiting* less and less often and stay there for shorter and shorter time. Hence, the probability that the event t' occurs in the state *C-waiting* is decreasing during the run. For precise computations proving the following claim, see [10].

Claim. The probability to reach *Sink* from *Init* is strictly less than 1.

The above claim directly implies the following theorem thus disproving statement 1.

Theorem 2. *There is a GSMP (with two fixed-delay and two variable delay events) where the probability to reach any BSCC of the region graph is strictly smaller than 1.*

Now consider in Figure 3 a transition under the event p from the state *Sink* to the state *Init* instead of the self-loop. This turns the whole region graph into a single BSCC. We prove that the state *Sink* is almost surely visited only finitely often. Indeed, let $p < 1$ be the original probability to reach *Sink* guaranteed by the claim above. The probability to reach *Sink* from *Sink* again is also p as the only transition leading from *Sink* enters the initial configuration. Therefore, the probability to reach *Sink* infinitely often is $\lim_{n\to\infty} p^n = 0$. This proves the following theorem. Hence, the statement 2 of [3,4] is disproved, as well.

Theorem 3. *There is a GSMP (with two fixed-delay and two variable delay events) with strongly connected region graph and with a region that is reached infinitely often with probability* 0*.*

4 Single-Ticking GSMP

First of all, motivated by the previous counterexamples, we identify the behavior of the fixed-delay events that may cause **d** and **c** to be undefined. The problem lies in fixed-delay events that can immediately schedule themselves whenever they occur; such an event can occur periodically like ticking of clocks. In the example of Figure 3, there are two such events p and c. The phase difference of their ticking gets smaller and smaller, causing the unstable behavior.

For two fixed-delay events e and e', we say that e *causes* e' if there are states s, s' and a set of events E such that $\mathrm{Succ}(s, E)(s') > 0$, $e \in E$, and e' is newly scheduled in s'.

Definition 2. *A GSMP is called* single-ticking *if either there is no fixed-delay event or there is a strict total order $<$ on fixed-delay events with the least element e (called* ticking *event) such that whenever f causes g then either $f < g$ or $f = g = e$.*

From now on we restrict to single-ticking GSMP and prove our main positive result.

Theorem 4. *In single-ticking GSMP, the random variables* **d** *and* **c** *are well-defined for almost every run and admit only finitely many values. Precisely, almost every run reaches a BSCC of the region graph and for each BSCC B there are values $d, c \in [0, 1]$ such that* $\mathbf{d}(\sigma) = d$ *and* $\mathbf{c}(\sigma) = c$ *for almost all runs σ that reach the BSCC B.*

The rest of this section is devoted to the proof of Theorem 4. First, we show that almost all runs end up trapped in some BSCC of the region graph. Second, we solve the problem while restricting to runs that *start* in a BSCC (as the initial part of a run outside of any BSCC is not relevant for the long run average behavior). We show that in a BSCC, the variables **d** and **c** are almost surely constant. The second part of the proof relies on several standard results from the theory of general state space Markov chains. Formally, the proof follows from Propositions 1 and 2 stated below.

4.1 Reaching a BSCC

Proposition 1. *In single-ticking GSMP, almost every run reaches a BSCC of the region graph.*

The proof uses similar methods as the proof in [4]. By definition, the process moves along the edges of the region graph. From every region, there is a minimal path through the region graph into a BSCC, let n be the maximal length of all such paths. Hence, in at most n steps the process reaches a BSCC with positive probability from any configuration. Observe that if this probability was bounded from below, we would eventually reach a BSCC from any configuration almost surely. However, this probability can be arbitrarily small. Consider the following example with event e uniform on $[0, 1]$ and event f uniform on $[2, 3]$. In an intuitive notation, let R be the region $[0 < e < f < 1]$. What is the probability that the event e occurs after the elapsed time of f reaches 1 (i.e. that the region $[e = 0; 1 < f < 2]$ is reached)? For a configuration in R with valuation $((e, 0.2), (f, 0.7))$ the probability is 0.5 but for another configuration in R with $((e, 0.2), (f, 0.21))$ it is only 0.01. Notice that the transition probabilities depend on the difference of the fractional values of the clocks, we call this difference *separation*. Observe that in other situations, the separation of clocks from value 0 also matters.

Definition 3. *Let $\delta > 0$. We say that a configuration (s, ν) is δ*-separated *if for every $x, y \in \{0\} \cup \{\nu(e) \mid e \in \mathbf{E}(s)\}$, we have either* $|\text{frac}(x) - \text{frac}(y)| > \delta$ *or* $\text{frac}(x) = \text{frac}(y)$.

We fix a $\delta > 0$. To finish the proof using the concept of δ-separation, we need two observations. First, from *any* configuration we reach in m steps a δ-separated configuration with probability at least $q > 0$. Second, the probability to reach a fixed region from *any* δ-separated configuration is bounded from below by some $p > 0$. By repeating the two observations ad infinitum, we reach some BSCC almost surely. Let us state the claims. For proofs, see [10].

Lemma 1. *There is $\delta > 0$, $m \in \mathbb{N}$ and $q > 0$ such that from every configuration we reach a δ-separated configuration in m steps with probability at least q.*

Lemma 2. *For every $\delta > 0$ and $k \in \mathbb{N}$ there is $p > 0$ such that for any pair of regions R, R' connected by a path of length k and for any δ-separated $z \in R$, we have $P^k(z, R') > p$.*

Lemma 2 holds even for unrestricted GSMP. Notice that Lemma 1 does not. As in the example of Figure 3, the separation may be non-increasing for all runs.

4.2 Frequency in a BSCC

From now on, we deal with the bottom strongly connected components that are reached almost surely. Hence, we assume that the region graph G is strongly connected. We have to allow an arbitrary initial configuration $z_0 = (s, \nu)$; in particular, ν does not have to be a zero vector.[3]

Proposition 2. *In a single-ticking GSMP with strongly connected region graph, there are values $d, c \in [0, 1]$ such that for any initial configuration z_0 and for almost all runs σ starting from z_0, we have that $\mathbf{d}$ and $\mathbf{c}$ are well-defined and $\mathbf{d}(\sigma) = d$ and $\mathbf{c}(\sigma) = c$.*

We assume that the region graph is aperiodic in the following sense. A *period p* of a graph G is the greatest common divisor of lengths of all cycles in G. The graph G is *aperiodic* if $p = 1$. Under this assumption[4], the chain Φ is in some sense stable. Namely, (i) Φ has a unique invariant measure that is independent of the initial measure and (ii) the strong law of large numbers (SLLN) holds for Φ.

First, we show that (i) and (ii) imply the proposition. Let us recall the notions. We say that a probability measure π on $(\Gamma, \mathcal{G})$ is *invariant* if for all $A \in \mathcal{G}$

$$\pi(A) \quad = \quad \int_{\Gamma} \pi(dx) P(x, A).$$

The SLLN states that if $h : \Gamma \to \mathbb{R}$ satisfies $E_\pi[h] < \infty$, then almost surely

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{n} h(\Phi_i)}{n} \quad = \quad E_\pi[h], \tag{1}$$

where $E_\pi[h]$ is the expected value of h according to the invariant measure π.

We set h as follows. For a run $(s_0, \nu_0)(s_1, \nu_1)\cdots$, let $h(\Phi_i) = 1$ if $s_i = \mathring{s}$ and 0, otherwise. We have $E_\pi[h] < \infty$ since $h \leq 1$. From (1) we obtain that almost surely

$$\mathbf{d} \quad = \quad \lim_{n\to\infty} \frac{\sum_{i=1}^{n} h(\Phi_i)}{n} \quad = \quad E_\pi[h].$$

As a result, $\mathbf{d}$ is well-defined and equals the constant value $E_\pi[h]$ for almost all runs. We treat the variable $\mathbf{c}$ similarly. Let $W((s, \nu))$ denote the expected waiting time of the GSMP in the configuration (s, ν). We use a function $\tau((s, \nu)) = W((s, \nu))$ if $s = \mathring{s}$ and 0, otherwise. Since all the events have finite expectation, the functions W and τ are bounded and we have $E_\pi[W] < \infty$ and $E_\pi[\tau] < \infty$. We show in [10] that almost surely

$$\mathbf{c} \quad = \quad \lim_{n\to\infty} \frac{\sum_{i=1}^{n} \tau(\Phi_i)}{\sum_{i=1}^{n} W(\Phi_i)} \quad = \quad \frac{E_\pi[\tau]}{E_\pi[W]}.$$

Therefore, $\mathbf{c}$ is well-defined and equals the constant $E_\pi[\tau]/E_\pi[W]$ for almost all runs.

Second, we prove (i) and (ii). A standard technique of general state space Markov chains (see, e.g., [24]) yields (i) and (ii) for chains that satisfy the following condition. Roughly speaking, we search for a set of configurations C that is visited infinitely often and for some ℓ the measures $P^\ell(x, \cdot)$ and $P^\ell(y, \cdot)$ are very similar for any $x, y \in C$. This is formalized by the following lemma.

[3] Technically, the initial measure is $\mu(A) = 1$ if $z_0 \in A$ and $\mu(A) = 0$, otherwise.

[4] If the region graph has period $p > 1$, we can employ the standard technique and decompose the region graph (and the Markov chain) into p aperiodic components. The results for individual components yield straightforwardly the results for the whole Markov chain, see, e.g., [9].

Lemma 3. *There is a measurable set of configurations C such that*

1. *there is $k \in \mathbb{N}$ and $\alpha > 0$ such that for every $z \in \Gamma$ we have $P^k(z, C) \geq \alpha$, and*
2. *there is $\ell \in \mathbb{N}$, $\beta > 0$, and a probability measure κ such that for every $z \in C$ and $A \in \mathcal{G}$ we have $P^\ell(z, A) \geq \beta \cdot \kappa(A)$.*

Proof (Sketch). Let e be the ticking event and R some reachable region where e is the event closest to its upper bound. We fix a sufficiently small $\delta > 0$ and choose C to be the set of δ-separated configurations of R. We prove the first part of the lemma similarly to Lemmata 1 and 2. As regards the second part, we define the measure κ uniformly on a hypercube X of configurations (s, ν) that have $\nu(e) = 0$ and $\nu(f) \in (0, \delta)$, for $f \neq e$. First, assume that e is the only fixed-delay event. We fix $z = (s', \nu')$ in R; let $d = u_e - \nu'(e) > \delta$ be the time left in z before e occurs. For simplicity, we assume that each variable-delay events can occur after an arbitrary delay $x \in (d - \delta, d)$. Precisely, that it can occur in an ε-neighborhood of x with probability bounded from below by $\beta \cdot \varepsilon$ where β is the minimal density value of all $\mathcal{E}$. Note that the variable-delay events can be "placed" this way arbitrarily in $(0, \delta)$. Therefore, when e occurs, it has value 0 and all variable-delay events can be in interval $(0, \delta)$. In other words, we have $P^\ell(z, A) \geq \beta \cdot \kappa(A)$ for any measurable $A \subseteq X$ and for $\ell = |\mathcal{E}|$.

Allowing other fixed-delay events causes some trouble because a fixed-delay event $f \neq e$ cannot be "placed" arbitrarily. In the total order $<$, the event f can cause only strictly greater fixed-delay events. The greatest fixed-delay event can cause only variable-delay events that can be finally "placed" arbitrarily as described above. □

5 Approximations

In the previous section we have proved that in single-ticking GSMP, $\mathbf{d}$ and $\mathbf{c}$ are almost surely well-defined and for almost all runs they attain only finitely many values $d_1 \ldots, d_k$ and $c_1, \ldots, c_k$, respectively. In this section we show how to approximate d_i's and c_i's and the probabilities that $\mathbf{d}$ and $\mathbf{c}$ attain these values, respectively.

Theorem 5. *In a single-ticking GSMP, let $d_1, \ldots, d_k$ and $c_1, \ldots, c_k$ be the discrete and timed frequencies, respectively, corresponding to BSCCs of the region graph. For all $1 \leq i \leq k$, the numbers d_i and c_i as well as the probabilities $\mathcal{P}(\mathbf{d} = d_i)$ and $\mathcal{P}(\mathbf{c} = c_i)$ can be approximated up to any $\varepsilon > 0$.*

Proof. Let $X_1, \ldots, X_k$ denote the sets of configurations in individual BSCCs and d_i and c_i correspond to X_i. Since we reach a BSCC almost surely, we have

$$\mathcal{P}(\mathbf{d} = d_i) = \sum_{j=1}^{k} \mathcal{P}(\mathbf{d} = d_i \mid Reach(X_j)) \cdot \mathcal{P}(Reach(X_j)) = \sum_{j=1}^{k} \mathbf{1}[d_j = d_i] \cdot \mathcal{P}(Reach(X_j))$$

where the second equality follows from the fact that almost all runs in the j-th BSCC yield the discrete frequency d_j. Therefore, $\mathcal{P}(\mathbf{d} = d_i)$ and d_i can be approximated as follows using the methods of [25].

Claim. Let X be a set of all configurations in a BSCC $\mathcal{B}$, $X_{\mathring{s}} \subseteq X$ the set of configurations with state $\mathring{s}$, and d the frequency corresponding to $\mathcal{B}$. There are computable constants $n_1, n_2 \in \mathbb{N}$ and $p_1, p_2 > 0$ such that for every $i \in \mathbb{N}$ and $z_X \in X$ we have

$$\begin{aligned} |\mathcal{P}(\mathit{Reach}(X)) - P^i(z_0, X)| &\leq (1 - p_1)^{\lfloor i/n_1 \rfloor} \\ |d - P^i(z_X, X_{\mathring{s}})| &\leq (1 - p_2)^{\lfloor i/n_2 \rfloor} \end{aligned}$$

Further, we want to approximate $c_i = E_\pi[\tau]/E_\pi[W]$, where π is the invariant measure on X_i. In other words, we need to approximate $\int_{X_i} \tau(x)\pi(dx)$ and $\int_{X_i} W(x)\pi(dx)$. An n-th approximation $w_n(x)$ of $W(x)$ can be gained by discretizing the regions into, e.g., $1/n$-large hypercubes. If W is continuous, then $(w_n)_{n=1}^{\infty}$ is its pointwise approximation. Moreover, if W is bounded, then it is dominated by the approximation function w_n. Hence the approximation is correct by the dominated convergence theorem. Note that τ only differs from W in being identically zero on some regions. Therefore, the following claim concludes the proof. For details, see [10].

Claim. On each region, W is continuous and bounded and can be approximated.

6 Conclusions, Future Work

We have studied long run average properties of generalized semi-Markov processes with both fixed-delay and variable-delay events. We have shown that two or more (unrestricted) fixed-delay events lead to considerable complications regarding stability of GSMP. In particular, we have shown that the frequency of states of a GSMP may not be well-defined and that bottom strongly connected components of the region graph may not be reachable with probability one. This leads to counterexamples disproving several results from literature. On the other hand, for single-ticking GSMP we have proved that the frequencies of states are well-defined for almost all runs. Moreover, we have shown that almost every run has one of finitely many possible frequencies that can be effectively approximated (together with their probabilities) up to a given error tolerance.

In addition, the frequency measures can be easily extended into the mean payoff setting. Consider assigning real rewards to states. The mean payoff then corresponds to the frequency weighted by the rewards.

Concerning future work, the main issue is efficiency of algorithms for computing performance measures for GSMP. We plan to work on both better analytical methods as well as practicable approaches to Monte Carlo simulation. One may also consider extensions of our positive results to controlled GSMP and games on GSMP.

References

1. de Alfaro, L.: How to specify and verify the long-run average behavior of probabilistic systems. In: Proceedings of LICS 1998, pp. 454–465. IEEE Computer Society Press, Los Alamitos (1998)
2. Alur, R., Bernadsky, M.: Bounded model checking for GSMP models of stochastic real-time systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 19–33. Springer, Heidelberg (2006)

3. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for probabilistic real-time systems. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 115–136. Springer, Heidelberg (1991)
4. Alur, R., Courcoubetis, C., Dill, D.: Verifying automata specifications of probabilistic real-time systems. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, Springer, Heidelberg (1992)
5. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
6. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Transaction on Software Engineering 29(6), 524–541 (2003)
7. Barbot, B., Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Efficient CTMC model checking of linear real-time objectives. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 128–142. Springer, Heidelberg (2011)
8. Bernadsky, M., Alur, R.: Symbolic analysis for GSMP models with one stateful clock. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 90–103. Springer, Heidelberg (2007)
9. Brázdil, T., Krčál, J., Křetínský, J., Kučera, A., Řehák, V.: Measuring performance of continuous-time stochastic processes using timed automata. In: Proceedings of 14th International Conference on Hybrid Systems: Computation and Control (HSCC 2011), pp. 33–42, ACM Press, New York (2011)
10. Brázdil, T.: Krčál, J., Křetínský, J., Řehák, V.: Fixed-delay Events in Generalized Semi-Markov Processes Revisited. ArXiv e-prints (September 2011)
11. Ciardo, G., Jones III, R., Miner, A., Siminiceanu, R.: Logic and stochastic modeling with SMART. Performance Evaluation 63(6), 578–608 (2006)
12. D'Argenio, P., Katoen, J.: A theory of stochastic systems Part I: Stochastic automata. Information and Computation 203(1), 1–38 (2005)
13. German, R., Lindemann, C.: Analysis of stochastic Petri nets by the method of supplementary variables. Performance Evaluation 20(1-3), 317–335 (1994)
14. Glynn, P.: A GSMP formalism for discrete event systems. Proceedings of the IEEE 77, 14–23 (1989)
15. Haas, P.: On simulation output analysis for generalized semi-markov processes. Commun. Statist. Stochastic Models 15, 53–80 (1999)
16. Haas, P.: Stochastic Petri Nets: Modelling, Stability, Simulation. Springer Series in Operations Research and Financial Engineering. Springer, Heidelberg (2010)
17. Haas, P., Shedler, G.: Regenerative generalized semi-Markov processes. Stochastic Models 3(3), 409–438 (1987)
18. Lindemann, C., Reuys, A., Thummler, A.: The DSPNexpress 2.000 performance and dependability modeling environment. In: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers, pp. 228–231. IEEE Computer Society Press, Los Alamitos (1999)
19. Lindemann, C., Shedler, G.: Numerical analysis of deterministic and stochastic Petri nets with concurrent deterministic transitions. Performance Evaluation 27, 565–582 (1996)
20. López, G., Hermanns, H., Katoen, J.: Beyond memoryless distributions: Model checking semi-markov chains. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 57–70. Springer, Heidelberg (2001)
21. Marsan, M., Chiola, G.: On Petri nets with deterministic and exponentially distributed firing times. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 132–145. Springer, Heidelberg (1987)
22. Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. Wiley, Chichester (1995)

23. Matthes, K.: Zur Theorie der Bedienungsprozesse. In: Transactions of the Third Prague Conference on Information Theory, Statistical Decision Functions, Random Processes, pp. 513–528 (1962)
24. Meyn, S., Tweedie, R.: Markov Chains and Stochastic Stability. Cambridge University Press, Cambridge (2009)
25. Roberts, G., Rosenthal, J.: General state space Markov chains and MCMC algorithms. Probability Surveys 1, 20–71 (2004)
26. Younes, H., Simmons, R.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002)

Semantic Analysis of Gossip Protocols for Wireless Sensor Networks

Ruggero Lanotte[1] and Massimo Merro[2]

[1] Dipartimento di Informatica e Comunicazione, Università dell'Insubria, Italy
[2] Dipartimento di Informatica, Università degli Studi di Verona, Italy

Abstract. Gossip protocols have been proposed as a robust and efficient method for disseminating information throughout large-scale networks. In this paper, we propose a compositional analysis technique to study formal probabilistic models of gossip protocols in the context of wireless sensor networks. We introduce a simple probabilistic timed process calculus for modelling wireless sensor networks. A simulation theory is developed to compare probabilistic protocols that have similar behaviour up to a certain probability. This theory is used to prove a number of algebraic laws which revealed to be very effective to evaluate the performances of gossip networks with and without communication collisions.

1 Introduction

Wireless sensor networks (WSNs) are (possibly large-scale) networks of sensor nodes deployed in strategic areas to gather data. Sensor nodes collaborate using wireless communications with an asymmetric many-to-one data transfer model. Typically, they send their sensed data to a sink node which collects the relevant information. WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature); they are used as embedded systems (e.g., biomedical sensor engineering, smart homes) or mobile applications (e.g., when attached to robots, soldiers, or vehicles). In wireless sensor networks, sensor nodes are usually battery-powered, and the energy expenditure of sensors has to be wisely managed by their architectures and protocols to prolong the overall network lifetime. Energy conservation is thus one of the major issues in sensor networks.

Flooding is a traditional robust algorithm that delivers data packets in a network from a source to a destination. In WSNs, each node that receives a message propagates it to all its neighbours by broadcast. This causes unnecessary retransmissions increasing the number of collisions, together depriving sensors of valuable battery power. Therefore, flooding algorithms may not be suitable in the context of dense networks like wireless sensor networks.

Gossipping [9] addresses some critical problems of flooding overhead. The goal of gossip protocols is to reduce the number of retransmissions by making some of the nodes discard the message instead of forwarding it. Gossip protocols exhibit both *nondeterministic* and *probabilistic* behaviour. Nondeterminism arises

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 156–170, 2011.

as they deal with distributed networks in which the activities of individual nodes occur nondeterministically. As to the probabilistic behaviour, nodes are required to forward packets with a pre-specified gossip probability p_{gsp}. When a node receives a message, rather than immediately retransmitting it as in flooding, it relies on the probability p_{gsp} to determine whether or not to retransmit. The main benefit is that when p_{gsp} is sufficiently large, the entire network receives the broadcast message with very high probability, even though only a nondeterministic subset of nodes has forwarded the message.

Most of the analyses of protocols for large-scale WSNs are usually based on discrete-event simulators (e.g., ns-2, Opnet and Glomosim). However, different simulators often support different models of the MAC physical-layer yielding different results, even for simple systems. In principle, as noticed in [2], owing to their often relatively simple structure, gossip protocols lend themselves very well to formal analysis, in order to predict their behaviour with high confidence. Formal analysis techniques are supported by (semi-)automated tools. For instance, *probabilistic model checking* [6,10] provides both an exhaustive search of all possible behaviours of the system, and exact, rather than approximate, quantitative results. Of course, model checking suffers from the so-called state explosion problem whereas simulation-based approaches are scalable to much larger systems, at the expense of exhaustiveness and numerical accuracy.

Contribution. In this paper, we propose a compositional analysis technique to study probabilistic models of gossip protocols in the context of WSNs. We introduce a simple probabilistic timed process calculus, called pTCWS, for modelling wireless sensor networks. We then develop a compositional simulation theory, denoted $\sqsubseteq_p$, to compare probabilistic protocols that have similar behaviour up to a certain probability p. Intuitively, we write $M \sqsubseteq_p N$ if M is simulated by N with a probability (at least) p. Compositionality is crucial when reasoning about large-scale protocols where all nodes run the same probabilistic (simple) code as in gossip protocols. For instance, it allows us to join and sometime merge the behaviour of different components of a network. In particular, for a gossip network $\mathrm{GSP}_{p_{\mathrm{gsp}}}$, which transmits with gossip probability p_{gsp}, we can estimate the probability p_{ok} to simulate a non-probabilistic network GSP_OK whose target nodes successfully receive the message:

$$\mathrm{GSP\_OK} \quad \sqsubseteq_{p_{\mathrm{ok}}} \quad \mathrm{GSP}_{p_{\mathrm{gsp}}} \ .$$

For this purpose, we prove and apply a number of algebraic laws, whose application can be *mechanised*, to evaluate the performances of gossip networks.

The paper uses the gossip protocol described above as baseline. That description, however, is incomplete. It does not specify, for instance, what happens in case of a collision, i.e. when a node receives two messages at the same time. We start our analysis by assuming no collision. Then, we study gossip protocols in the presence of *communication collision*, to determine its effect on the performance results.

In this paper proofs are sketched or omitted; full proofs can be found in [12].

Table 1 Syntax

Networks:		
M, N	$::= \mathbf{0}$	empty network
	$\mid M_1 \mid M_2$	parallel composition
	$\mid n[P]^{\nu}$	node
Processes:		
P, Q	$::= \mathsf{nil}$	stuck
	$\mid !\langle u \rangle.C$	broadcast
	$\mid \lfloor ?(x).C \rfloor D$	receiver with timeout
	$\mid \lfloor \tau.C \rfloor D$	internal with timeout
	$\mid \sigma.C$	sleep
	$\mid X$	process variable
	$\mid \mathsf{fix}\, X.P$	recursion
Probabilistic Choice:		
C, D	$::= \bigoplus_{i \in I} p_i{:}P_i$	

2 A Probabilistic Timed Process Calculus

In Table 1, we define the syntax of `pTCWS` in a two-level structure, a lower one for *processes* and an upper one for *networks*. We use letters $m, n, \ldots$ for logical names, x, y, z for *variables*, u for *values*, and v and w for *closed values*, i.e. values that do not contain variables.

A network in `pTCWS` is a (possibly empty) collection of nodes (which represent devices) running in parallel and using a unique common radio channel to communicate with each other. All nodes are assumed to have the same transmission range (this is a quite common assumption in models for ad hoc networks).The communication paradigm is *local broadcast*; only nodes located in the range of the transmitter may receive data. We write $n[P]^{\nu}$ for a node named n (the device network address) executing the sequential process P. The tag ν contains (the names of) the neighbours of n. Said in other words, ν contains all nodes laying in the transmission cell of n (except n). In this manner, we model the network topology.[1] Our wireless networks have a fixed topology as node mobility is not relevant to sensor networks. Moreover, nodes cannot be created or destroyed.

Processes are sequential and live within the nodes. The symbol nil denotes the stuck process. The sender process $!\langle v \rangle.C$ broadcasts the value v, the continuation being C. The process $\lfloor ?(x).C \rfloor D$ denotes a receiver with timeout. Intuitively, this process either receives a value v, in the current time interval, and then continues as C where the variable x is instantiated with v, or it idles for one time unit, and then continues as D. Similarly, the process $\lfloor \tau.C \rfloor D$ either performs an internal action, in the current time interval, or it idles for one time unit and then continues

[1] We could have represented the topology in terms of a restriction operator à la CCS on node names; we preferred our notation to keep at hand the neighbours of a node.

as D. The process $\sigma.C$ models sleeping for one time unit. In sub-terms of the form $\sigma.D$, $\lfloor\tau.C\rfloor D$ and $\lfloor ?(x).C\rfloor D$ the occurrence of D is said to be *time-guarded*. The process $\mathsf{fix}\, X.P$ denotes *time-guarded recursion*, as all occurrences of the process variable X may only occur time-guarded in P.

Remark 1. In the remainder of the paper, with an abuse of notation, we will write $?(x).C$ to denote a persistent listener, defined as $\mathsf{fix}\, X.\lfloor ?(x).C\rfloor X$. Similarly, we will write $\tau.C$ as an abbreviation for $\mathsf{fix}\, X.\lfloor\tau.C\rfloor X$.

The construct $\bigoplus_{i\in I} p_i{:}P_i$ denotes probabilistic choice, where I is an indexing finite set and $p_i \in (0,1]$ denotes the probability to execute the process P_i, with $\sum_{i\in I} p_i = 1$. In process $\lfloor ?(x).C\rfloor D$ the variable x is bound in C. Similarly, in process $\mathsf{fix}\, X.P$ the process variable X is bound in P. This gives rise to the standard notions of *free (process) variables* and *bound (process) variables* and *α-conversion*. We identify processes and networks up to α-conversion. A term is said to be *closed* if it does not contain free (process) variables. We always work with closed networks: The absence of free variables is trivially maintained at run-time. We write $\{^v/_x\}T$ for the substitution of the variable x with the value v in the term T. Similarly, we write $\{^P/_X\}T$ for the substitution of the process variable X with the process P in T.

We report some notational *conventions*. $\prod_{i\in I} M_i$ denotes the parallel composition of all M_i, for $i \in I$. We identify $\prod_{i\in I} M_i = \mathbf{0}$ if $I = \emptyset$. We write $P_1 \oplus_p P_2$ to denote the probabilistic process $p{:}P_1 \oplus (1{-}p){:}P_2$. We identify the probabilistic process $1{:}P$ with P. We write $!\langle v\rangle$ as an abbreviation for $!\langle v\rangle.1{:}\mathsf{nil}$. For $k > 0$ we write $\sigma^k.P$ as an abbreviation for $\sigma.\sigma.\ldots.\sigma.P$, where prefix σ appears k times.

Here are some definitions that will be useful in the remainder of the paper. Given a network M, $\mathsf{nds}(M)$ returns the names of M. If $m \in \mathsf{nds}(M)$, the function $\mathsf{ngh}(m, M)$ returns the set of the neighbours of m in M. Thus, for $M = M_1 \mid m[P]^\nu \mid M_2$ it holds that $\mathsf{ngh}(m, M) = \nu$. We write $\mathsf{ngh}(M)$ for $\bigcup_{m\in\mathsf{nds}(M)} \mathsf{ngh}(m, M)$.

Definition 1. Structural congruence *over* **pTCWS***, written* $\equiv$*, is defined as the smallest equivalence relation, preserved by parallel composition, which is a commutative monoid with respect to parallel composition and for which* $n[\mathsf{fix}\, X.P]^\nu \equiv n[P\{^{\mathsf{fix}\, X.P}/_X\}]^\nu$.

The syntax presented in Table 1 allows to derive networks which are somehow ill-formed. With the following definition we rule out networks containing two nodes with the same name. Moreover, as all nodes have the same transmission range, the neighbouring relation must be symmetric. Finally, in order to guarantee clock synchronisation, we impose network connectivity.

Definition 2 (Well-formedness). *M is said to be* well-formed *if*

- *whenever* $M \equiv M_1 \mid m_1[P_1]^{\nu_1} \mid m_2[P_2]^{\nu_2}$ *it holds that* $m_1 \neq m_2$*;*
- *whenever* $M \equiv N \mid m_1[P_1]^{\nu_1} \mid m_2[P_2]^{\nu_2}$ *with* $m_1 \in \nu_2$ *it holds that* $m_2 \in \nu_1$*;*
- *for all* $m, n \in \mathsf{nds}(M)$ *there are* $m_1, \ldots, m_k \in \mathsf{nds}(M)$*, such that* $m{=}m_1$*,* $n{=}m_k$*,* $\nu_j = \mathsf{ngh}(m_j, M)$*, for* $1{\le}j{\le}k$*, and* $m_i \in \nu_{i+1}$*, for* $1{\le}i{\le}k{-}1$.

Henceforth we will always work with well-formed networks.

2.1 Probabilistic Labelled Transition Semantics

Along the lines of [5,11], we propose an *operational semantics* for pTCWS associating with each network a graph-like structure representing its possible reactions: We use a generalisation of labelled transition system that includes probabilities.

Below, we report the mathematical machinery for doing that.

Definition 3 (Deng et al. [5]). *A (discrete)* probability sub-distribution *over a countable set S is a function $\Delta : S \to [0,1]$ such that $\sum_{s \in S} \Delta(s) \in (0..1]$. The* support *of a probability sub-distribution Δ is given by $\lceil \Delta \rceil = \{s \in S \mid \Delta(s) > 0\}$. We write $\mathcal{D}_{\text{sub}}(S)$, ranged over Δ, Θ, Φ, for the set of all probability sub-distributions over S with finite support. For any $s \in S$, the* point distribution at *s, written $\overline{s}$, assigns probability 1 to s and 0 to all others elements of S.*

If $p_i \geq 0$ and Δ_i is a sub-distribution for each i in some finite index set I, and $\sum_{i \in I} p_i \in (0,1]$, then the probability sub-distribution $\sum_{i \in I} p_i \cdot \Delta_i$ is given by

$$(\sum_{i \in I} p_i \cdot \Delta_i)(s) \overset{\text{def}}{=} \sum_{i \in I} p_i \cdot \Delta_i(s) \ .$$

We write a sub-distribution as $p_1 \cdot \Delta_1 + ... + p_n \cdot \Delta_n$, when the index set I is $\{1, \dots, n\}$. Sometimes, with an abuse of notation, in the previous decomposition, the terms Δ_i are not necessarily distinct (for instance $1 \cdot \Delta$ may be rewritten as $p \cdot \Delta + (1{-}p) \cdot \Delta$, for any $p \in [0..1]$). A probability sub-distribution $\Delta \in \mathcal{D}_{\text{sub}}(S)$ is said to be a *probability distribution* if $\sum_{s \in S} \Delta(s) = 1$. With $\mathcal{D}(S)$ we denote the set of all probability distributions over S with finite support.

Definition 1 and Definition 2 generalise to sub-distributions in $\mathcal{D}_{\text{sub}}(\texttt{pTCWS})$. Given two probability sub-distributions Δ and Θ, we write $\Delta \equiv \Theta$ whenever $\Delta([M]_\equiv) = \Theta([M]_\equiv)$ for all equivalence classes $[M]_\equiv \subseteq \texttt{pTCWS}$ of $\equiv$. Moreover, a probability sub-distribution $\Delta \in \mathcal{D}_{\text{sub}}(\texttt{pTCWS})$ is said to be well-formed if its support contains only well-formed networks.

We now give the probabilistic generalisation of labelled transition system:

Definition 4 (Deng et al. [5]). *A* probabilistic labelled transition system[2] *(pLTS) is a triple $\langle S, \mathcal{L}, \to \rangle$ where i) S is a set of states; ii) $\mathcal{L}$ is a set of transition labels; iii) $\to$ is a labelled transition relation contained in $S \times \mathcal{L} \times \mathcal{D}(S)$.*

The operational semantics of pTCWS is given by a particular pLTS $\langle \texttt{pTCWS}, \mathcal{L}, \to \rangle$, where $\mathcal{L} = \{m!v \triangleright \mu, m?v, \tau, \sigma\}$ contains the labels denoting broadcasting, reception, internal actions and time passing, respectively. As regards the labelled transition relation, we need to formalise the interpretation of nodes containing probabilistic processes as probability distributions.

Definition 5. *For any probabilistic choice $\bigoplus_{i \in I} p_i{:}P_i$ over a finite indexing set I, $[\![n[\bigoplus_{i \in I} p_i{:}P_i]^\nu]\!]$ denotes the probability distribution defined as follows:*

- *if $I {\neq} \emptyset$ then for any $M {\in} \texttt{pTCWS}$: $[\![n[\bigoplus_{i \in I} p_i{:}P_i]^\nu]\!](M) \overset{\text{def}}{=} \sum_{i \in I \,\wedge\, n[P_i]^\nu = M} p_i$*

[2] Essentially the same model has appeared in the literature under different names such as, for instance, *NP-systems* [8] or *simple probabilistic automata* [15].

Table 2 Probabilistic Labelled Transition System

$$\text{(Snd)}\ \frac{-}{m[!\langle v\rangle.C]^{\nu} \xrightarrow{m!v\triangleright\nu} \llbracket m[C]^{\nu}\rrbracket} \qquad \text{(Rcv)}\ \frac{m\in\nu}{n[\lfloor ?(x).C\rfloor D]^{\nu} \xrightarrow{m?v} \llbracket n[\{^{v}\!/_{x}\}C]^{\nu}\rrbracket}$$

$$\text{(Rcv-}\mathbf{0}\text{)}\ \frac{-}{\mathbf{0} \xrightarrow{m?v} \overline{\mathbf{0}}} \qquad \text{(RcvEnb)}\ \frac{\neg(m\in\nu\ \wedge\ \mathsf{rcv}(P))\ \wedge\ m\neq n}{n[P]^{\nu} \xrightarrow{m?v} \overline{n[P]^{\nu}}}$$

$$\text{(RcvPar)}\ \frac{M \xrightarrow{m?v} \Delta\ \ N \xrightarrow{m?v} \Theta}{M \mid N \xrightarrow{m?v} \Delta \mid \Theta} \qquad \text{(Bcast)}\ \frac{M \xrightarrow{m!v\triangleright\nu} \Delta\ \ N \xrightarrow{m?v} \Theta\ \ {\scriptstyle \mu:=\nu\setminus\mathsf{nds}(N)}}{M \mid N \xrightarrow{m!v\triangleright\mu} \Delta \mid \Theta}$$

$$\text{(Tau)}\ \frac{-}{m[\lfloor\tau.C\rfloor D]^{\nu} \xrightarrow{\tau} \llbracket m[C]^{\nu}\rrbracket} \qquad \text{(TauPar)}\ \frac{M \xrightarrow{\tau} \Delta}{M \mid N \xrightarrow{\tau} \Delta \mid \overline{N}}$$

$$(\sigma\text{-}\mathbf{0})\ \frac{-}{\mathbf{0} \xrightarrow{\sigma} \overline{\mathbf{0}}} \qquad (\sigma\text{-nil})\ \frac{-}{n[\mathsf{nil}]^{\nu} \xrightarrow{\sigma} \overline{n[\mathsf{nil}]^{\nu}}}$$

$$\text{(Timeout)}\ \frac{-}{n[\lfloor\ldots\rfloor D]^{\nu} \xrightarrow{\sigma} \llbracket n[D]^{\nu}\rrbracket} \qquad \text{(Sleep)}\ \frac{-}{n[\sigma.C]^{\nu} \xrightarrow{\sigma} \llbracket n[C]^{\nu}\rrbracket}$$

$$(\sigma\text{-Par})\ \frac{M \xrightarrow{\sigma} \Delta\quad N \xrightarrow{\sigma} \Theta}{M \mid N \xrightarrow{\sigma} \Delta \mid \Theta} \qquad \text{(Rec)}\ \frac{n[\{^{\mathsf{fix}\,X.P}\!/_{X}\}P]^{\nu} \xrightarrow{\lambda} \Delta}{n[\mathsf{fix}\,X.P]^{\nu} \xrightarrow{\lambda} \Delta}$$

– *if* $I=\emptyset$ *then* $\llbracket n[\bigoplus_{i\in I} p_i{:}P_i]^{\nu}\rrbracket \stackrel{\text{def}}{=} \overline{n[\mathsf{nil}]^{\nu}}$.

The definition of the relations $\xrightarrow{\lambda}$, for $\lambda\in\mathcal{L}$, is given in Table 2. Some of these rules use an obvious notation for distributing parallel composition over a (sub-)distribution:

$$(\Delta \mid \Theta)(M) \stackrel{\text{def}}{=} \begin{cases} \Delta(M_1)\cdot\Theta(M_2) & \text{if } M = M_1 \mid M_2 \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

In rule (Snd) a sender m broadcasts a message v to its neighbours ν, and then continues as C. In the label $m!v\triangleright\nu$ the set ν contains the neighbours of m which may receive the message v. In rule (Rcv) a receiver gets a message v from a neighbour node m, and then evolves as $\{^{v}\!/_{x}\}C$. If no message is received in the current time interval the node n will continue with process D, as specified in rule (Timeout). Rules (Rcv-0) and (RcvEnb) serve to model reception enabling for synchronisation purposes. For instance, rule (RcvEnb) regards nodes which are not involved in transmissions originating from m. This may happen either because the two nodes are out of range (i.e. $m\notin\nu$) or because n is not willing to receive ($\mathsf{rcv}(P)$ is a boolean predicate that returns true if $n[P]^{\nu}\equiv n[\lfloor ?(x).C\rfloor D]^{\nu}$, for some x, C, D). In both cases, node n is not affected by the transmission. In rule (RcvPar) we model the composition of two networks receiving the same message from the same transmitter. Rule (Bcast) models the propagation of messages on the broadcast channel. Note that we loose track of those transmitter's neighbours that are in N. Rule (Tau) models internal computations in a single node. Rule (TauPar) propagates internal computations on parallel components. Rules (σ-nil) and (σ-**0**) are straightforward as both terms **0** and $n[\mathsf{nil}]^{\nu}$ do not prevent time-passing. Rule (Sleep) models sleeping for one time unit. Rule (σ-Par) models time synchronisation between parallel components. Rule (Rec) is standard. Rules (Bcast) and (TauPar) have their symmetric counterparts.

Below, we report a number of basic properties of our LTS.

Proposition 1. *Let M, M_1 and M_2 be well-formed networks.*

1. *$m \notin \mathsf{nds}(M)$ if and only if $M \xrightarrow{m?v} \Delta$, for some distribution Δ.*
2. *If $M_1 \mid M_2 \xrightarrow{m?v} \Delta$ if and only if there are Δ_1 and Δ_2 such that $M_1 \xrightarrow{m?v} \Delta_1$, $M_2 \xrightarrow{m?v} \Delta_2$ with $\Delta = \Delta_1 \mid \Delta_2$.*
3. *If $M \xrightarrow{m!v\triangleright\mu} \Delta$ then $M \equiv m[!\langle v\rangle.C]^{\nu} \mid N$, for some m, ν, C and N such that $m[!\langle v\rangle.C]^{\nu} \xrightarrow{m!v\triangleright\nu} [\![m[C]^{\nu}]\!]$, $N \xrightarrow{m?v} \Theta$, $\Delta \equiv [\![m[C]^{\nu}]\!] \mid \Theta$ and $\mu = \nu \setminus \mathsf{nds}(N)$.*
4. *If $M \xrightarrow{\tau} \Delta$ then $M \equiv m[\lfloor\tau.C\rfloor D]^{\nu} \mid N$, for some m, ν, C, D and N such that $m[\lfloor\tau.C\rfloor D]^{\nu} \xrightarrow{\tau} [\![m[C]^{\nu}]\!]$ and $\Delta \equiv [\![m[C]^{\nu}]\!] \mid \overline{N}$.*
5. *$M_1 \mid M_2 \xrightarrow{\sigma} \Delta$ if and only if there are Δ_1 and Δ_2 such that $M_1 \xrightarrow{\sigma} \Delta_1$, $M_2 \xrightarrow{\sigma} \Delta_2$ and $\Delta = \Delta_1 \mid \Delta_2$.*

As the topology of our networks is static and nodes cannot be created or destroyed, it is easy to prove the following result.

Proposition 2 (Well-formedness preservation). *Let M be a well-formed network. If $M \xrightarrow{\lambda} \Theta$ then Θ is a well-formed distribution.*

2.2 Time Properties

Our calculus enjoys a number of desirable time properties. Proposition 3 formalises the determinism nature of time passing: a network can reach at most one new state by executing the action σ.

Proposition 3 (Time Determinism). *Let M be a well-formed network. If $M \xrightarrow{\sigma} \Delta$ and $M \xrightarrow{\sigma} \Theta$ then Δ and Θ are syntactically the same.*

The maximal progress property says that sender nodes transmit immediately. Said in other words, the passage of time cannot block transmissions.

Proposition 4 (Maximal Progress). *Let M be a well-formed network. If $M \equiv m[!\langle v\rangle.C]^{\nu} \mid N$ then $M \xrightarrow{\sigma} \Delta$ for no distribution Δ.*

Patience guarantees that a process will wait indefinitely until it can communicate [7]. In our setting, this means that if no transmissions can start then it must be possible to execute a σ-action to let time pass.

Proposition 5 (Patience). *Let $M = \prod_{i \in I} m_i[P_i]^{\nu_i}$ be a well-formed network, such that for all $i \in I$ it holds that $P_i \neq !\langle v \rangle.C$, then there is a distribution Δ such that $M \xrightarrow{\sigma} \Delta$.*

Finally, as recursion is time-guarded, our networks satisfy the *well-timedness* (or *finite variability*) property [14]. Intuitively, only a finite number of instantaneous actions can fire between two contiguous σ-actions.

Proposition 6 (Well-Timedness). *For any well-formed network M there is an upper bound $k \in \mathbb{N}$ such that whenever $M \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_h} \Delta$, $\alpha_j \neq \sigma$ for $1 \leq j \leq h$, then $h \leq k$.*

3 Simulation Up to Probability

In this section, we use our pLTS to define an appropriate *probabilistic timed simulation theory* for `pTCWS`. Our focus is on weak similarities which abstract away non-observable actions. To this end, we extend the set of rules of Table 2 with the following two rules:

$$(\mathrm{Shh})\ \frac{M \xrightarrow{m!v\triangleright\emptyset} \Delta}{M \xrightarrow{\tau} \Delta} \qquad\qquad (\mathrm{Obs})\ \frac{M \xrightarrow{m!v\triangleright\nu} \Delta \quad \nu \neq \emptyset}{M \xrightarrow{!v\triangleright\nu} \Delta}$$

Rule (Shh) models transmissions that cannot be observed because none of the potential receivers is in the environment. Rule (Obs) models transmissions that can be observed by those nodes of the environment contained in ν. Notice that the name of the transmitter is removed from the label. This is motivated by the fact that nodes may refuse to reveal their identities, e.g. for security reasons or limited sensory capabilities in perceiving these identities. Notice also that in a derivation tree the rule (Obs) can only be applied at top-level.

In the rest of the paper, the metavariable α ranges over the following actions: $!v\triangleright\nu$, $m?v$, τ, and σ.

Let us provide the definition of weak transition. In a probabilistic setting, this definition is somewhat complicated by the fact that transitions go from processes (in our case networks) to distributions; consequently if we use weak transitions $\xRightarrow{\alpha}$, which abstract from sequences of internal actions, then we need to generalise transitions, so that they go from (sub-)distributions to (sub-)distributions. We write $M \xrightarrow{\hat{\tau}} \Delta$ if either $M \xrightarrow{\tau} \Delta$ or $\Delta = \overline{M}$, and $M \xrightarrow{\hat{\alpha}} \Delta$ if $M \xrightarrow{\alpha} \Delta$, for $\alpha \neq \tau$. Let $\Delta = \sum_{i \in I} p_i \cdot \overline{M_i}$ be a sub-distribution; we write $\Delta \xrightarrow{\hat{\alpha}} \Theta$ whenever $\Theta = \sum_{j \in J} p_i \cdot \Theta_j$, with $J \subseteq I$, and $M_j \xrightarrow{\hat{\alpha}} \Theta_j$, for any $j \in J$. We define the weak transition relation $\xRightarrow{\hat{\tau}}$ as the transitive and reflexive closure of $\xrightarrow{\hat{\tau}}$, i.e. $(\xrightarrow{\hat{\tau}})^*$, while for $\alpha \neq \tau$ we let $\xRightarrow{\hat{\alpha}}$ to denote $\xRightarrow{\hat{\tau}} \xrightarrow{\alpha} \xRightarrow{\hat{\tau}}$. Finally, independently whether α is τ or not, we write $\xRightarrow{\alpha}$ to denote $\xRightarrow{\hat{\tau}} \xrightarrow{\alpha} \xRightarrow{\hat{\tau}}$. Proposition 6 ensures that weak transitions always contain a bounded number of $\xrightarrow{\tau}$ actions.

Since transitions go from networks to distributions we need to lift our relations over networks to sub-distribution. Let $\mathcal{R} \subseteq \mathtt{pTCWS} \times \mathtt{pTCWS}$ be a binary relation over networks. We lift it to a relation $\overline{\mathcal{R}} \subseteq \mathcal{D}_{\mathrm{sub}}(\mathtt{pTCWS}) \times \mathcal{D}_{\mathrm{sub}}(\mathtt{pTCWS})$ by letting $\Delta \, \overline{\mathcal{R}} \, \Theta$ whenever:

- $\Delta = \sum_{i \in I} p_i \cdot \overline{M_i}$, where I is a finite index set
- for each $i \in I$ there is a network N_i such that $M_i \; \mathcal{R} \; N_i$ and $\Theta = \sum_{i \in I} p_i \cdot \overline{N_i}$.

Definition 6 (Simulation up to probability). *Let $p \in (0..1]$ be a probability. A parameterised relation $\mathcal{R}_p \subseteq$ pTCWS $\times$ pTCWS is said to be a* simulation up to probability p *if whenever $(M, N) \in \mathcal{R}_p$ and $M \xrightarrow{\alpha} \Delta$, there are a probability q, with $\frac{p}{q} \in (0..1]$, and a distribution Θ such that $N \overset{\hat{\alpha}}{\Longrightarrow} q \cdot \Theta$ and $\Delta \; \overline{\mathcal{R}_{\frac{p}{q}}} \; \Theta$. We write $M \sqsubseteq_p N$ if $(M, N) \in \mathcal{R}_p$ for some simulation up to probability $\mathcal{R}_p$. The equivalence induced by $\sqsubseteq_p$ is denoted $\simeq_p$.*

Intuitively, if $M \sqsubseteq_p N$ then M is simulated by N up to a probability (at least) p. Within the remaining probability $1 - p$, the network N might still simulate M. That is why the probability p is a lower-bound, i.e. $\sqsubseteq_q \subseteq \sqsubseteq_p$, for any $q \leq p$.

Example 1. 1. $n[P]^\nu \sqsubseteq_p n[\tau.(P \oplus_p Q)]^\nu$
2. $n[P]^\nu \sqsubseteq_q n[\tau.(P \oplus_p Q)]^\nu$ with $0 \leq q \leq p$
3. $n[Q]^\nu \sqsubseteq_{p(1-q)} n[\tau.(\tau.(P \oplus_q Q) \oplus_p R)]^\nu$
4. $n[!\langle v \rangle.!\langle w \rangle]^\nu \sqsubseteq_{pq} n[\tau.(!\langle v \rangle.\tau.(!\langle w \rangle \oplus_q P) \oplus_p Q)]^\nu$.

From these examples one can realise that when $M \sqsubseteq_p N$ the network N may contain a number of probabilistic choices which are resolved in M with a probability (at least) p. Unfortunately, this notion of similarity is not transitive, in the sense that is it not true that $\sqsubseteq_p \sqsubseteq_q = \sqsubseteq_{pq}$.[3] This would be a highly desirable property to algebraically reason on our networks. However, as one may have noticed from the first three algebraic laws of the previous example, the probability p is often manifested when executing the first action. So, to recover transitivity we add a root condition and replace weak transitions $\overset{\hat{\alpha}}{\Longrightarrow}$ with $\overset{\alpha}{\Longrightarrow}$.

Definition 7 (Rooted simulation up to probability). *Let $p \in (0..1]$ be a probability. A parameterised relation $\mathcal{R}_p \subseteq$ pTCWS $\times$ pTCWS is said to be a* rooted simulation up to probability p *if whenever $(M, N) \in \mathcal{R}_p$ and $M \xrightarrow{\alpha} \Delta$ there is a distribution θ such that $N \overset{\alpha}{\Longrightarrow} p \cdot \Theta$ and $\Delta \; \overline{\mathcal{R}_1} \; \Theta$. We write $M \sqsubseteq_p^{\mathbf{1}} N$ if $(M, N) \in \mathcal{R}_p$ for some rooted simulation up to probability $\mathcal{R}_p$. The equivalence induced by $\sqsubseteq_p^{\mathbf{1}}$ is denoted $\simeq_p^{\mathbf{1}}$.*

Proposition 7. *$M \sqsubseteq_p^{\mathbf{1}} N$ implies $M \sqsubseteq_p N$.*

Proposition 8. *If $M \sqsubseteq_p^{\mathbf{1}} N$ and $N \sqsubseteq_q^{\mathbf{1}} O$ then $M \sqsubseteq_{pq}^{\mathbf{1}} O$.*

Here comes a crucial result on the compositionality of our simulation theory.

[3] For details the reader is deferred to [12].

Theorem 1. *Let M, N and O be well-formed networks such that both $M \mid O$ and $N \mid O$ are well-formed as well. Then,*

1. *$M \sqsubseteq^{\mathbf{1}}_{p} N$ implies $M \mid O \sqsubseteq^{\mathbf{1}}_{p} N \mid O$*
2. *$M \sqsubseteq_{p} N$ implies $M \mid O \sqsubseteq_{p} N \mid O$.*

Below, we report a number of algebraic laws that will be useful in the next section when analysing gossip protocols.

Theorem 2 (Some algebraic laws).

1. $n[\sigma.\mathsf{nil}]^{\nu} \simeq^{\mathbf{1}}_{1} n[\mathsf{nil}]^{\nu}$
2. $\prod_{i\in I} m_i[P_i]^{\nu_{m_i}} \simeq^{\mathbf{1}}_{1} \prod_{j\in J} n_j[Q_j]^{\nu_{n_j}}$ *iff* $\prod_{i\in I} m_i[\sigma.P_i]^{\nu_{m_i}} \simeq^{\mathbf{1}}_{1} \prod_{j\in J} n_j[\sigma.Q_j]^{\nu_{n_j}}$
3. $n[\lfloor ?(x).P \rfloor Q]^{\nu} \simeq^{\mathbf{1}}_{1} n[\sigma.Q]^{\nu}$ *if no nodes in ν send in the current time interval.*
4. $n[?(x).P]^{\nu} \simeq^{\mathbf{1}}_{1} n[\mathsf{nil}]^{\nu}$ *if no nodes in ν contain sender processes*
5. $n[?(x).P]^{\nu} \simeq^{\mathbf{1}}_{1} n[\sigma.?(x).P]^{\nu}$ *if no nodes in ν send in the current time unit.*
6. $m[\tau.(!\langle v\rangle \oplus_p \mathsf{nil})]^{\nu} \mid \prod_{i\in I} n_i[P_i]^{\nu_i} \;{}^{\mathbf{1}}_{1}\!\sqsupseteq\; m[\mathsf{nil}]^{\nu} \mid \prod_{i\in I} n_i[P_i]^{\nu_i}$ *if $\nu = \bigcup_{i\in I} n_i$, and for all $i \in I$ either $P_i = \mathsf{nil}$ or $P_i = \sigma.Q_i$, for some Q_i.*

4 Gossipping without Collisions

The baseline model for our study is gossipping without collisions where all nodes are perfectly synchronised. For the sake of clarity, communication proceeds in synchronous rounds: A node can transmit or receive one message per round. In our implementation rounds are separated by σ-actions.

The processes involved in the protocol are the following:

$$\mathsf{snd}\langle u\rangle_{p_\mathrm{g}} \stackrel{\mathrm{def}}{=} \tau.(!\langle u\rangle \oplus_{p_\mathrm{g}} \mathsf{nil}) \quad \mathsf{resnd}\langle u\rangle_{p_\mathrm{g}} \stackrel{\mathrm{def}}{=} \sigma.\mathsf{snd}\langle u\rangle_{p_\mathrm{g}} \quad \mathsf{fwd}_{p_\mathrm{g}} \stackrel{\mathrm{def}}{=} ?(x).\mathsf{resnd}\langle x\rangle_{p_\mathrm{g}} \ .$$

Here, a sender broadcasts a value u with gossip probability $p_\mathrm{g}\in(0..1]$, and a forwarder gossips the received value, in the next round, with the same probability.

Now, we can apply our simulation theory to prove algebraic laws on message propagation. For instance, consider a fragment of a network with a sender m and two forwarder neighbours n_1 and n_2. Then, the following holds:

$$m[\mathsf{snd}\langle v\rangle_p]^{\nu} \mid n_1[\mathsf{fwd}_q]^{\nu_1} \mid n_2[\mathsf{fwd}_r]^{\nu_2} \;{}^{\mathbf{1}}_{p}\!\sqsupseteq\; m[\mathsf{nil}]^{\nu} \mid n_1[\mathsf{resnd}\langle v\rangle_q]^{\nu_1} \mid n_2[\mathsf{resnd}\langle v\rangle_r]^{\nu_2}$$

whenever $\nu = \{n_1, n_2\}$ and the nodes in $\nu_1 \cup \nu_2 \setminus \{m\}$ cannot transmit in the current instant of time. A complementary law is

$$m_1[\mathsf{snd}\langle v\rangle_{p_1}]^{n} \mid m_2[\mathsf{snd}\langle v\rangle_{p_2}]^{n} \mid n[\mathsf{fwd}_q]^{\nu} \;{}^{\mathbf{1}}_{p}\!\sqsupseteq\; m_1[\mathsf{nil}]^{n} \mid m_2[\mathsf{nil}]^{n} \mid n[\mathsf{resnd}\langle v\rangle_q]^{\nu}$$

with $p = 1-(1-p_1)(1-p_2)$, whenever the nodes in $\nu \setminus \{m_1, m_2\}$ cannot transmit in the current instant of time. More generally, the following result holds.

Theorem 3 (Message propagation). *Let K, I and J be pairwise disjoint subsets of $\mathbb{N}$. Let M be a well-formed network defined as*

$$M \equiv \prod_{k\in K} m_k[\mathsf{nil}]^{\nu_{m_k}} \;\big|\; \prod_{i\in I} m_i[\mathsf{snd}\langle v\rangle_{p_i}]^{\nu_{m_i}} \;\big|\; \prod_{j\in J} n_j[\mathsf{fwd}_{q_j}]^{\nu_{n_j}}$$

such that for all $i \in I$ it holds that $\bigcup_{j\in J} n_j \subseteq \nu_{m_i} \subseteq \bigcup_{j\in J} n_j \cup \bigcup_{k\in K\cup I} m_k$. Then,

$$M \quad {}^{\mathbf{1}}_{r}\!\sqsupseteq \quad \prod_{h\in K\cup I} m_h[\mathsf{nil}]^{\nu_{m_h}} \;\big|\; \prod_{j\in J} n_j[\mathsf{resnd}\langle v\rangle_{q_j}]^{\nu_{n_j}}$$

with $r = 1 - \prod_{i\in I}(1-p_i)$.

The previous theorem is a powerful tool to reason on gossip networks. However, it requires that all senders transmit to all subsequent forwarders. This may represent a limitation. Consider, for example, a simple gossip network GSP_1, with gossip probability p, composed by two source nodes s_1 and s_2, a destination node d and three intermediate nodes n_1, n_2 and n_3:

$$\mathrm{GSP}_1 \stackrel{\text{def}}{=} \prod_{i=1}^{2} s_i[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{fwd}_p]^{\nu_{n_i}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d}$$

with $\nu_{s_1}{=}\{n_1\}$, $\nu_{s_2}{=}\{n_1, n_2\}$, $\nu_{n_1}{=}\{s_1, n_3\}$, $\nu_{n_2}{=}\{s_1, s_2, n_3\}$, $\nu_{n_3}{=}\{n_1, n_2, d\}$.

The reader should notice that we cannot directly apply Theorem 3 to GSP_1. This is because node s_1, unlike s_2, can transmit to n_1 but not to n_2. Theorem 3 becomes much more effective when used together with Theorem 4 which allows us to compose estimates concerning partial networks. Roughly speaking, Theorem 4 allows us to consider in our calculation the probability that a sender transmits as well as the probability that the same sender does not transmit.

Theorem 4 (Composing networks).

$$M \;\big|\; m[\mathsf{snd}\langle v\rangle_p]^{\nu_m} \;\big|\; \prod_{j\in J} n_j[\lfloor ?(x_j).P_j\rfloor Q_j]^{\nu_{n_j}} \;\; {}^{\mathbf{1}}_{ps_1+(1-p)s_2}\!\sqsupseteq \;\; N$$

whenever

- $M \;|\; m[\mathsf{nil}]^{\nu_m} \;|\; \prod_{j\in J} n_j[\{{}^{v}\!/\!{}_{x_j}\}P_j]^{\nu_{n_j}} \; {}^{\mathbf{1}}_{s_1}\!\sqsupseteq \; N$
- $M \;|\; m[\mathsf{nil}]^{\nu_m} \;|\; \prod_{j\in J} n_j[\lfloor ?(x_j).P_j\rfloor Q_j]^{\nu_{n_j}} \; {}^{\mathbf{1}}_{s_2}\!\sqsupseteq \; N$
- $\bigcup_{j\in J} n_j \subseteq \nu_m \subseteq \bigcup_{j\in J} n_j \cup \mathsf{nds}(M)$
- *nodes in $\nu_m \cap \mathsf{nds}(M)$ cannot receive in the current instant of time.*[4]

Let us compute an estimate of success for the network GSP_1 previously defined. For verification reasons we assume that the environment contains a fresh node *test*, close to the destination, i.e. $\nu_d = \{n_3, \mathit{test}\}$, to test successful gossipping. For simplicity, we assume that the *test* node can receive messages but it cannot transmit.

[4] We could generalise the result to take into account more senders at the same time. This would not add expressivity, it would just speed up the reduction process.

We start proving the following chain of similarities by applying, in sequence, Theorem 2(6), Theorem 2(5), Theorem 3 together with Theorem 2(2), with $q = 1-(1-p)^2$, Theorem 2(5) together with Theorem 2(1), again Theorem 3 together with Theorem 2(2), and Theorem 2(1):

$$\begin{array}{l}
\quad s_1[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_1}} \;\big|\; s_2[\mathsf{nil}]^{\nu_{s_2}} \;\big|\; \prod_{i=1}^{2} n_i[\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_i}} \;\big|\; n_3[\mathsf{fwd}_p]^{\nu_{n_3}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{2} n_i[\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_i}} \;\big|\; n_3[\mathsf{fwd}_p]^{\nu_{n_3}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{2} n_i[\sigma.\mathsf{snd}\langle v\rangle_p]^{\nu_{n_i}} \;\big|\; n_3[\sigma.\mathsf{fwd}_p]^{\nu_{n_3}} \;\big|\; d[\sigma.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{q}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{2} n_i[\sigma.\mathsf{nil}]^{\nu_{n_i}} \;\big|\; n_3[\sigma.\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_3}} \;\big|\; d[\sigma.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{2} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; n_3[\sigma^2.\mathsf{snd}\langle v\rangle_p]^{\nu_{n_3}} \;\big|\; d[\sigma^2.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{p}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{2} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; n_3[\sigma^2.\mathsf{nil}]^{\nu_{n_3}} \;\big|\; d[\sigma^2.\mathsf{resnd}\langle v\rangle_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d} .
\end{array}$$

By Proposition 8 it follows that

$$\begin{array}{l}
s_1[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_1}} \;\big|\; s_2[\mathsf{nil}]^{\nu_{s_2}} \;\big|\; \prod_{i=1}^{2} n_i[\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_i}} \;\big|\; n_3[\mathsf{fwd}_p]^{\nu_{n_3}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \\
{}^{\qquad\mathbf{1}}_{p^2(2-p)}\!\sqsupseteq \;\prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d} \;\; .
\end{array}$$

Similarly, by applying in sequence, Theorem 3, Theorem 2(5), Theorem 3 together with Theorem 2(2), Theorem 2(5) together Theorem 2(1), again Theorem 3 together with Theorem 2(2), and finally Theorem 2(6) together with Theorem 2(1) we get:

$$\begin{array}{l}
\quad s_1[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_1}} \;\big|\; s_2[\mathsf{nil}]^{\nu_{s_2}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{fwd}_p]^{\nu_{n_i}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{p}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; n_1[\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_1}} \;\big|\; \prod_{i=2}^{3} n_i[\mathsf{fwd}_p]^{\nu_{n_i}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; n_1[\sigma.\mathsf{snd}\langle v\rangle_p]^{\nu_{n_1}} \;\big|\; \prod_{i=2}^{3} n_i[\sigma.\mathsf{fwd}_p]^{\nu_{n_i}} \;\big|\; d[\sigma.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{p}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; n_1[\sigma.\mathsf{nil}]^{\nu_{n_1}} \;\big|\; n_2[\sigma.\mathsf{fwd}_p]^{\nu_{n_2}} \;\big|\; n_3[\sigma.\mathsf{resnd}\langle v\rangle_p]^{\nu_{n_3}} \;\big|\; d[\sigma.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; n_1[\mathsf{nil}]^{\nu_{n_1}} \;\big|\; n_2[\sigma^2.\mathsf{fwd}_p]^{\nu_{n_2}} \;\big|\; n_3[\sigma^2.\mathsf{snd}\langle v\rangle_p]^{\nu_{n_3}} \;\big|\; d[\sigma^2.\mathsf{fwd}_1]^{\nu_d} \\
{}^{\mathbf{1}}_{p}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \,\big|\, n_1[\mathsf{nil}]^{\nu_{n_1}} \,\big|\, n_2[\sigma^3.\mathsf{snd}\langle v\rangle_p]^{\nu_{n_2}} \,\big|\, n_3[\sigma^2.\mathsf{nil}]^{\nu_{n_3}} \,\big|\, d[\sigma^2.\mathsf{resnd}\langle v\rangle_1]^{\nu_d} \\
{}^{\mathbf{1}}_{1}\!\sqsupseteq \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d} .
\end{array}$$

By Proposition 8 it follows that

$$\begin{array}{c}
s_1[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_1}} \;\big|\; s_2[\mathsf{nil}]^{\nu_{s_2}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{fwd}_p]^{\nu_{n_i}} \;\big|\; d[\mathsf{fwd}_1]^{\nu_d} \quad {}^{\mathbf{1}}_{p^3}\!\sqsupseteq \\
\prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d} \;\; .
\end{array}$$

Finally, we can apply Theorem 4 and Proposition 7 to derive:

$$\mathrm{GSP}_1 \;\; {}_{p^3(3-2p)}\!\sqsupseteq \;\; \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \;\big|\; \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \;\big|\; d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d} \;\; .$$

This result essentially says that the gossip network GSP_1 succeeds in transmitting the message v to the destination d, after three rounds, with probability (at least) $p^3(3-2p)$. Thus, for a gossip probability $p = 0.8$ the destination will receive the message with probability 0.72, with a margin of 10%. For $p = 0.85$ the probability at the destination increases to 0.8, with a margin of 6%; while for $p = 0.9$ the probability at destination rises to 0.88, with a difference of only 2%. So, $p = 0.9$ can be considered the threshold of our small network.[5]

5 Gossipping with Collisions

An important characteristic of the wireless domain is that transmissions are prone to collisions due to the well-known *hidden-terminal problem*. In the previous section we have reasoned assuming no collisions. In this section, we formally demonstrate that, as expected, the presence of communication collisions deteriorates the performances of gossip protocols.

A receiver node faces a collision if it is exposed to more than one transmission in the same round and as a result drops some of these transmissions. We can model this behaviour in pTCWS as follows:

$$\mathsf{resndc}\langle u\rangle_{p_{\mathrm{g}}} \stackrel{\text{def}}{=} \lfloor ?(x).\mathsf{nil}\rfloor \mathsf{snd}\langle u\rangle_{p_{\mathrm{g}}} \qquad\qquad \mathsf{fwdc}_{p_{\mathrm{g}}} \stackrel{\text{def}}{=} ?(x).\mathsf{resndc}\langle x\rangle_{p_{\mathrm{g}}} \quad .$$

Here, the forwarder process waits for a message in the current instant of time. If it receives a second message in the same round then it is doomed to fail. Otherwise, it moves to the next round and broadcasts the received message with gossip probability p_{g}. Thus, for example, the first law of the previous section becomes:

$$m_1[\mathsf{snd}\langle v\rangle_{p_1}]^n \mid m_2[\mathsf{snd}\langle v\rangle_{p_2}]^n \mid n[\mathsf{fwdc}_q]^\nu \;{}^{\mathbf{1}}_{p}\!\sqsupseteq\; m_1[\mathsf{nil}]^n \mid m_2[\mathsf{nil}]^n \mid n[\mathsf{resndc}\langle v\rangle_q]^\nu$$

with $p = p_1(1-p_2)+p_2(1-p_1)$ which is definitely smaller than $1-(1-p_1)(1-p_2)$, the lower bound seen in the previous section without collisions.

More generally, if collisions are taken into account Theorem 3 needs to be slightly changed as follows.

Theorem 5 (Message propagation with collision). *The same as Theorem 3 except for processes* fwd_{q_j} *and* $\mathsf{resnd}\langle v\rangle_{q_j}$ *which are replaced by* fwdc_{q_j} *and* $\mathsf{resndc}\langle v\rangle_{q_j}$*, respectively; the probability* r *is* $\sum_{i\in I} p_i \prod_{j\in I\setminus\{i\}}(1-p_j)$*.*

Here, the probability changes with respect to Theorem 3 because a forwarder successfully receives the value v only if exactly one sender transmits.

Let us apply our theorems to compute the probability of successful gossipping in the presence of collisions. Let us define:

$$\mathrm{GSP}_2 \stackrel{\text{def}}{=} \prod_{i=1}^{2} s_i[\mathsf{snd}\langle v\rangle_p]^{\nu_{s_i}} \mid \prod_{i=1}^{3} n_i[\mathsf{fwdc}_p]^{\nu_{n_i}} \mid d[\mathsf{fwdc}_1]^{\nu_d}$$

with the same network topology as GSP_1.

[5] Had we considered a larger network, with more senders, we would have obtained a more significant threshold.

By applying Theorem 5 and Theorem 2 to compute estimates, Theorem 4 to compose such estimates, and Proposition 7, we obtain:

$$\mathrm{GSP}_2 \ \ {}_q\!\sqsupseteq \ \ \prod_{i=1}^{2} s_i[\mathsf{nil}]^{\nu_{s_i}} \mid \prod_{i=1}^{3} n_i[\mathsf{nil}]^{\nu_{n_i}} \mid d[\sigma^3.\mathsf{snd}\langle v\rangle_1]^{\nu_d}$$

with $q = p(2p^2(1-p)^2 + p^3) + (1-p)p^3 = p^3(3 - 4p + 2p^2)$. This probability is definitely smaller than that computed for GSP_1, demonstrating that collisions degrade the performances of gossip protocols. Thus, for instance, for a gossip probability $p = 0.8$ the destination in GSP_2 will receive the message with probability 0.55 while in GSP_1 this probability is 0.72; similarly for $p = 0.9$ the probability of success in GSP_2 is about 0.74 while in GSP_1 it is 0.88.

6 Conclusions, Future and Related Work

We have proposed a probabilistic simulation theory to compare the performances of gossip protocols for wireless sensor networks. This theory is used to prove a number of algebraic laws which revealed to be very effective to evaluate the performances of gossip networks with and without communication collisions. Our simulation theory provides lower-bound probabilities. However, due to the inherent structure of gossip networks, the probabilities of our algebraic laws are actually *precise* (see [12] for details). As future work, we will study gossip networks with *random delays* and *lossy channels*. Moreover, we intend to mechanise the application of our laws to deal with large-scale gossip networks.

A nice survey of formal verification techniques for the analysis of gossip protocols is presented in [2]. Probabilistic model-checking has been used in [6] to study the influence of different modelling choices on message propagation in flooding and gossip protocols. It has been used also in [10] to investigate the expected rounds of gossipping required to form a connected network. However, the analysis of gossip protocols in large-scale networks remains beyond the capabilities of current probabilistic model-checking tools. For this reason, the paper [3] suggests to apply mean-field analysis for a formal evaluation of gossip protocols.

Several process calculi for wireless systems have been proposed in the last five years. Our calculus is a probabilistic variant of [4] which takes inspiration from [5,11]. The paper [17] contains the first probabilistic untimed calculus for wireless systems, where connections are established with a given probability.

Our notion of simulation up to probability may remind one of the idea of simulation with a fixed precision. A first version of probabilistic bisimulation with ϵ precision appeared in [1] to relax security constrains. Indeed their simulation is able to tolerate local fluctuations by allowing small differences in the probability of occurrence of weak actions. In [13] a theory of approximate equivalence for task-structured Probabilistic I/O Automata is proposed. In this case, the distance between probabilities is based on trace distributions. Afterwards, in order to study simulations in cryptographic protocols [16] proposed a notion of simulation where distances may grow by a negligible value at each step.

Acknowledgements. The referees provided useful suggestions.

References

1. Aldini, A., Bravetti, M., Gorrieri, R.: A process-algebraic approach for the analysis of probabilistic noninterference. Journal of Computer Security 12, 191–245 (2004)
2. Bakhshi, R., Bonnet, F., Fokkink, W., Haverkort, B.: Formal analysis techniques for gossiping protocols. Operating Systems Review 41(5), 28–36 (2007)
3. Bakhshi, R., Cloth, L., Fokkink, W., Haverkort, B.: Mean-field analysis for the evaluation of gossip protocols. In: Huth, M., Nicol, D. (eds.) QEST, pp. 247–256. IEEE Computer Society, Budapest (2009)
4. Ballardin, F., Merro, M.: A calculus for the analysis of wireless network security protocols. In: Degano, P., Etalle, S., Guttman, J. (eds.) FAST 2010. LNCS, vol. 6561, pp. 206–222. Springer, Heidelberg (2011)
5. Deng, Y., van Glabbeek, R., Hennessy, M., Morgan, C.: Characterising testing preorders for finite probabilistic processes. Logical Methods in Computer Science 4(4) (2008)
6. Fehnker, A., Gao, P.: Formal verification and simulation for performance analysis for probabilistic broadcast protocols. In: Kunz, T., Ravi, S. (eds.) ADHOC-NOW 2006. LNCS, vol. 4104, pp. 128–141. Springer, Heidelberg (2006)
7. Hennessy, M., Regan, T.: A process algebra for timed systems. Information and Computation 117(2), 221–239 (1995)
8. Jonsson, B., Ho-Stuart, C., Yi, W.: Testing and refinement for nondeterministic and probabilistic processes. In: Langmaack, H., de Roever, W., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 418–430. Springer, Heidelberg (1994)
9. Kermarrec, A., van Steen, M.: Gossiping in distributed systems. Operating Systems Review 41(5), 2–7 (2007)
10. Kwiatkowska, M., Norman, G., Parker, D.: Analysis of a gossip protocol in prism. SIGMETRICS Performance Evaluation Review 36(3), 17–22 (2008)
11. Kwiatkowska, M., Norman, G., Parker, D., Vigliotti, G.M.: Probabilistic mobile ambients. Theoretical Computuer Science 410(12-13), 1272–1303 (2009)
12. Lanotte, R., Merro, M.: Semantic analysis of gossip protocols for wireless sensor networks, Forthcoming Technical Report, Dept. Computer Science, Verona (2011)
13. Mitra, S., Lynch, N.: Proving approximate implementations for probabilistic I/O automata. Electronic Notes in Theoret. Comput. Science 174(8), 71–93 (2007)
14. Nicollin, X., Sifakis, J.: An overview and synthesis on timed process algebras. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 376–398. Springer, Heidelberg (1992)
15. Segala, R.: Modeling and Verification of Randomized Distributed Real-Time Systems. Ph.D. thesis, Laboratory for Computer Science, MIT (1995)
16. Segala, R., Turrini, A.: Approximated computationally bounded simulation relations for probabilistic automata. In: Sabelfeld, A. (ed.) CSF, pp. 140–156. IEEE Computer Society, Venice (2007)
17. Song, L., Godskesen, J.: Probabilistic mobility models for mobile and wireless networks. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 86–100. Springer, Heidelberg (2010)

An Automaton over Data Words That Captures EMSO Logic

Benedikt Bollig

LSV, ENS Cachan, CNRS & INRIA, France
bollig@lsv.ens-cachan.fr

Abstract. We develop a general framework for the specification and implementation of systems whose executions are words, or partial orders, over an infinite alphabet. As a model of an implementation, we introduce class register automata, a one-way automata model over words with multiple data values. Our model combines register automata and class memory automata. It has natural interpretations. In particular, it captures communicating automata with an unbounded number of processes, whose semantics can be described as a set of (dynamic) message sequence charts. On the specification side, we provide a local existential monadic second-order logic that does not impose any restriction on the number of variables. We study the realizability problem and show that every formula from that logic can be effectively, and in elementary time, translated into an equivalent class register automaton.

1 Introduction

A recent research stream, motivated by models from XML database theory, considers *data words*, i.e., strings over an infinite alphabet [2, 8, 11, 17, 19]. The alphabet is the cartesian product of a finite supply of *labels* and an infinite supply of *data values*. While labels may represent, e.g., an XML tag or reveal the type of an action that a system performs, data values can be used to model time stamps [8], process identifiers [5, 21], or text contents in XML documents.

We will consider data words as behavioral models of concurrent systems. In this regard, it is natural to look at suitable logics and automata. Logical formulas may serve as specifications, and automata as system models or tools for deciding logical theories. This viewpoint raises the following classical problems/tasks: *satisfiability* (does a given logical formula have a model ?), *model checking* (do all executions of an automaton satisfy a given formula ?), and *realizability* (given a formula, construct a system model in terms of an automaton whose executions are precisely the models of the formula). Much work has indeed gone into defining logics and automata for data words, with a focus on satisfiability [4, 10].

One of the first logical approaches to data words is due to [8]. Since then, a two-variable logic has become a commonly accepted yardstick wrt. expressivity and decidability [4]. The logic contains a predicate to compare data values of two positions for equality. Its satisfiability problem is decidable, indeed, but supposedly of very high complexity. An elementary upper bound has been obtained

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 171–186, 2011.

only for weaker fragments [4, 10]. For specification of communicating systems, however, two-variable logic is of limited use: it cannot express properties like "whenever a process Pid1 spawns some Pid2, then this is followed by a message from Pid2 to Pid1". Actually, the logic was studied for words with only one data value at each each position, which is not enough to encode executions of message-passing systems. But three-variable logics as well as extensions to two data values lead to undecidability. To put it bluntly, any "interesting" logic for dynamic communicating systems has an undecidable satisfiability problem.

Instead of satisfiability or model checking, we therefore consider realizability. A system model that *realizes* a given formula can be considered correct by construction. Realizability questions for data words have, so far, been neglected. One reason may be that there is actually no automaton that could serve as a realistic system model. Though data words naturally reflect executions of systems with an unbounded number of threads, existing automata fail to model distributed computation. Three features are minimum requirements for a suitable system model. First, the automaton should be a *one-way device*, i.e., read an execution once, processing it "from left to right" (unlike data automata [4], class automata [3], two-way register automata, and pebble automata [17]). Second, it should be *non-deterministic* (unlike alternating automata [11, 17]). Third, it should reflect paradigms that are used in concurrent programming languages such as process creation and message passing. Two known models match the first two properties: register automata [13, 14, 21] and class memory automata [2]; but they clearly do not fulfill the last requirement.

Contribution. We provide an existential MSO logic over data words, denoted rEMSO, which does not impose any restriction on the number of variables. The logic is strictly more expressive than the two-variable logic from [4] and suitable to express interesting properties of dynamic communicating systems.

We then define *class register automata* as a system model. They are a mix of register automata [13, 14, 21] and class memory automata [2]. A class register automaton is a non-deterministic one-way device. Like a class memory automaton, it can access certain configurations in the past. However, we extend the notion of a configuration, which is no longer a simple state but composed of a state *and* some data values that are stored in registers. This is common in concurrent programming languages and can be interpreted as "read current state of a process" or "send process identity from one to another process". Moreover, it is in the spirit of communicating finite-state machines [9] or nested-word automata [1], where more than one resource (state, channel, stack, etc.) can be accessed at a time. Actually, our automata run over directed acyclic graphs rather than words. To our knowledge, they are the first automata model of true concurrency that deals with structures over infinite alphabets.

We study the realizability problem and show that, for every rEMSO formula, we can compute, in elementary time, an equivalent class register automaton. The translation is based on Hanf's locality theorem [12] and properly generalizes [7] to a dynamic setting.

Outline. Sections 2 and 3 introduce data words and their logics. In Section 4, we define the new automata model. Section 5 is devoted to the realizability problem and states our main result. In Section 6, we give translations from automata back to logic. We conclude in Section 7. Omitted proofs, as well as an extension of our main result to infinite data words, can be found in the full version of this paper available at: `http://hal.archives-ouvertes.fr/hal-00558757/`

2 Data Words

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote the set of natural numbers. For $m \in \mathbb{N}$, we denote by $[m]$ the set $\{1, \ldots, m\}$. A *boolean formula* over a (possibly infinite) set A of *atoms* is a finite object generated by the grammar $\beta ::= \mathit{true} \mid \mathit{false} \mid a \in A \mid \neg\beta \mid \beta \vee \beta \mid \beta \wedge \beta$. For an assignment of truth values to elements of A, a boolean formula β is evaluated to true or false as usual. Its size $|\beta|$ is the number of vertices of its syntax tree. Moreover, $|A| \in \mathbb{N} \cup \{\infty\}$ denotes the size of a set A. The symbol $\cong$ will be used to denote isomorphism of two structures.

We fix an infinite set $\mathfrak{D}$ of *data values*. Note that $\mathfrak{D}$ can be *any* infinite set. For examples, however, we usually choose $\mathfrak{D} = \mathbb{N}$. In a data word, every position will carry $m \geq 0$ data values. It will also carry a *label* from a non-empty finite alphabet Σ. Thus, a *data word* is a finite sequence over $\Sigma \times \mathfrak{D}^m$ (over Σ if $m = 0$). Given a data word $w = (a_1, d_1) \ldots (a_n, d_n)$ with $a_i \in \Sigma$ and $d_i = (d_i^1, \ldots, d_i^m) \in \mathfrak{D}^m$, we let $\ell(i)$ refer to label a_i and $d^k(i)$ to data value d_i^k.

Classical words without data come with natural relations on word positions such as the direct successor relation $\prec_{+1}$ and its transitive closure $<$. In the context of data words with one data value (i.e., $m = 1$), it is natural to consider also a relation $\prec_{\sim}$ for successive positions with identical data values [4]. As, in the present paper, we deal with multiple data values, we generalize these notions in terms of a signature. A *signature* $\mathcal{S}$ is a pair $(\sigma, \mathfrak{I})$. It consists of a finite set σ of binary *relation symbols* and an *interpretation* $\mathfrak{I}$. The latter associates, with every $\lhd \in \sigma$ and every data word $w = w_1 \ldots w_n \in (\Sigma \times \mathfrak{D}^m)^*$, a relation $\lhd^w \subseteq [n] \times [n]$ such that the following hold, for all word positions $i, j, i', j' \in [n]$:

(1) $i \lhd^w j$ implies $i < j$
(2) there is at most one k such that $i \lhd^w k$
(3) there is at most one k such that $k \lhd^w i$
(4) if $i \lhd^w j$ and $i' \lhd^w j'$ and $w_i = w_{i'}$ and $w_j = w_{j'}$, then $i < i'$ iff $j < j'$

In other words, we require that $\lhd^w$ (1) complies with $<$, (2) has out-degree at most one, (3) has in-degree at most one, and (4) is monotone. Our translation from logic into automata will be symbolic and independent of $\mathfrak{I}$, but its applicability and correctness rely upon the above conditions. However, several examples will demonstrate that the framework is quite flexible and allows us to capture existing logics and automata for data words. Note that $\lhd^w$ can indeed be *any* relation satisfying (1)–(4). It could even assume an order on $\mathfrak{D}$.

As the interpretation $\mathfrak{I}$ is mostly understood, we may identify $\mathcal{S}$ with σ and write $\lhd \in \mathcal{S}$ instead of $\lhd \in \sigma$, or $|\mathcal{S}|$ to denote $|\sigma|$. If not stated otherwise, we let in the following $\mathcal{S}$ be any signature.

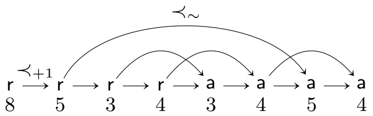


Fig. 1. Data word over $\mathbb{S}^1_{+1,\sim}$

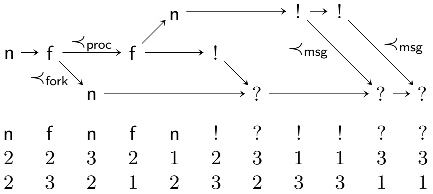


Fig. 2. Data word over $\mathbb{S}^2_{\mathsf{dyn}}$

Example 1. Typical examples of relation symbols include $\prec_{+1}$ and $\prec^k_\sim$ relating direct successors and, respectively, successive positions with the same k-th data value: For $w = w_1 \dots w_n$, we let $\prec^w_{+1} = \{(i, i+1) \mid i \in \{1, \dots, n-1\}\}$ and $(\prec^k_\sim)^w = \{(i,j) \mid 1 \leq i < j \leq n,\ d^k(i) = d^k(j)$, and there is no $i < i' < j$ such that $d^k(i) = d^k(i')\}$. When $m = 1$, we write $\prec_\sim$ instead of $\prec^1_\sim$. Automata and logic have been well studied in the presence of one single data value ($m = 1$) and for signature $\mathbb{S}^1_{+1,\sim} = \{\prec_{+1}, \prec_\sim\}$ with the above interpretation [2, 4]. Here, and in the following, we adopt the convention that the upper index of a signature denotes the number m of data values. Figure 1 depicts a data word over $\Sigma = \{\mathsf{r}, \mathsf{a}\}$ (request/acknowledgment) and $\mathfrak{D} = \mathbb{N}$ as well as the relations $\prec_{+1}$ (straight arrows) and $\prec_\sim$ (curved arrows) imposed by $\mathbb{S}^1_{+1,\sim}$. ◇

Example 2. We develop a framework for message-passing systems with dynamic process creation. Each process has a unique identifier from $\mathfrak{D} = \mathbb{N}$. Process $c \in \mathbb{N}$ can execute an action $\mathsf{f}(c,d)$, which forks a new process with identity d. This action is eventually followed by $\mathsf{n}(d,c)$, indicating that d is new (created by c) and begins its execution. Processes can exchange messages. When c executes $!(c,d)$, it sends a message through an unbounded first-in-first-out (FIFO) channel $c \to d$. Process d may execute $?(d,c)$ to receive the message. Elements from $\Sigma_{\mathsf{dyn}} = \{\mathsf{f}, \mathsf{n}, !, ?\}$ reveal the nature of an action, which requires two identities so that we choose $m = 2$. When a process performs an action, it should access the current state of (i) its own, (ii) the spawning process if a new-action is executed, and (iii) the sending process if a receive is executed (message contents are encoded in states). To this aim, we define a signature $\mathbb{S}^2_{\mathsf{dyn}} = \{\prec_{\mathsf{proc}}, \prec_{\mathsf{fork}}, \prec_{\mathsf{msg}}\}$ with the following interpretation. Assume $w = w_1 \dots w_n \in (\Sigma_{\mathsf{dyn}} \times \mathbb{N} \times \mathbb{N})^*$ and consider, for $a, b \in \Sigma_{\mathsf{dyn}}$ and $i, j \in [n]$, the property

$$P_{(a,b)}(i,j) \;=\; (\ell(i) = a \wedge \ell(j) = b \wedge d^1(i) = d^2(j) \wedge d^2(i) = d^1(j))\,.$$

We set $\prec^w_{\mathsf{proc}} = (\prec^1_\sim)^w$, which relates successive positions with the same executing process. Moreover, let $i \prec^w_{\mathsf{fork}} j$ if $i < j$, $P_{(\mathsf{f},\mathsf{n})}(i,j)$, and there is no $i < k < j$ such that $P_{(\mathsf{f},\mathsf{n})}(i,k)$ or $P_{(\mathsf{f},\mathsf{n})}(k,j)$. Finally, we set $i \prec^w_{\mathsf{msg}} j$ if $i < j$, $P_{(!,?)}(i,j)$, and

$$|\{i' < i \mid P_{(!,?)}(i',j)\}| \;=\; |\{j' < j \mid P_{(!,?)}(i,j')\}|\,.$$

This models FIFO communication. An example data word is given in Figure 2, which also depicts the relations induced by $\mathbb{S}^2_{\mathsf{dyn}}$. Horizontal arrows reflect $\prec_{\mathsf{proc}}$, vertical arrows either $\prec_{\mathsf{fork}}$ or $\prec_{\mathsf{msg}}$, depending on the labels. Note that $\mathsf{n}(2,2)$ is executed by "root process" 2, which was not spawned by some other process. $\Diamond$

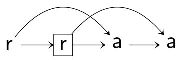
Our principal proof technique relies on a graph abstraction of data words. Let $Part(m)$ be the set of all partitions of $[m]$. An $\mathbb{S}$-*graph* is a (node- and edge-labeled) graph $G = (V, (\lhd^G)_{\lhd \in \mathbb{S}}, \lambda, \nu)$. Here, V is the finite set of nodes, $\lambda : V \to \Sigma$ and $\nu : V \to Part(m)$ are node-labeling functions, and each $\lhd^G \subseteq V \times V$ is a set of edges such that, for all $i \in V$, there is at most one $j \in V$ with $i \lhd^G j$, and there is at most one $j \in V$ with $j \lhd^G i$. We represent $\lhd^G$ and $(\lhd^G)^{-1}$ as partial functions and set $\mathsf{next}^G_{\lhd}(i) = j$ if $i \lhd^G j$, and $\mathsf{prev}^G_{\lhd}(i) = j$ if $j \lhd^G i$.

Local graph patterns, so-called spheres, will also play a key role. For nodes $i, j \in V$, we denote by $dist^G(i,j)$ the *distance* between i and j, i.e., the length of the shortest path from i to j in the undirected graph $(V, \bigcup_{\lhd \in \mathbb{S}} \lhd^G \cup (\lhd^G)^{-1})$ (if such a path exists). In particular, $dist^G(i,i) = 0$. For some *radius* $B \in \mathbb{N}$, the *B-sphere of G around i*, denoted by $B\text{-}Sph^G(i)$, is the substructure of G induced by $\{j \in V \mid dist^G(i,j) \leq B\}$. In addition, it contains the distinguished element i as a constant, called *sphere center*.

These notions naturally transfer to data words: With word w of length n, we associate the graph $G(w) = ([n], (\lhd^w)_{\lhd \in \mathbb{S}}, \lambda, \nu)$ where λ maps i to $\ell(i)$ and ν maps i to $\{\{l \in [m] \mid d^k(i) = d^l(i)\} \mid k \in [m]\}$. Thus, $K \in \nu(i)$ contains indices with the same data value at position i. Now, $\mathsf{next}^w_{\lhd}$, $\mathsf{prev}^w_{\lhd}$, $dist^w$, and $B\text{-}Sph^w(i)$ are defined with reference to the graph $G(w)$. We hereby assume that $\mathbb{S}$ is understood. We might also omit the index w if it is clear from the context.

Data words u and v are called ($\mathbb{S}$-)*equivalent* if $G(u) \cong G(v)$. For a language L, we let $[L]_{\mathbb{S}}$ denote the set of words that are equivalent to some word in L.

Given the data word w from Figure 1, we have $dist^w(1,8) = 3$. The picture on the right shows $1\text{-}Sph^w(4)$. The sphere center is framed by a rectangle; node labelings of the form $\{\{1\}\}$ are omitted.



3 Logic

We consider monadic second-order logic to specify properties of data words. Let us fix countably infinite supplies of first-order variables $\{x, y, \ldots\}$ and second-order variables $\{X, Y, \ldots\}$.

The set $\mathrm{MSO}(\mathbb{S})$ of *monadic second-order formulas* is given by the grammar

$$\varphi ::= \ell(x) = a \mid d^k(x) = d^l(y) \mid x \lhd y \mid x = y \mid x \in X \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x\, \varphi \mid \exists X\, \varphi$$

where $a \in \Sigma$, $k, l \in [m]$, $\lhd \in \mathbb{S}$, x and y are first-order variables, and X is a second-order variable. The *size* $|\varphi|$ of φ is the number of nodes of its syntax tree.

Important fragments of $\mathrm{MSO}(\mathbb{S})$ are $\mathrm{FO}(\mathbb{S})$, the set of first-order formulas, which do not use any second-order quantifier, and $\mathrm{EMSO}(\mathbb{S})$, the set of formulas of the form $\exists X_1 \ldots \exists X_n\, \varphi$ with $\varphi \in \mathrm{FO}(\mathbb{S})$.

The models of a formula are data words. First-order variables are interpreted as word positions and second-order variables as sets of positions. Formula $\ell(x) =$

a holds in data word w if position x carries an a, and formula $d^k(x) = d^l(y)$ holds if the k-th data value at position x equals the l-th data value at position y. Moreover, $x \lhd y$ is satisfied if $x \lhd^w y$. The atomic formulas $x = y$ and $x \in X$ as well as quantification and boolean connectives are interpreted as usual.

For realizability, we will actually consider a restricted, more "local" logic: let rMSO($\mathcal{S}$) denote the fragment of MSO($\mathcal{S}$) where we can only use $d^k(x) = d^l(x)$ instead of the more general $d^k(x) = d^l(y)$. Thus, data values of *distinct* positions can only be compared via $x \lhd y$. This implies that rMSO($\mathcal{S}$) cannot distinguish between words u and v such that $G(u) \cong G(v)$. The fragments rFO($\mathcal{S}$) and rEMSO($\mathcal{S}$) of rMSO($\mathcal{S}$) are defined as expected.

In the case of one data value ($m = 1$), we will also refer to the logic $\mathrm{EMSO}_2(\mathcal{S}^1_{+1,\sim} \cup \{<\})$ that was considered in [4] and restricts EMSO logic to two first-order variables. The predicate $<$ is interpreted as the strict linear order on word positions (strictly speaking, it is not part of a signature as we defined it). We shall later see that rEMSO($\mathcal{S}^1_{+1,\sim}$) is strictly more expressive than $\mathrm{EMSO}_2(\mathcal{S}^1_{+1,\sim} \cup \{<\})$, though the latter involves the non-local predicates $d^1(x) = d^1(y)$ and $<$. This gain in expressiveness comes at the price of an undecidable satisfiability problem.

A *sentence* is a formula without free variables. The language defined by sentence φ, i.e., the set of its models, is denoted by $L(\varphi)$. By $\mathbb{MSO}(\mathcal{S})$, r$\mathbb{MSO}(\mathcal{S})$, r$\mathbb{EMSO}(\mathcal{S})$, etc., we refer to the corresponding language classes.

Example 3. Think of a server that can receive requests (r) from an unbounded number of processes, and acknowledge (a) them. We let $\Sigma = \{\mathsf{r}, \mathsf{a}\}$, $\mathfrak{D} = \mathbb{N}$, and $m = 1$. A data value from $\mathfrak{D}$ is used to model the process identity of the requesting and acknowledged process. We present three properties formulated in rFO($\mathcal{S}^1_{+1,\sim}$). Formula $\varphi_1 = \exists x \exists y\, (\ell(x) = \mathsf{r} \wedge \ell(y) = \mathsf{a} \wedge x \prec_\sim y)$ expresses that there is a request that is acknowledged. Dually, $\varphi_2 = \forall x \exists y\, (\ell(x) = \mathsf{r} \rightarrow \ell(y) = \mathsf{a} \wedge x \prec_\sim y)$ says that every request is acknowledged before the same process sends another request. A last formula guarantees that two *successive* requests are acknowledged in the order they were received:

$$\varphi_3 = \forall x, y \left(\begin{array}{c} \ell(x) = \mathsf{r} \wedge \ell(y) = \mathsf{r} \wedge x \prec_{+1} y \\ \rightarrow \exists x', y' \left(\, \ell(x') = \mathsf{a} \wedge \ell(y') = \mathsf{a} \wedge x \prec_\sim x' \prec_{+1} y' \wedge y \prec_\sim y' \, \right) \end{array} \right)$$

This is not expressible in $\mathrm{EMSO}_2(\mathcal{S}^1_{+1,\sim} \cup \{<\})$. We will see that $\varphi_1, \varphi_2, \varphi_3$ form a hierarchy of languages that correspond to different automata models, our new model capturing φ_3. ◊

Example 4. We pursue Example 2 and consider Σ_{dyn} with signature $\mathcal{S}^2_{\mathsf{dyn}}$. Recall that we wish to model systems where an unbounded number of processes communicate via message-passing through unbounded FIFO channels. Obviously, not every data word represents an execution of such a system. Therefore, we identify some *well formed* data words, which have to satisfy $\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \in$ rFO($\mathcal{S}^2_{\mathsf{dyn}}$) given as follows. We require that there is exactly one root process: $\varphi_1 = \exists x \left(\ell(x) = \mathsf{n} \wedge d^1(x) = d^2(x) \wedge \forall y\, (d^1(y) = d^2(y) \rightarrow x = y) \right)$. Next, we assume that every fork is followed by a corresponding new-action, the first action of a process is a new-event, and every new process was forked by some other process:

$$\varphi_2 = \forall x \left(\begin{array}{l} \quad \ell(x) = \mathsf{f} \rightarrow \exists y \, (x \prec_{\mathsf{fork}} y) \\ \wedge \, \ell(x) = \mathsf{n} \leftrightarrow \neg \exists y \, (y \prec_{\mathsf{proc}} x) \\ \wedge \, \ell(x) = \mathsf{n} \rightarrow \big(d^1(x) = d^2(x) \vee \exists y \, (y \prec_{\mathsf{fork}} x) \big) \end{array} \right)$$

Finally, every send should be followed by a receive, and a receive be preceded by a send action: $\varphi_3 = \forall x \, \big(\ell(x) \in \{\,!\,,?\,\} \rightarrow \exists y \, (x \prec_{\mathsf{msg}} y \vee y \prec_{\mathsf{msg}} x)\big)$. This formula actually ensures that, for every $c, d \in \mathbb{N}$, there are as many symbols $!(c,d)$ as $?(d,c)$, the N-th send symbol being matched with the N-th receive symbol. We call a data word over Σ_{dyn} and $\mathcal{S}^2_{\mathsf{dyn}}$ a *message sequence chart* (MSC, for short) if it satisfies $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Figure 2 shows an MSC and the induced relations. When we restrict to MSCs, our logic corresponds to that from [16]. Note that model checking rMSO($\mathcal{S}^2_{\mathsf{dyn}}$) specifications against *fork-and-join grammars*, which can generate infinite sets of MSCs, is decidable [16].

A last rFO($\mathcal{S}^2_{\mathsf{dyn}}$)-formula (which is not satisfied by all MSCs) specifies that, whenever a process c forks some d, then this is followed by a message from d to c: $\forall x_1, y_1 \; (x_1 \prec_{\mathsf{fork}} y_1 \; \rightarrow \; \exists x_2, y_2 \, (x_1 \prec_{\mathsf{proc}} x_2 \; \wedge \; y_1 \prec_{\mathsf{proc}} y_2 \prec_{\mathsf{msg}} x_2))$. ◇

4 Class Register Automata

In this section, we define class register automata, a non-deterministic one-way automata model that captures rEMSO logic. It combines register automata [13, 14] and class memory automata [2]. When processing a data word, data values from the current position can be stored in registers. The automaton reads the data word from left to right but can look back on certain states and register contents from the past (e.g., at the last position that is executed by the same process). Positions that can be accessed in this way are determined by the signature $\mathcal{S}$. Their register entries can be compared with one another, or with current values from the input. Moreover, when taking a transition, registers can be updated by either a current value, an old register entry, or a guessed value.

Definition 1. *A* class register automaton *(over signature $\mathcal{S}$) is a tuple $\mathcal{A} = (Q, R, \Delta, (F_\triangleleft)_{\triangleleft \in \mathcal{S}}, \Phi)$ where Q is a finite set of* states*, R is a finite set of* registers*, the $F_\triangleleft \subseteq Q$ are sets of* local final states*, and Φ is the* global acceptance condition*: a boolean formula over $\{\, `q \leq N' \mid q \in Q$ and $N \in \mathbb{N}\}$. Moreover, Δ is a finite set of* transitions *of the form*

$$(p, g) \xrightarrow{a} (q, f) \,.$$

Here, $p : \mathcal{S} \rightharpoonup Q$ is a partial mapping representing the source states. Moreover, g is a guard, i.e., a boolean formula over $\{\, `\theta_1 = \theta_2' \mid \theta_1, \theta_2 \in [m] \cup (\mathrm{dom}(p) \times R)\}$ to perform comparisons of values that are are currently read and those that are stored in registers. Finally, $a \in \Sigma$ is the current label, $q \in Q$ is the target state, and $f : R \rightharpoonup (\mathrm{dom}(p) \times R) \cup ([m] \times \mathbb{N})$ is a partial mapping to update registers.

Hereby, $\mathrm{dom}(p)$ denotes the domain of p. In the following, we write $p_\lhd$ instead of $p(\lhd)$. Transition $(p, g) \xrightarrow{a} (q, f)$ can be executed at position i of a data word if the state at position $\mathsf{prev}_\lhd(i)$ is $p_\lhd$ (for all $\lhd \in \mathrm{dom}(p)$) and, for a register guard $(\lhd_1, r_1) = (\lhd_2, r_2)$, the entry of register r_1 at $\mathsf{prev}_{\lhd_1}(i)$ equals that of r_2 at $\mathsf{prev}_{\lhd_2}(i)$. The automaton then reads the label a together with a tuple of data values that also passes the test given by g, and goes to q. Moreover, register r obtains a new value according to $f(r)$: if $f(r) = (\lhd, r') \in \mathrm{dom}(p) \times R$, then the new value of r is the value of r' at position $\mathsf{prev}_\lhd(i)$; if $f(r) = (k, B) \in [m] \times \mathbb{N}$, then r obtains any k-th data value in the B-sphere around i. In particular, $f(r) = (k, 0)$ assigns to r the (unique) k-th data value of the current position. To some extent, $f(r) = (k, B)$ calls an oracle to guess a data value. The guess is local and, therefore, weaker than [14], where a non-deterministic reassignment allows one to write *any* data value into a register. This latter approach can indeed simulate our local version (this is not immediately clear, but can be shown using the *sphere automaton* from Section 5).

Let us be more precise. A configuration of $\mathcal{A}$ is a pair (q, ρ) where $q \in Q$ is the current state and $\rho : R \rightharpoonup \mathfrak{D}$ is a partial mapping denoting the current register contents. If $\rho(r)$ is undefined, then there is no entry in r. Let $w = w_1 \dots w_n \in (\Sigma \times \mathfrak{D}^m)^*$ be a data word and $\xi = (q_1, \rho_1) \dots (q_n, \rho_n)$ be a sequence of configurations. For $i \in [n]$, $k \in [m]$, and $B \in \mathbb{N}$, let $\mathfrak{D}^k_B(i) = \{d^k(j) \mid j \in [n]$ such that $\mathit{dist}^w(i, j) \leq B\}$. We call ξ a *run* of $\mathcal{A}$ on w if, for every position $i \in [n]$, there is a transition $(p_i, g_i) \xrightarrow{\ell(i)} (q_i, f_i)$ such that the following hold:

(1) $\mathrm{dom}(p_i) = \{\lhd \in \mathbb{S} \mid \mathsf{prev}_\lhd(i) \text{ is defined}\}$

(2) for all $\lhd \in \mathrm{dom}(p_i)$: $(p_i)_\lhd = q_{\mathsf{prev}_\lhd(i)}$

(3) g_i is evaluated to true on the basis of its atomic subformulas: $\theta_1 = \theta_2$ is true iff $\mathit{val}_i(\theta_1) = \mathit{val}_i(\theta_2) \in \mathfrak{D}$ where $\mathit{val}_i(k) = d^k(i)$ and $\mathit{val}_i((\lhd, r)) = \rho_{\mathsf{prev}_\lhd(i)}(r)$ (the latter might be undefined and, therefore, not be in $\mathfrak{D}$)

(4) for all $r \in R$: $\begin{cases} \rho_i(r) = \rho_{\mathsf{prev}_\lhd(i)}(r') & \text{if } f_i(r) = (\lhd, r') \in \mathrm{dom}(p) \times R \\ \rho_i(r) \in \mathfrak{D}^k_B(i) & \text{if } f_i(r) = (k, B) \in [m] \times \mathbb{N} \\ \rho_i(r) \text{ undefined} & \text{if } f_i(r) \text{ undefined} \end{cases}$

Run ξ is accepting if $q_i \in F_\lhd$ for all $i \in [n]$ and $\lhd \in \mathbb{S}$ such that $\mathsf{next}_\lhd(i)$ is undefined. Moreover, we require that the global condition Φ is met. Hereby, an atomic constraint $q \leq N$ is satisfied by ξ if $|\{\, i \in [n] \mid q_i = q\}| \leq N$. The language $L(\mathcal{A}) \subseteq (\Sigma \times \mathfrak{D}^m)^*$ of $\mathcal{A}$ is defined in the obvious manner. The corresponding language class is denoted by $\mathbb{CRA}(\mathbb{S})$.

The acceptance conditions are inspired by Björklund and Schwentick [2], who also distinguish between local and global acceptance. Local final states can be motivated as follows. When data values model process identities, a $\prec_\sim$-maximal position of a data word is the last position of some process and must give rise to a local final state. Moreover, in the context of $\mathbb{S}^2_{\mathsf{dyn}}$, a sending position that does not lead to a local final state in $F_{\prec_{\mathsf{msg}}}$ requires a matching receive event. Thus, local final states can be used to model "communication requests". The global acceptance condition of class register automata is more general than that of [2] to cope with all possible signatures. However, in the special case of $\mathcal{S}^1_{+1,\sim}$, there is some global control in terms of $\prec_{+1}$. We could then perform some counting up to a finite threshold and restrict, like [2], to a set of global final states.

We can classify many of the non-deterministic one-way models from the literature (most of them defined for $m = 1$) in our unifying framework:

- A *class memory automaton* [2] is a class register automaton where, in all transitions $(p, g) \xrightarrow{a} (q, f)$, the update function f is undefined everywhere. The corresponding language class is denoted by $\mathbb{CMA}(\mathcal{S})$.
- As an intermediary subclass of class register automata, we consider *non-guessing class register automata*: for all transitions $(p, g) \xrightarrow{a} (q, f)$ and registers r, one requires $f(r) \in (\mathrm{dom}(p) \times R) \cup ([m] \times \{0\})$. We denote the corresponding language class by $\mathbb{CRA}^-(\mathcal{S})$.
- A *register automaton* [11, 13] is a non-guessing class register automaton over $\mathcal{S}^m_{+1} = \{\prec_{+1}\}$. Moreover, non-guessing class register automata over $\mathcal{S}^1_{+1,\sim}$ capture *fresh-register automata* [21], which can dynamically generate data values that do not occur in the history of a run. Actually, this feature is also present in dynamic communicating automata [5] and in class memory automata over $\mathcal{S}^1_{+1,\sim}$ where a fresh data value is guaranteed by a transition $(p, g) \xrightarrow{a} (q, f)$ such that $p_{\prec_\sim}$ is undefined.
- Class register automata are a model of distributed computation: considered over Σ_{dyn} and $\mathcal{S}^2_{\mathsf{dyn}}$, they subsume dynamic communicating automata [5]. In particular, they can handle unbounded process creation and message passing. Updates of the form $f(r) = (\prec_{\mathsf{fork}}, r')$ and $f(r) = (\prec_{\mathsf{msg}}, r')$ correspond to receiving a process identity from the spawning/sending process. Moreover, when a process requests a message from the thread whose identity is stored in register r, a corresponding transition is guarded by $(\prec_{\mathsf{proc}}, r) = (\prec_{\mathsf{msg}}, r_0)$ where we assume that every process keeps its identity in some register r_0.

Example 5. Let us give a concrete example. Suppose $\Sigma = \{\mathsf{r}, \mathsf{a}\}$ and $\mathfrak{D} = \mathbb{N}$. We pursue Example 3 and build a non-guessing class register automaton $\mathcal{A}$ over $\mathcal{S}^1_{+1,\sim}$ for $L = [\{(\mathsf{r}, 1) \dots (\mathsf{r}, n)(\mathsf{a}, 1) \dots (\mathsf{a}, n) \mid n \geq 1\}]_{\mathcal{S}^1_{+1,\sim}}$. Roughly speaking, there is a request phase followed by an acknowledgment phase, and requests are acknowledged in the order they are received. Figure 3 presents $\mathcal{A}$ and an accepting run on $(\mathsf{r}, 8)(\mathsf{r}, 5)(\mathsf{a}, 8)(\mathsf{a}, 5)$. The states of $\mathcal{A}$ are q_1 and q_2. State q_1 is assigned to request positions (first phase), state q_2 to acknowledgments (second phase). Moreover, $\mathcal{A}$ is equipped with registers r_1 and r_2. During the first phase, r_1 always contains the data value of the current position, and r_2 the data value of the $\prec_{+1}$-predecessor (unless we deal with the very first position, where r_2 is undefined, denoted $\perp$). These invariants are ensured by transitions 1 and 2. In the second phase, by transition 3, position $n+1$ carries the same data value as the first position, which is the only request with undefined r_2. Guard $(\prec_\sim, r_2) = \perp$ is actually an abbreviation for $\neg((\prec_\sim, r_2) = (\prec_\sim, r_2))$. By transition 4, position $n + i$ with $i \geq 2$ has to match the request position whose r_2-contents equals r_1 at $n + i - 1$. Finally, $F_{\prec_\sim} = \{q_2\}$, $F_{\prec_{+1}} = \{q_2\}$, and $\Phi = \neg(q_1 \leq 0)$. ◊

Transitions

	source (p)		guard (g)	input	q	update (f)
	$\prec_\sim$	$\prec_{+1}$				
1				(r, d)	q_1	$r_1 := d$
2		q_1		(r, d)	q_1	$r_1 := d$ $r_2 := (\prec_{+1}, r_1)$
3	q_1	q_1	$(\prec_\sim, r_2) = \bot$	(a, d)	q_2	$r_1 := d$
4	q_1	q_2	$(\prec_\sim, r_2) = (\prec_{+1}, r_1)$	(a, d)	q_2	$r_1 := d$

Run

input	state	r_1	r_2
$(\mathsf{r}, 8)$	q_1	8	$\bot$
$(\mathsf{r}, 5)$	q_1	5	8
$(\mathsf{a}, 8)$	q_2	8	$\bot$
$(\mathsf{a}, 5)$	q_2	5	$\bot$

Fig. 3. A non-guessing class register automaton over $\mathbb{S}^1_{+1,\sim}$ and a run

For the language L from Example 5, one can show $L \notin \mathbb{CMA}(\mathbb{S}^1_{+1,\sim})$, using an easy pumping argument. Next, we will see that non-guessing class register automata, though more expressive than class memory automata, are not yet enough to capture rEMSO logic. Thus, dropping just one feature such as registers or guessing data values makes class register automata incomparable to the logic. Assume $m = 2$ and consider $\mathbb{S}^2_\sim = \{\prec^1_\sim, \prec^2_\sim\}$ (cf. Example 1).

Lemma 1. $\mathrm{r}\mathbb{FO}(\mathbb{S}^2_\sim) \not\subseteq \mathbb{CRA}^-(\mathbb{S}^2_\sim)$.

The proof of Lemma 1 can be adapted to show $\mathrm{r}\mathbb{FO}(\mathbb{S}^2_{\mathsf{dyn}}) \not\subseteq \mathbb{CRA}^-(\mathbb{S}^2_{\mathsf{dyn}})$. It reveals that non-guessing class register automata can in general not detect *cycles*. However, this is needed to capture rFO logic [12]. In Section 5, we show that *full* class register automata capture rFO and, as they are closed under projection, also rEMSO logic. Closure under projection is meant in the following sense. Let Γ be a non-empty finite alphabet. Given $\mathbb{S} = (\sigma, \mathfrak{I})$, we define another signature $\mathbb{S}_\Gamma$ for data words over $(\Sigma \times \Gamma) \times \mathfrak{D}^m$. Its set of relation symbols is $\{\lhd_\Gamma \mid \lhd \in \mathbb{S}\}$. For $w \in ((\Sigma \times \Gamma) \times \mathfrak{D}^m)^*$, we set $i \lhd^w_\Gamma j$ iff $i \lhd^{proj_\Sigma(w)} j$. Hereby, the projection $proj_\Sigma$ just removes the Γ component while keeping Σ and the data values. For $\mathcal{C} \in \{\mathbb{CRA}, \mathbb{CRA}^-, \mathbb{CMA}\}$, we say that $\mathcal{C}(\mathbb{S})$ is *closed under projection* if, for every Γ and $L \subseteq ((\Sigma \times \Gamma) \times \mathfrak{D}^m)^*$, $L \in \mathcal{C}(\mathbb{S}_\Gamma)$ implies $proj_\Sigma(L) \in \mathcal{C}(\mathbb{S})$.

Lemma 2. *For every signature* $\mathbb{S}$, $\mathbb{CRA}(\mathbb{S})$, $\mathbb{CRA}^-(\mathbb{S})$, *and* $\mathbb{CMA}(\mathbb{S})$ *are closed under union, intersection, and projection. They are, in general, not closed under complementation.*

5 Realizability of EMSO Specifications

In this section, we solve the realizability problem for rEMSO specifications:

Theorem 1. *For all signatures* $\mathbb{S}$, $\mathrm{r}\mathbb{EMSO}(\mathbb{S}) \subseteq \mathbb{CRA}(\mathbb{S})$. *An automaton can be computed in elementary time and is of elementary size.*

Classical procedures that translate formulas into automata follow an inductive approach, use two-way mechanisms and tools such as pebbles, or rely on reductions to existing translations. There is no obvious way to apply any of these techniques to prove our theorem.

We therefore follow a technique from [7], which is based on ideas from [18, 20]. We first transform the first-order kernel of the formula at hand into a normal form due to Hanf [12]. According to that normal form, satisfaction of a first-order formula wrt. data word w only depends on the spheres that occur in $G(w)$, and on how often they occur, counted up to a threshold. The size of a sphere is bounded by a radius that depends on the formula. The threshold can be computed from the radius and $|\mathcal{S}|$. We can indeed apply Hanf's Theorem, as the structures that we consider have *bounded degree*: every node/word position has at most $|\mathcal{S}|$ incoming and at most $|\mathcal{S}|$ outgoing edges. In a second step, we transform the formula in normal form into a class register automaton.

Recall that $B\text{-}Sph^G(i)$ denotes the B-sphere of graph/data word G around i (cf. Section 2). Its size (number of nodes) is bounded by $maxSize := (2|\mathcal{S}|+2)^B$. Let $B\text{-}Spheres_{\mathcal{S}} = \{B\text{-}Sph^G(i) \mid G = (V,\ldots)$ is an $\mathcal{S}$-graph and $i \in V\}$. We do not distinguish between isomorphic structures so that $B\text{-}Spheres_{\mathcal{S}}$ is finite.

Theorem 2 (cf. [6, 12]). *Let $\varphi \in \mathrm{rFO}(\mathcal{S})$. One can compute, in elementary time, $B \in \mathbb{N}$ and a boolean formula β over $\{$'$S \leq N$' $\mid S \in B\text{-}Spheres_{\mathcal{S}}$ and $N \in \mathbb{N}\}$ such that $L(\varphi)$ is the set of data words that satisfy β. Here, we say that $w = w_1 \ldots w_n$ satisfies atom $S \leq N$ iff $|\{i \in [n] \mid B\text{-}Sph^w(i) \cong S\}| \leq N$. The radius B and the size of β and its constants N are elementary in $|\varphi|$ and $|\mathcal{S}|$.*

By Theorem 2, it will be useful to have a class register automaton that, when reading a position i of data word w, outputs the sphere of w around i. Its construction is actually the main difficulty in the proof of Theorem 1, as spheres have to be computed "in one go", i.e., reading the word from left to right, while accessing only certain configurations from the past.

Proposition 1. *Let $B \in \mathbb{N}$. One can compute, in elementary time, a class register automaton $\mathcal{A}_B = (Q, R, \Delta, (F_\lhd)_{\lhd \in \mathcal{S}}, true)$ over $\mathcal{S}$, as well as a mapping $\pi : Q \to B\text{-}Spheres_{\mathcal{S}}$ such that $L(\mathcal{A}_B) = (\Sigma \times \mathfrak{D}^m)^*$ and, for every data word $w = w_1 \ldots w_n$, every accepting run $(q_1, \rho_1) \ldots (q_n, \rho_n)$ of $\mathcal{A}_B$ on w, and every $i \in [n]$, $\pi(q_i) \cong B\text{-}Sph^w(i)$. Moreover, $|Q|$ and $|R|$ are elementary in B and $|\mathcal{S}|$.*

The proposition is proved below. Let us first show how we can use it, together with Theorem 2, to translate an rEMSO formula into a class register automaton.

Proof (of Theorem 1). Let $\varphi = \exists X_1 \ldots \exists X_n \, \psi \in \mathrm{rEMSO}(\mathcal{S})$ be a sentence with $\psi \in \mathrm{rFO}(\mathcal{S})$ (we also assume $n \geq 1$). Since Theorem 2 applies to first-order formulas only, we extend Σ to $\Sigma \times \Gamma$ where $\Gamma = 2^{\{1,\ldots,n\}}$. Consider the extended signature $\mathcal{S}_\Gamma$ (cf. Section 4). From ψ, we obtain a formula $\psi_\Gamma \in \mathrm{rFO}(\mathcal{S}_\Gamma)$ by replacing $\ell(x) = a$ with $\bigvee_{M \in \Gamma} \ell(x) = (a, M)$ and $x \in X_j$ with $\bigvee_{a \in \Sigma,\, M \in \Gamma} \ell(x) = (a, M \cup \{j\})$. Consider the radius $B \in \mathbb{N}$ and the normal form β_Γ for ψ_Γ due to Theorem 2. Let $\mathcal{A}_B = (Q, R, \Delta, (F_\lhd)_{\lhd \in \mathcal{S}_\Gamma}, true)$ be the class register automaton over $\mathcal{S}_\Gamma$ from Proposition 1 and π be the associated mapping. The global acceptance condition of $\mathcal{A}_B$ is obtained from β_Γ by replacing every atom $S \leq N$ with $\pi^{-1}(S) \leq N$ (which can be expressed as a suitable boolean formula). We hold $\mathcal{A}'_B$, a class register automaton satisfying $L(\mathcal{A}'_B) = L(\psi_\Gamma)$. Exploiting closure under projection (Lemma 2), we obtain a class register automaton over $\mathbb{S}$ that recognizes $L(\varphi) = \mathit{proj}_\Sigma(L(\psi_\Gamma))$. □

The Sphere Automaton. In the remainder of this section, we construct the class register automaton $\mathcal{A}_B = (Q, R, \Delta, (F_\lhd)_{\lhd \in \mathbb{S}}, \mathit{true})$ from Proposition 1, together with $\pi : Q \to B\text{-}\mathit{Spheres}_\mathbb{S}$. The idea is that, at each position i in the data word w at hand, $\mathcal{A}_B$ guesses the B-sphere S of w around i. To verify that the guess is correct, i.e., $S \cong B\text{-}\mathit{Sph}^w(i)$, S is passed to each position that is connected to i by an edge in $G(w)$. That new position locally checks label and data equalities imposed by S, then also forwards S to its neighbors, and so on. Thus, at any time, several local patterns have to be validated simultaneously so that a state $q \in Q$ is actually a *set* of spheres. In fact, we consider *extended* spheres $E = (S, \alpha, \mathit{col})$ where $S = (U, (\lhd^E)_{\lhd \in \mathbb{S}}, \lambda, \nu, \gamma)$ is a sphere (with universe U and sphere center γ), $\alpha \in U$ is the *active node*, and col is a color from a finite set, which will be specified later. The active node α indicates the current context, i.e., it corresponds to the position currently read.

Let $B\text{-}\mathit{eSpheres}_\mathbb{S}$ denote the set of extended spheres, which is finite up to isomorphism. For $E = (S, \alpha, \mathit{col}) \in B\text{-}\mathit{eSpheres}_\mathbb{S}$, $S = (U, (\lhd^E)_{\lhd \in \mathbb{S}}, \lambda, \nu, \gamma)$, and $j \in U$, we let $E[j]$ refer to the extended sphere (S, j, col) where the active node α has been replaced with j. Now suppose that the state q of $\mathcal{A}_B$ that is reached after reading position i of data word w contains $E = (S, \alpha, \mathit{col})$. Roughly speaking, this means that the neighborhood of i in w shall look like the neighborhood of α in S. Thus, if S contains j' such that $\alpha \lhd^E j'$, then we must find i' such that $i \lhd^w i'$ in the data word. Local final states will guarantee that i' indeed exists. Moreover, the state assigned to i' in a run of $\mathcal{A}_B$ will contain the new proof obligation $E[j']$ and so forth. Similarly, an edge in (the graph of) w has to be present in spheres, unless it is beyond their scope, which is limited by B. All this is reflected below, in conditions T2–T6 of a transition.

We are still facing two major difficulties. Several *isomorphic* spheres have to be verified simultaneously, i.e., a state must be allowed to include isomorphic spheres in different contexts. A solution to this problem is provided by the additional coloring col. It makes sure that centers of overlapping isomorphic spheres with different colors refer to distinct nodes in the input word. To put it differently, for a given position i in data word w, there may be i' such that $0 < \mathit{dist}^w(i, i') \leq 2B+1$ and $B\text{-}\mathit{Sph}^w(i) \cong B\text{-}\mathit{Sph}^w(i')$. Fortunately, there cannot be more than $(2|\mathbb{S}|+1) \cdot \mathit{maxSize}^2$ such positions. As a consequence, the coloring col can be restricted to the set $\{1, \ldots, (2|\mathbb{S}|+1) \cdot \mathit{maxSize}^2 + 1\}$.

Implementing these ideas alone would do without registers and yield a class memory automaton. But this cannot work due to Lemma 1. Indeed, a faithful simulation of cycles in spheres has to make use of data values. They need to be anticipated, stored in registers, and locally compared with current data values from the input word. We introduce a register (E, k) for every extended sphere E and $k \in [m]$. To get the idea behind this, consider a run $(q_1, \rho_1) \ldots (q_n, \rho_n)$ of $\mathcal{A}_B$ on $w = (a_1, d_1) \ldots (a_n, d_n)$. Pick a position i of w and suppose that $E = (U, (\lhd^E)_{\lhd \in \mathbb{S}}, \lambda, \nu, \gamma, \alpha, \mathit{col}) \in q_i$. If α is minimal in E, then there is no pending requirement to check. Now, as α shall correspond to the current position

i of w, we write, for every $k \in [m]$, d_i^k into register (E, k) (first case of T8 below). For all $j \in U \setminus \{\alpha\}$, on the other hand, we anticipate data values and store them in $(E[j], k)$ (also first case of T8). They will be forwarded (second case of T8) and checked later against both the guesses made at other minimal nodes of E (second line in T7) and the actual data values in w (end of line 1 in T7). This procedure makes sure that the values that we carry along within an accepting run agree with the actual data values of w.

Now, as $\mathsf{prev}_{\lhd}^{w}$ and $\mathsf{next}_{\lhd}^{w}$ are monotone wrt. positions with identical labels and data values, two isomorphic cycles cannot be "merged" into one larger one, unlike in non-guessing class register automata where different parts may act erroneously on the assumption of inconsistent data values (cf. Lemma 1). As a consequence, spheres are correctly simulated by the input word.

Let us formalize $\mathcal{A}_B = (Q, R, \Delta, (F_{\lhd})_{\lhd \in \mathcal{S}}, \mathit{true})$ and the mapping $\pi : Q \to B\text{-}\mathit{Spheres}_{\mathcal{S}}$, following the above ideas. The set of registers is $R = B\text{-}\mathit{eSpheres}_{\mathcal{S}} \times [m]$. A state from Q is a non-empty set $q \subseteq B\text{-}\mathit{eSpheres}_{\mathcal{S}}$ such that

(i) there is a unique $E = (U, (\lhd^E)_{\lhd \in \mathcal{S}}, \lambda, \nu, \gamma, \alpha, \mathit{col}) \in q$ such that $\gamma = \alpha$ (we set $\pi(q) = (U, (\lhd^E)_{\lhd \in \mathcal{S}}, \lambda, \nu, \gamma)$ to obtain the mapping required by Prop. 1),

(ii) there are $a \in \Sigma$ and $\eta \in \mathit{Part}(m)$ such that, for all $E = (\ldots, \lambda, \nu, \ldots) \in q$, we have $\lambda(\alpha) = a$ and $\nu(\alpha) = \eta$ (we let $\mathit{label}(q) = a$ and $\mathit{data}(q) = \eta$), and

(iii) for every $(S, \alpha, \mathit{col}), (S, \alpha', \mathit{col}) \in q$, we have $\alpha = \alpha'$.

Before we turn to the transitions, we introduce some notation. Below, E will always denote $(S, \alpha, \mathit{col})$ with $S = (U, (\lhd^E)_{\lhd \in \mathcal{S}}, \lambda, \nu, \gamma)$; in particular, α refers to the active node of E. The mappings $\mathsf{next}_{\lhd}^{E}$, $\mathsf{prev}_{\lhd}^{E}$, and dist^E are defined for extended spheres in the obvious manner. For $j \in U$, we set $\mathit{type}^-(j) = \{\lhd \in \mathcal{S} \mid \mathsf{prev}_{\lhd}^{E}(j) \text{ is defined}\}$. Let us fix, for all $E \in B\text{-}\mathit{eSpheres}_{\mathcal{S}}$ such that $\mathit{type}^-(\alpha) \neq \emptyset$, some arbitrary $\lhd_E \in \mathit{type}^-(\alpha)$. Finally, for state q and $k_1, k_2 \in [m]$, we write $k_1 \sim_q k_2$ if there is $K \in \mathit{data}(q)$ such that $\{k_1, k_2\} \subseteq K$.

We have a transition $(p, g) \xrightarrow{a} (q, f)$ iff the following hold:

T1 $\mathit{label}(q) = a$

T2 for all $\lhd \in \mathcal{S}$, $E \in q$: $\quad \lhd \notin \mathrm{dom}(p) \implies \mathsf{prev}_{\lhd}^{E}(\alpha)$ is undefined

T3 for all $\lhd \in \mathrm{dom}(p)$, $E \in q$, $j \in U$: $\quad j \lhd^E \alpha \iff E[j] \in p_{\lhd}$

T4 for all $\lhd \in \mathrm{dom}(p)$, $E \in p_{\lhd}$, $j \in U$: $\quad \alpha \lhd^E j \iff E[j] \in q$

T5 for all $\lhd \in \mathrm{dom}(p)$, $E \in q$: $\quad \mathsf{prev}_{\lhd}^{E}(\alpha)$ undefined $\implies \mathit{dist}^E(\gamma, \alpha) = B$

T6 for all $\lhd \in \mathrm{dom}(p)$, $E \in p_{\lhd}$: $\quad \mathsf{next}_{\lhd}^{E}(\alpha)$ undefined $\implies \mathit{dist}^E(\gamma, \alpha) = B$

T7
$$g = \bigwedge_{\substack{k_1, k_2 \in [m] \\ k_1 \sim_q k_2}} k_1 = k_2 \;\wedge \bigwedge_{\substack{k_1, k_2 \in [m] \\ k_1 \not\sim_q k_2}} \neg\,(k_1 = k_2) \;\wedge \bigwedge_{\substack{k \in [m] \;\; E \in q \\ \lhd \in \mathit{type}^-(\alpha)}} k = (\lhd, (E, k))$$
$$\wedge \bigwedge_{\substack{k \in [m] \quad E \in q \quad j \in U \\ \lhd_1, \lhd_2 \in \mathit{type}^-(\alpha)}} (\lhd_1, (E[j], k)) = (\lhd_2, (E[j], k))$$

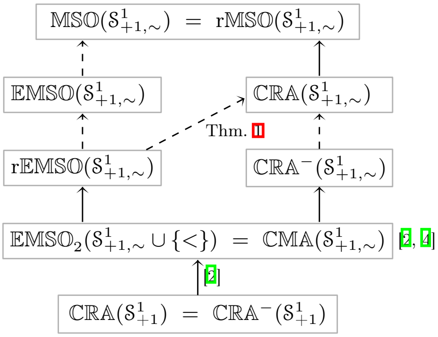

$\mathbb{MSO}(\mathcal{S}^1_{+1,\sim}) = \mathrm{r}\mathbb{MSO}(\mathcal{S}^1_{+1,\sim})$
$\mathbb{EMSO}(\mathcal{S}^1_{+1,\sim})$
$\mathbb{CRA}(\mathcal{S}^1_{+1,\sim})$
Thm. 1
$\mathrm{r}\mathbb{EMSO}(\mathcal{S}^1_{+1,\sim})$
$\mathbb{CRA}^-(\mathcal{S}^1_{+1,\sim})$
$\mathbb{EMSO}_2(\mathcal{S}^1_{+1,\sim} \cup \{<\}) = \mathbb{CMA}(\mathcal{S}^1_{+1,\sim})$ [2, 4]
[2]
$\mathbb{CRA}(\mathcal{S}^1_{+1}) = \mathbb{CRA}^-(\mathcal{S}^1_{+1})$

Fig. 4. A hierarchy of automata and logics over one-dimensional data words

T8 for all $k \in [m]$ and $E \in B\text{-}eSpheres_{\mathcal{S}}$:

$$f((E,k)) = \begin{cases} (k, dist^E(j,\alpha)) & \text{if } \exists j \in U : E[j] \in q \text{ and } type^-(j) = \emptyset \\ (\lhd_{E[j]}, (E,k)) & \text{if } \exists j \in U : E[j] \in q \text{ and } type^-(j) \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

For every $\lhd \in \mathcal{S}$, the local acceptance condition is given by $F_\lhd = \{q \in Q \mid \text{for all } E \in q,\ \mathsf{next}^E_\lhd(\alpha) \text{ is undefined}\}$. Recall that the global one is *true*.

As the maximal size of a sphere is exponential in B and polynomial in $|\mathcal{S}|$, the numbers $|Q|$ and $|R|$ are elementary in B and $|\mathcal{S}|$. Note that $\mathcal{A}_B$ can actually be constructed in elementary time.

6 From Automata to Logic

Next, we give translations from automata back to logic. Note that $\mathrm{r}\mathbb{EMSO}(\mathcal{S}^1_{+1}) \subsetneq \mathbb{CRA}(\mathcal{S}^1_{+1})$, as rEMSO$(\mathcal{S}^1_{+1})$ cannot reason about data values. However, we show that the behavior of a class register automaton is always MSO definable and, in a sense, "regular". There are natural finite-state automata that do not share this property: two-way register automata (even deterministic ones) over one-dimensional data words are incomparable to MSO$(\mathcal{S}^1_{+1,\sim})$ [17].

Theorem 3. *For every signature $\mathcal{S}$, we have $\mathbb{CRA}(\mathcal{S}) \subseteq \mathbb{MSO}(\mathcal{S})$.*

In the proof, the non-local predicate $d^k(x) = d^l(y)$ is indeed essential to simulate register assignments, as we need to compare data values at positions where registers are updated. For one-dimensional data words, however, the predicate can be easily defined in rMSO$(\mathcal{S}^1_{+1,\sim})$. The following theorem is dedicated to this classical setting over $\mathcal{S}^1_{+1,\sim}$.

Theorem 4. *We have the inclusions depicted in Figure 4. Here, $\longrightarrow$ means 'strictly included' and $\dashrightarrow$ means 'included'.*

The remaining (strict) inclusions are left open. When there are no data values, we have expressive equivalence of EMSO logic and class register automata (which then reduce to class memory automata). The translation from automata to logic follows the standard approach. The following theorem is a proper generalization of the main result of [7].

Theorem 5. *Suppose* $m = 0$. *For every signature* $\mathcal{S}$, $\mathbb{EMSO}(\mathcal{S}) = \mathbb{CRA}(\mathcal{S})$.

7 Conclusion

We studied the realizability problem for data-word languages. A particular case of this general framework constitutes a first step towards a logically motivated automata theory for dynamic message-passing systems. As future work, it remains to study alternative specification formalisms such as temporal logic [15]. It would also be interesting to extend [16], whose logic corresponds to ours in the case of $\mathcal{S}^2_{\mathsf{dyn}}$, to general data words.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journal of the ACM 56(3), 1–43 (2009)
2. Björklund, H., Schwentick, T.: On notions of regularity for data languages. Theoretical Computer Science 411(4-5), 702–715 (2010)
3. Bojańczyk, M., Lasota, S.: An extension of data automata that captures XPath. In: LICS 2010, pp. 243–252. IEEE Computer Society, Los Alamitos (2010)
4. Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: LICS 2006, pp. 7–16. IEEE Computer Society, Los Alamitos (2006)
5. Bollig, B., Hélouët, L.: Realizability of dynamic MSC languages. In: Ablayev, F., Mayr, E. (eds.) CSR 2010. LNCS, vol. 6072, pp. 48–59. Springer, Heidelberg (2010)
6. Bollig, B., Kuske, D.: An optimal construction of Hanf sentences, arXiv:1105.5487 (2011)
7. Bollig, B., Leucker, M.: Message-passing automata are expressively equivalent to EMSO logic. Theoretical Computer Science 358(2), 150–172 (2006)
8. Bouyer, P.: A logical characterization of data languages. Information Processing Letters 84(2), 75–85 (2002)
9. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM 30(2) (1983)
10. David, C., Libkin, L., Tan, T.: On the satisfiability of two-variable logic over data words. In: Fermüller, C., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 248–262. Springer, Heidelberg (2010)
11. Demri, S., Lazić, R.: LTL with the freeze quantifier and register automata. ACM Transactions on Computational Logic 10(3) (2009)
12. Hanf, W.: Model-theoretic methods in the study of elementary logic. In: Addison, J.W., Henkin, L., Tarski, A. (eds.) The Theory of Models. North-Holland, Amsterdam (1965)
13. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329–363 (1994)

14. Kaminski, M., Zeitlin, D.: Finite-memory automata with non-deterministic reassignment. International Journal of Foundations of Computer Science 21(5), 741–760 (2010)
15. Kara, A., Schwentick, T., Zeume, T.: Temporal logics on words with multiple data values. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2010. LIPIcs, vol. 8, pp. 481–492 (2010)
16. Leucker, M., Madhusudan, P., Mukhopadhyay, S.: Dynamic message sequence charts. In: Agrawal, M., Seth, A. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 253–264. Springer, Heidelberg (2002)
17. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Transactions on Computational Logic 5(3), 403–435 (2004)
18. Schwentick, T., Barthelmann, K.: Local normal forms for first-order logic with applications to games and automata. Discrete Mathematics & Theoretical Computer Science 3(3), 109–124 (1999)
19. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
20. Thomas, W.: Elements of an automata theory over partial orders. In: POMIV 1996. DIMACS, vol. 29. AMS, Providence (1996)
21. Tzevelekos, N.: Fresh-register automata. In: Ball, T., Sagiv, M. (eds.) POPL 2011, pp. 295–306. ACM, New York (2011)

Advanced Ramsey-Based Büchi Automata Inclusion Testing*

Parosh Aziz Abdulla[1], Yu-Fang Chen[2], Lorenzo Clemente[3], Lukáš Holík[1,4],
Chih-Duo Hong[2], Richard Mayr[3], and Tomáš Vojnar[4]

[1] Uppsala University
[2] Academia Sinica
[3] University of Edinburgh
[4] Brno University of Technology

Abstract. Checking language inclusion between two nondeterministic Büchi automata $\mathcal{A}$ and $\mathcal{B}$ is computationally hard (PSPACE-complete). However, several approaches which are efficient in many practical cases have been proposed. We build on one of these, which is known as the *Ramsey-based approach*. It has recently been shown that the basic Ramsey-based approach can be drastically optimized by using powerful subsumption techniques, which allow one to prune the search-space when looking for counterexamples to inclusion. While previous works only used subsumption based on set inclusion or forward simulation on $\mathcal{A}$ and $\mathcal{B}$, we propose the following new techniques: (1) A larger subsumption relation based on a combination of backward and forward simulations on $\mathcal{A}$ and $\mathcal{B}$. (2) A method to additionally use forward simulation *between* $\mathcal{A}$ and $\mathcal{B}$. (3) Abstraction techniques that can speed up the computation and lead to early detection of counterexamples. The new algorithm was implemented and tested on automata derived from real-world model checking benchmarks, and on the Tabakov-Vardi random model, thus showing the usefulness of the proposed techniques.

1 Introduction

Checking inclusion between finite-state models is a central problem in automata theory. First, it is an intriguing theoretical problem. Second, it has many practical applications. For example, in the automata-based approach to model-checking [19], both the system and the specification are represented as finite-state automata, and the model-checking problem reduces to testing whether any behavior of the system is allowed by the specification, i.e., to a language inclusion problem.

We consider language inclusion for Büchi automata (BA), i.e., automata over infinite words. While checking language inclusion between nondeterministic BA is computationally hard (PSPACE-complete [13]), much effort has been devoted to devising approaches that can solve as many practical cases as possible. A naïve approach to

* This work was supported by Royal Society grant JP080268, the UPMARC project, the CONNECT project, National Science Council of Taiwan project no. 99-2218-E-001-002-MY3, the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the BUT FIT project FIT-S-11-1.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 187–202, 2011.

language inclusion between BA $\mathcal{A}$ and $\mathcal{B}$ would first complement the latter into a BA $\mathcal{B}^c$, and then check emptiness of $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}^c)$. The problem is that $\mathcal{B}^c$ is in general exponentially larger than $\mathcal{B}$. Yet, one can determine whether $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}^c) \neq \emptyset$ by only looking at some "small" portion of $\mathcal{B}^c$. The *Ramsey-based approach* [16,9,10] gives a recipe for doing this. It is a descendant of Büchi's original BA complementation procedure, which uses the infinite Ramsey theorem in its correctness proof.

The essence of the Ramsey-based approach for checking language inclusion between $\mathcal{A}$ and $\mathcal{B}$ lies in the notion of *supergraph*, which is a data-structure representing a class of finite words sharing similar behavior in the two automata. Ramsey-based algorithms contain (i) an initialization phase where a set of supergraph seeds are identified, (ii) a search loop in which supergraphs are iteratively generated by composition with seeds, and (iii) a test operation where pairs of supergraphs are inspected for the existence of a counterexample. Intuitively, this counterexample has the form of an infinite ultimately periodic word $w_1(w_2)^\omega \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B}^c)$, where one supergraph witnesses the prefix and the other the loop. While supergraphs themselves are small, and the test in (iii) can be done efficiently, the limiting factor in the basic algorithm lies in the exponential number of supergraphs that need to be generated. Therefore, a crucial challenge in the design of Ramsey-based algorithms is to limit the supergraphs explosion problem. This can be achieved by carefully designing certain *subsumption relations* [10,1], which allow one to safely discard subsumed supergraphs, thus reducing the search space. Moreover, methods based on minimizing supergraphs [1] by pruning their structure can further reduce the search space, and improve the complexity of (iii) above.

This paper contributes to the Ramsey-based approach to language inclusion in several ways. (1) We define a new subsumption relation based on both *forward* and *backward simulation* within the two automata. Our notion generalizes the subset-based subsumption of [10] and the forward simulation-based subsumption of [1]. (2) On a similar vein, we improve minimization of supergraphs by employing forward and backward simulation for minimizing supergraphs. (3) We introduce a method of exploiting forward simulation *between* the two automata, while previously only simulations internal to each automaton have been considered. (4) Finally, we provide a method to speed up the tests performed on supergraphs by grouping similar supergraphs together in a combined representation and extracting more abstract test-relevant information from it.

The correctness of the combined use of forward and backward simulation turns out to be far from trivial, requiring suitable generalizations of the basic notions of composition and test. Technically, we consider generalized composition and test operations where *jumps* are allowed—a jump occurring between states related by backward simulation. The proofs justifying the use of jumping composition and test, which can be found in [2], are much more involved than in previous works.

We have implemented our techniques and tested them on BA derived from a set of real-world model checking benchmarks [15] and from the Tabakov-Vardi random model [18]. The new technique is able to finish many of the difficult problem instances in minutes while the algorithm of [1] cannot finish them even in one day. All our benchmarks, the source code, and the executable of our implementation are available at `http://www.languageinclusion.org/CONCUR2011`. Due to limited space, some details of the experiments are deferred to [2].

Related Work. An alternative approach to language inclusion for BA is given by *rank-based* methods [14], which provide a different complementation procedure based on a rank-based analysis of rejecting runs. This approach is orthogonal to Ramsey-based algorithms. In fact, while rank-based approaches have a better worst-case complexity, Ramsey-based approaches can still perform better on many examples [10]. A subsumption-based algorithm for the rank-based approach has been given in [5]. Subsumption techniques have recently been considered also for automata over *finite words* [20,3].

2 Preliminaries

A *Büchi Automaton (BA)* $\mathcal{A}$ is a tuple $(\Sigma, Q, I, F, \delta)$ where Σ is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a non-empty set of *initial* states, $F \subseteq Q$ is a set of *accepting* states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. A *run* of $\mathcal{A}$ on a word $w = \sigma_1\sigma_2 \ldots \in \Sigma^\omega$ *starting* in a state $q_0 \in Q$ is an infinite sequence $q_0q_1\ldots$ s.t. $(q_{j-1}, \sigma_j, q_j) \in \delta$ for all $j > 0$. The run is *accepting* iff $q_i \in F$ for infinitely many i. The *language of* $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{w \mid \mathcal{A}$ has an accepting run on w starting from some $q_0 \in I\}$.

A *path* in $\mathcal{A}$ on a finite word $w = \sigma_1 \ldots \sigma_n \in \Sigma^+$ is a finite sequence $q_0q_1 \ldots q_n$ s.t. $\forall 0 < j \leq n: \ (q_{j-1}, \sigma_j, q_j) \in \delta$. The path is *accepting* iff $\exists 0 \leq i \leq n: \ q_i \in F$. For any $p, q \in Q$, let $p \overset{w}{\rightsquigarrow}_F q$ iff there is an accepting path on w from p to q, and $p \overset{w}{\rightsquigarrow} q$ iff there is a (not necessarily accepting) path on w from p to q.

A *forward simulation* [4] on $\mathcal{A}$ is a relation $R \subseteq Q \times Q$ such that pRr only if $p \in F \implies r \in F$, and for every transition $(p, \sigma, p') \in \delta$, there exists a transition $(r, \sigma, r') \in \delta$ s.t. $p'Rr'$. A *backward simulation* on $\mathcal{A}$ ([17], where it is called *reverse simulation*) is a relation $R \subseteq Q \times Q$ s.t. $p'Rr'$ only if $p' \in F \implies r' \in F$, $p' \in I \implies r' \in I$, and for every $(p, \sigma, p') \in \delta$, there exists $(r, \sigma, r') \in \delta$ s.t. pRr. Note that this notion of backward simulation is stronger than the usual finite-word automata version, as we require not only compatibility w.r.t. initial states, but also w.r.t. final states. It can be shown that there exists a unique maximal forward simulation denoted by $\preceq_{\mathsf{f}}^{\mathcal{A}}$ and also a unique maximal backward simulation denoted by $\preceq_{\mathsf{b}}^{\mathcal{A}}$, which are both polynomial-time computable preorders [11]. We drop the superscripts when no confusion can arise.

In the rest of the paper, we fix two BA $\mathcal{A} = (\Sigma, Q_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \delta_\mathcal{A})$ and $\mathcal{B} = (\Sigma, Q_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \delta_\mathcal{B})$. The *language inclusion problem* consists in deciding whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. It is well known that deciding language inclusion is PSPACE-complete [13], and that forward simulations [4] can be used as an underapproximation thereof. Here, we focus on deciding language inclusion precisely, by giving a complete algorithm.

3 Ramsey-Based Language Inclusion Testing

Abstractly, the Ramsey-based approach for checking $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ consists in building a *finite* set $\mathcal{X} \subseteq 2^{\mathcal{L}(\mathcal{A})}$ of fragments of $\mathcal{L}(\mathcal{A})$ satisfying the following two properties:

α (covering) $\bigcup \mathcal{X} = \mathcal{L}(\mathcal{A})$.
β (dichotomy) For all $X \in \mathcal{X}$, either $X \subseteq \mathcal{L}(\mathcal{B})$ or $X \cap \mathcal{L}(\mathcal{B}) = \emptyset$.

The covering property ensures that the considered fragments cover $\mathcal{L}(\mathcal{A})$, and the dichotomy property states that the fragments are either entirely in $\mathcal{L}(\mathcal{B})$ or disjoint from

$\mathcal{L}(\mathcal{B})$. Moreover, the fragments are chosen such that they can be effectively generated and such that their inclusion in $\mathcal{L}(B)$ is easy to test. During the generation of the fragments, it then suffices to test each of them for inclusion in $\mathcal{L}(\mathcal{B})$. If this is the case, the inclusion $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ holds. Otherwise, there is a fragment $X \subseteq \mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{B})$ s.t. every ω-word $w \in X$ is a counterexample to the inclusion of $\mathcal{L}(\mathcal{A})$ in $\mathcal{L}(\mathcal{B})$.

We now instantiate the above described abstract algorithm by giving primitives for representing fragments of $\mathcal{L}(\mathcal{A})$ satisfying the conditions of covering and dichotomy. Much like in [9], we introduce the notion of *arcs* for satisfying Condition α, the notion of *graphs* for Condition β, and then we put them together in the notion of *supergraphs* as to satisfy $\alpha + \beta$. Then, we explain that supergraphs can be effectively generated and that the fragment languages they represent can be easily tested for inclusion in $\mathcal{L}(B)$.

Condition α*: Edges and Properness.* An *edge* $\langle p,a,q \rangle$ is an element of $E_{\mathcal{A}} = Q_{\mathcal{A}} \times \{0,1\} \times Q_{\mathcal{A}}$. Its language $\mathcal{L}\langle p,a,q \rangle \subseteq \Sigma^+$ contains a word $w \in \Sigma^+$ iff either (1) $a = 1$ and $p \overset{w}{\rightsquigarrow}_F q$, or (2) $a = 0$, $p \overset{w}{\rightsquigarrow} q$, but not $p \overset{w}{\rightsquigarrow}_F q$. A pair of edges $(\langle q_1,a,q_2 \rangle, \langle q_3,b,q_4 \rangle)$ is *proper* iff $q_1 \in I_{\mathcal{A}}$, $q_2 = q_3 = q_4$, and $b = 1$. A pair of edges (x,y) can be used to encode the ω-language $Y_{xy} = \mathcal{L}(x) \cdot (\mathcal{L}(y))^{\omega}$. Clearly, if the pair of edges is proper, $Y_{xy} \subseteq \mathcal{L}(\mathcal{A})$. Intuitively, the language of a proper pair of edges contains words accepted by lasso-shaped accepting runs starting from q_1 and looping through q_2. Furthermore, it is clearly the case that one can completely cover $\mathcal{L}(\mathcal{A})$ by languages Y_{xy}. Thus, the set $\mathcal{X}_{\mathsf{edges}} = \{Y_{xy} \mid (x,y) \text{ is proper } \}$ satisfies Condition α.

Condition β*: Graphs.* A *graph* g is a subset of edges from $E_{\mathcal{B}} = Q_{\mathcal{B}} \times \{0,1\} \times Q_{\mathcal{B}}$ containing at most one edge for every pair of states. Its language is defined as the set of words over Σ^+ that are consistent with all the edges of the graph. Namely, $w \in \mathcal{L}(g)$ iff, for any pair of states $p,q \in Q_{\mathcal{B}}$, either (1) $p \overset{w}{\rightsquigarrow}_F q$ and $\langle p,1,q \rangle \in g$, (2) $p \overset{w}{\rightsquigarrow} q$, $\neg(p \overset{w}{\rightsquigarrow}_F q)$, and $\langle p,0,q \rangle \in g$, or (3) $\neg(p \overset{w}{\rightsquigarrow} q)$ and there is no edge in g of the form $\langle p,a,q \rangle$. Intuitively, the language of a graph consists of words that all connect any chosen pair of states in the same way (i.e., possibly through an accepting state, through non-accepting states only, or not at all). Let G be the set of all graphs. Not all graphs, however, contain meaningful information, e.g., a graph may contain an edge between states not reachable from each other. Such contradictory information makes the language of a graph empty. Define $G^f = \{g \in G \mid \mathcal{L}(g) \neq \emptyset\}$ as the set of graphs with non-empty languages.

It can be shown that the languages of graphs partition Σ^+. Like with edges, a pair of graphs (g,h) can be used to encode the ω-language $Y_{gh} = \mathcal{L}(g) \cdot (\mathcal{L}(h))^{\omega}$. Intuitively, the pair of graphs g, h encodes *all* runs in $\mathcal{B}$ over the ω-words in Y_{gh}. These runs can be obtained by selecting an edge from g and possibly multiple edges from h that can be connected by their entry/exit states to form a lasso. Since the words in the language of graphs have the same power for connecting states, accepting runs exist for all elements of Y_{gh} or for none of them. The following lemma [16,9,10] shows that the set $\mathcal{X}_{\mathsf{graphs}} = \{Y_{gh} \mid g,h \in G^f\}$ satisfies Condition β.

Lemma 1. *For graphs g,h, either $Y_{gh} \subseteq \mathcal{L}(\mathcal{B})$ or $Y_{gh} \cap \mathcal{L}(\mathcal{B}) = \emptyset$.*

Condition $\alpha + \beta$*: Supergraphs.* We combine edges and graphs to build more complex objects satisfying, at the same time, Conditions α and β. A *supergraph* is a pair

$\mathbf{g} = \langle x, g\rangle \in E_{\mathcal{A}} \times G$.[1] A supergraph is only meaningful if the information in the edge-part is consistent with that in the graph-part. To this end, let $\mathcal{L}(\mathbf{g}) = \mathcal{L}(x) \cap \mathcal{L}(g)$ and let $S^f = \{\mathbf{g} \mid \mathcal{L}(\mathbf{g}) \neq \emptyset\}$ be the set of supergraphs with non-empty language. For two supergraphs $\mathbf{g} = \langle x, g\rangle$ and $\mathbf{h} = \langle y, h\rangle$, the pair $(\mathbf{g}, \mathbf{h})$ is *proper* if the edge-pair (x, y) is proper. Let $Y_{\mathbf{gh}} = \mathcal{L}(\mathbf{g}) \cdot (\mathcal{L}(\mathbf{h}))^{\omega}$. Notice that $Y_{\mathbf{gh}} \subseteq Y_{xy} \cap Y_{gh}$. Therefore, since Y_{gh} satisfies Condition β, so does $Y_{\mathbf{gh}} \subseteq Y_{gh}$. For Condition α, we show that Y_{xy} can be covered by a family of languages of the form $Y_{\langle x,g\rangle\langle y,h\rangle}$. This is sound since $Y_{\langle x,g\rangle\langle y,h\rangle} \subseteq Y_{xy}$ for *any* g, h. Completeness follows from the lemma below, stating that every word $w \in Y_{xy}$ lies in a set of the form $Y_{\langle x,g\rangle\langle y,h\rangle}$. It is proved by a Ramsey-based argument.

Lemma 2. *For proper edges (x, y) and $w \in Y_{xy}$, there exist graphs g, h s.t. $w \in Y_{\langle x,g\rangle\langle y,h\rangle}$.*

Thus, Y_{xy} can be covered by $\mathcal{X}_{xy} = \{Y_{\mathbf{gh}} \mid g, h \in G^f, \mathbf{g} = \langle x, g\rangle, \mathbf{h} = \langle y, h\rangle\}$. Since $\mathcal{X}_{\mathsf{edges}}$ covers $\mathcal{L}(\mathcal{A})$, and each $Y_{xy} \in \mathcal{X}_{\mathsf{edges}}$ can be covered by $\mathcal{X}_{xy}$, it follows that $\mathcal{X} = \{Y_{\mathbf{gh}} \mid \mathbf{g}, \mathbf{h} \in S^f, (\mathbf{g}, \mathbf{h}) \text{ is proper}\}$ covers $\mathcal{L}(\mathcal{A})$. Thus, $\mathcal{X}$ fulfills $\alpha + \beta$.

Generating and Testing Supergraphs. While supergraphs in S^f are a convenient syntactic object for manipulating languages in $\mathcal{X}$, testing that a given supergraph has non-empty language is expensive (PSPACE-complete). In [12], this problem is elegantly solved by introducing a natural notion of *composition* of supergraphs, which preserves non-emptiness: The idea is to start with a (small) set of supergraphs which have non-empty language by construction, and then to obtain S^f by composing supergraphs until no more supergraphs can be generated.

For a BA $\mathcal{C}$ and a symbol $\sigma \in \Sigma$, let $E^{\sigma}_{\mathcal{C}} = \{\langle p, a, q\rangle \mid (p, \sigma, q) \in \delta_{\mathcal{C}}, (a = 1 \iff p \in F \vee q \in F)\}$ be the set of edges induced by σ. The initial seed for the procedure is given by *one-letter supergraphs* in $S^1 = \bigcup_{\sigma\in\Sigma}\{(x, E^{\sigma}_{\mathcal{B}}) \mid x \in E^{\sigma}_{\mathcal{A}}\}$. Notice that $S^1 \subseteq S^f$ by construction. Next, two edges $x = \langle p, a, q\rangle$ and $y = \langle q', b, r\rangle$ are *composable* iff $q = q'$. For composable edges x and y, let $x; y = \langle p, \max(a, b), r\rangle$. Further, the *composition* $g; h$ of graphs g and h is defined as follows: $\langle p, c, r\rangle \in g; h$ iff there is a state q s.t. $\langle p, a, q\rangle \in g$ and $\langle q, b, r\rangle \in h$, and $c = \max_{q\in Q}\{\max(a, b) \mid \langle p, a, q\rangle \in g, \langle q, b, r\rangle \in h\}$. Then, supergraphs $\mathbf{g} = \langle x, g\rangle$ and $\mathbf{h} = \langle y, h\rangle$ are *composable* iff $\langle x, y\rangle$ are composable, and their *composition* is the supergraph $\mathbf{g}; \mathbf{h} = \langle x; y, g; h\rangle$. Notice that S^f is closed under composition, i.e., $\mathbf{g}, \mathbf{h} \in S^f \implies \mathbf{g}; \mathbf{h} \in S^f$. Composition is also *complete* for generating S^f:

Lemma 3. *[1] A supergraph $\mathbf{g}$ is in S^f iff $\exists \mathbf{g}_1, \ldots, \mathbf{g}_n \in S^1$ such that $\mathbf{g} = \mathbf{g}_1; \ldots; \mathbf{g}_n$.*

Now that we have a method for generating all relevant supergraphs, we need a way of checking inclusion of (supergraphs representing) fragments of $\mathcal{L}(\mathcal{A})$ in $\mathcal{L}(\mathcal{B})$. Let $(\mathbf{g}, \mathbf{h})$ be a (proper) pair of supergraphs. By the dichotomy property, $Y_{\mathbf{gh}} \subseteq \mathcal{L}(\mathcal{B})$ iff $Y_{\mathbf{gh}} \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$. We test the latter condition by the so-called *double graph test*: For a pair of supergraphs $(\mathbf{g}, \mathbf{h})$, $DGT(\mathbf{g}, \mathbf{h})$ iff, whenever $(\mathbf{g}, \mathbf{h})$ is proper, then $LFT(g, h)$. Here, LFT is the so-called *lasso-finding test*: Intuitively, LFT checks for a lasso with a handle in g and an accepting loop in h. Formally, $LFT(g, h)$ iff there is an edge $\langle p, a_0, q_0\rangle \in g$ and an infinite sequence of edges $\langle q_0, a_1, q_1\rangle$, $\langle q_1, a_2, q_2\rangle, \ldots \in h$ s.t. $p \in I$ and $a_j = 1$ for infinitely many j's.

[1] The definition of supergraph given here is slightly different from [9,1], where the edge-part is just a pair of states (p, q). Having labels allows us to give a notion of properness which does not require to have $q \in F$.

Lemma 4. *[1] $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff for all $\mathbf{g},\mathbf{h} \in S^f$, $DGT(\mathbf{g},\mathbf{h})$.*

Basic Algorithm [9]. The basic algorithm for checking inclusion enumerates all supergraphs from S^f by extending supergraphs on the right by one-letter supergraphs from S^1; that is, a supergraph $\mathbf{g}$ generates new supergraphs by selecting some $\mathbf{h} \in S^1$ and building $\mathbf{g};\mathbf{h}$. Then, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ holds iff all the generated pairs pass the DGT.

Intuitively, the algorithm processes all lasso-shaped runs that can be used to accept some words in $\mathcal{A}$. These runs are represented by the edge-parts of proper pairs of generated supergraphs. For each such run of $\mathcal{A}$, the algorithm uses LFT to test whether there is a corresponding accepting run of $\mathcal{B}$ among *all* the possible runs of $\mathcal{B}$ on the words represented by the given pair of supergraphs. These latter runs are encoded by the graph-parts of the respective supergraphs.

4 Optimized Language Inclusion Testing

The basic algorithm of Section 3 is wasteful for two reasons. First, not all edges in the graph component of a supergraph are needed to witness a counterexample to inclusion: Hence, we can reduce a graph by keeping only a certain subset of its edges (Optimization 1). Second, not all supergraphs need to be generated and tested: We show a method which safely allows the algorithm to discard certain supergraphs (Optimization 2). Both optimizations rely on various notions of *subsumption*, which we introduce next.

Given two edges $x = \langle p,a,q\rangle$ and $y = \langle r,b,s\rangle$, we say that y *subsumes* x, written $x \sqsubseteq y$, if $p = r$, $a \leq b$, and $q = s$; that x *forward-subsumes* y, written $x \sqsubseteq_{\mathsf{f}} y$, if $p = r$, $a \leq b$, and $q \preceq_{\mathsf{f}} s$; that x *backward-subsumes* y, written $x \sqsubseteq_{\mathsf{b}} y$, if $p \preceq_{\mathsf{b}} r$, $a \leq b$, and $q = s$; and that x *forward-backward-subsumes* y, written $x \sqsubseteq_{\mathsf{fb}} y$, if $p \preceq_{\mathsf{b}} r$, $a \leq b$, and $q \preceq_{\mathsf{f}} s$. We lift all the notions of subsumption to graphs: For any $z \in \{\mathsf{f},\mathsf{b},\mathsf{fb},\llcorner\!\lrcorner\}$ and for graphs g and h, let $g \sqsubseteq_z h$ iff, for every edge $x \in g$, there exists an edge $y \in h$ s.t. $x \sqsubseteq_z y$. Since the simulations $\preceq_{\mathsf{f}}$ and $\preceq_{\mathsf{b}}$ are preorders, all subsumptions are preorders. We define backward and forward-backward subsumption equivalence as $\simeq_{\mathsf{b}} = \sqsubseteq_{\mathsf{b}} \cap \sqsubseteq_{\mathsf{b}}^{-1}$ and $\simeq_{\mathsf{fb}} = \sqsubseteq_{\mathsf{fb}} \cap \sqsubseteq_{\mathsf{fb}}^{-1}$, respectively.

4.1 Optimization 1: Minimization of Supergraphs

The first optimization concerns the structure of individual supergraphs. Let $\mathbf{g} = \langle x,g\rangle \in S$ be a supergraph, with g its graph-component. We minimize g by deleting edges therein which are subsumed by $\sqsubseteq_{\mathsf{fb}}$-larger ones. That is, whenever we have $x \sqsubseteq_{\mathsf{fb}} y$ for two edges $x,y \in g$, we remove x and keep y. Intuitively, subsumption-larger arcs contribute more to the capability of representing lassoes since their right and left endpoints are $\preceq_{\mathsf{f}}$/$\preceq_{\mathsf{b}}$-larger, respectively, and have therefore a richer choice of possible futures and pasts. Subsumption smaller arcs are thus redundant, and removing them does not change the capability of g to represent lassoes in $\mathcal{B}$. Formally, we define a minimization operation *Min* mapping a supergraph $\mathbf{g} = \langle x,g\rangle$ to its minimized version $Min(\mathbf{g}) = \langle x, Min(g)\rangle$ where $Min(g)$ is the minimization applied to the graph-component.[2]

[2] In [1], we used $\sqsubseteq_{\mathsf{f}}$ for minimization. The theory allowing the use of $\sqsubseteq_{\mathsf{fb}}$ is significantly more involved, but as as shown in Section 8, the use of $\sqsubseteq_{\mathsf{fb}}$ turns out to be much more advantageous.

Definition 1. *For two graphs g and h, let $g \leqslant h$ iff (1) $g \sqsubseteq h$ and (2) $h \sqsubseteq_{\mathsf{fb}} g$. For supergraphs $\mathbf{g} = \langle x, g\rangle$ and $\mathbf{h} = \langle y, h\rangle$, let $\mathbf{g} \leqslant \mathbf{h}$ iff $x = y$ and $g \leqslant h$. A* minimization *of graphs is any function Min such that, for any graph h, $Min(h) \leqslant h$.*

Point 1 in the definition of $\leqslant$ allows some edges to be erased or their label decreased. Point 2 states that only subsumed arcs can be removed or have their label decreased. Note also that, clearly, $Min(\mathbf{h}) \leqslant \mathbf{h}$ holds for any supergraph $\mathbf{h}$. Finally, note that Min is not uniquely determined: First, there are many candidates satisfying $Min(h) \leqslant h$. Yet, an implementation will usually remove a maximal number of edges to keep the size of graphs to a minimum. Second, even if we required $Min(h)$ to be a $\leqslant$-smallest element (i.e., no further edge can be removed), the minimization process might encounter $\sqsubseteq_{\mathsf{fb}}$-equivalent edges, and in this case, we do not specify which ones get removed. Therefore, we prove correctness for any minimization satisfying $Min(h) \leqslant h$.

Intuitively, a minimized supergraph $\mathbf{g}$ can be seen as a small *representative* of all supergraphs $\mathbf{h} \in G^f$ with $\mathbf{g} \leqslant \mathbf{h}$, and of all the fragments of $\mathcal{L}(A)$ encoded by them. Using representatives allows us to deal with a smaller number of smaller supergraphs. We now explain how (sufficiently many) representatives encoding fragments of $\mathcal{L}(\mathcal{A})$ can be *generated* and *tested* for inclusion in $\mathcal{L}(\mathcal{B})$.

Generating Representatives of Supergraphs. We need to create a representative of each supergraph in S^f by composing representatives only. Let $\mathbf{g} = \langle x, g\rangle$ and $\mathbf{h} = \langle y, h\rangle$ be two composable supergraphs, representing $\mathbf{g}' = \langle x, g'\rangle$ and $\mathbf{h}' = \langle y, h'\rangle$, respectively. If graph composition were $\leqslant$-monotone, i.e., $\mathbf{g};\mathbf{h} \leqslant \mathbf{g}';\mathbf{h}'$, then we would be done. However, graph composition is not monotone: The reason is that some composable edges $e \in g'$ and $f \in h'$ may be erased by minimization, and be represented by some $\hat{e} \in g$ and $\hat{f} \in h$ instead, with $e \sqsubseteq_{\mathsf{fb}} \hat{e}$ and $f \sqsubseteq_{\mathsf{fb}} \hat{f}$. But now, $\hat{e}$ and $\hat{f}$ are not necessarily composable anymore. Thus, $\mathbf{g};\mathbf{h} \not\leqslant \mathbf{g}';\mathbf{h}'$. We solve this problem in two steps: We allow composition to jump to $\preceq_{\mathsf{b}}$-larger states (Def. 2), and relax the notion of representative (Def. 3).

Definition 2. *Given graphs $g, h \in G$, their* jumping composition *$g \mathbin{⨟_{\mathsf{b}}} h$ contains an edge $\langle p, c, r\rangle \in g \mathbin{⨟_{\mathsf{b}}} h$ iff there are edges $\langle p, a, q\rangle \in g$, $\langle q', b, r\rangle \in h$ s.t. $q \preceq_{\mathsf{b}} q'$, and $c = \max_{q,q'}\{\max(a,b) \mid \langle p,a,q\rangle \in g, \langle q',b,r\rangle \in h, q \preceq_{\mathsf{b}} q'\}$. For two composable supergraphs $\mathbf{g} = \langle x, g\rangle$ and $\mathbf{h} = \langle y, h\rangle$, let $\mathbf{g} \mathbin{⨟_{\mathsf{b}}} \mathbf{h} = \langle x;y, g \mathbin{⨟_{\mathsf{b}}} h\rangle$.*

Jumping composition alone does not yet give the required monotonicity property. The problem is that $\mathbf{g} \mathbin{⨟_{\mathsf{b}}} \mathbf{h}$ is not necessarily a minimized version of $\mathbf{g}';\mathbf{h}'$, but it is only a minimized version of something $\simeq_{\mathsf{b}}$-equivalent to $\mathbf{g}';\mathbf{h}'$. This leads us to the following more liberal notion of representatives, which is based on $\leqslant$ modulo the equivalence $\simeq_{\mathsf{b}}$, and for which Lemma 5 proves the required monotonicity property.

Definition 3. *A graph $g \in G$ is a* representative *of a graph $h \in G^f$, denoted $g \trianglelefteq h$, iff there exists $\bar{h} \in G$ such that $g \leqslant \bar{h} \simeq_{\mathsf{b}} h$. For supergraphs $\mathbf{g} = \langle x, g\rangle, \mathbf{h} = \langle y, h\rangle \in S$, we say that $\mathbf{g}$ is a* representative *of $\mathbf{h}$, written $\mathbf{g} \trianglelefteq \mathbf{h}$, iff $x = y$ and $g \trianglelefteq h$. Let $S^R = \{\mathbf{g} \mid \exists \mathbf{h} \in S^f.\ \mathbf{g} \trianglelefteq \mathbf{h}\}$ be the set of representatives of supergraphs.*

Lemma 5. *For supergraphs $\mathbf{g}, \mathbf{h} \in S^R$ and $\mathbf{g}', \mathbf{h}' \in S^f$, if $\mathbf{g} \trianglelefteq \mathbf{g}'$, $\mathbf{h} \trianglelefteq \mathbf{h}'$ and $\mathbf{g}', \mathbf{h}'$ are composable, then $\mathbf{g}, \mathbf{h}$ are composable and $\mathbf{g} \mathbin{⨟_{\mathsf{b}}} \mathbf{h} \trianglelefteq \mathbf{g}';\mathbf{h}'$ and $\mathbf{g} \mathbin{⨟_{\mathsf{b}}} \mathbf{h} \in S^R$.*

Lemma 6. *Let* $\mathbf{f} \in S$, $\mathbf{g} \in S^R$, *and* $\mathbf{h} \in S^f$. *If* $\mathbf{f} \leqslant \mathbf{g}$ *and* $\mathbf{g} \trianglelefteq \mathbf{h}$, *then* $\mathbf{f} \trianglelefteq \mathbf{h}$ *(and thus* $\mathbf{f} \in S^R$*). In particular, the statement holds when* $\mathbf{f} = Min(\mathbf{g})$.

Lemmas 5, 6, and 3 imply that creating supergraphs by $\mathbin{⨾}_{\mathsf{b}}$-composing representatives, followed by further minimization, suffices to create a representative of each supergraph in S^f. This solves the problem of generating representatives of supergraphs.

Weak Properness and Relaxed DGT. We now present a relaxed DGT proposed in [1], which we further improve below. The idea is to weaken the properness condition in order to allow more pairs of supergraphs to be eligible for LFT on their graph part. This may lead to a quicker detection of a counterexample. Weak properness is sound since it still produces fragments $Y_{\mathbf{gh}} \subseteq \mathcal{L}(\mathcal{A})$ as required by Condition α. Completeness is guaranteed since properness implies weak properness.

Definition 4. *(adapted from [1]) A pair of edges* $(\langle p,a,q\rangle, \langle r,b,s\rangle)$ *is* weakly proper *iff* $p \in I_{\mathcal{A}}$, $r \preceq_{\mathsf{f}} q$, $r \preceq_{\mathsf{f}} s$, *and* $b = 1$,[3] *and a pair of supergraphs* $(\mathbf{g} = \langle x,g\rangle, \mathbf{h} = \langle y,h\rangle)$ *is weakly proper when* (x,y) *is weakly proper. Supergraphs* $\mathbf{g},\mathbf{h}$ *pass the* relaxed double graph test, *denoted* $RDGT(\mathbf{g},\mathbf{h})$, *iff whenever* $(\mathbf{g},\mathbf{h})$ *is weakly proper, then* $LFT(g,h)$.

Lemma 7. *[1]* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ *iff for all* $\mathbf{g},\mathbf{h} \in S^f$, $RDGT(\mathbf{g},\mathbf{h})$.

Testing Representatives of Supergraphs. We need a method for testing inclusion in $\mathcal{L}(\mathcal{B})$ of the fragments of $\mathcal{L}(\mathcal{A})$ encoded by representatives of supergraphs that is equivalent to testing inclusion of fragments of $\mathcal{L}(\mathcal{A})$ encoded by the represented supergraphs. As with composition, minimization is not compatible with such testing since edges needed to find loops may be erased during the minimization process. Technically, this results in the LFT (and therefore RDGT) not being $\trianglelefteq$-monotone. Therefore, we generalize the LFT by allowing jumps to $\preceq_{\mathsf{b}}$-larger states, in a similar way as with $\mathbin{⨾}_{\mathsf{b}}$. Lemma 8 establishes the required monotonicity property.

Definition 5. *A pair of graphs* (g,h) *passes the* jumping lasso-finding test, *denoted* $LFT_{\mathsf{b}}(g,h)$, *iff there is an edge* $\langle p,a_0,q_0\rangle$ *in* g *and an infinite sequence of edges* $\langle q_0',a_1,q_1\rangle$, $\langle q_1',a_2,q_2\rangle,\ldots$ *in* h *s.t.* $p \in I$, $q_i \preceq_{\mathsf{b}} q_i'$ *for all* $i \geq 0$, *and* $a_j = 1$ *for infinitely many* j*'s. A pair of supergraphs* $(\mathbf{g},\mathbf{h})$ *passes the* jumping relaxed double graph test, *denoted* $RDGT_{\mathsf{b}}(\mathbf{g},\mathbf{h})$, *iff whenever* $(\mathbf{g},\mathbf{h})$ *is weakly proper, then* $LFT_{\mathsf{b}}(g,h)$.

Lemma 8. *For any* $\mathbf{g},\mathbf{h} \in S^R$ *and* $\mathbf{g}',\mathbf{h}' \in S^f$ *such that* $\mathbf{g} \trianglelefteq \mathbf{g}'$ *and* $\mathbf{h} \trianglelefteq \mathbf{h}'$, *it holds that* $RDGT_{\mathsf{b}}(\mathbf{g},\mathbf{h}) \Longleftrightarrow RDGT(\mathbf{g}',\mathbf{h}')$.

Algorithm with Minimization. By Lemma 8, $\mathrm{RDGT}_{\mathsf{b}}$ on representatives is equivalent to RDGT on the represented supergraphs. Together with Lemma 7, this means that it is enough to generate a representative of each supergraph from S^f, and test all pairs of the generated supergraphs with $\mathrm{RDGT}_{\mathsf{b}}$. Thus, we have obtained a modification of the basic algorithm which starts from minimized 1-letter supergraphs in $Min(S^1) = \{Min(\mathbf{g}) \mid \mathbf{g} \in S^1\}$, and constructs new supergraphs by $\mathbin{⨾}_{\mathsf{b}}$-composing already generated supergraphs with $Min(S^1)$ on the right. New supergraphs are further minimized with Min. Inclusion holds iff all pairs of generated supergraphs pass $\mathrm{RDGT}_{\mathsf{b}}$.

[3] We note that instead of testing $r \preceq_{\mathsf{f}} q$, testing inclusion of the languages of the states is sufficient. Furthermore, instead of testing $r \preceq_{\mathsf{f}} s$, one can test for *delayed simulation*, but not for language inclusion. See [2] for details.

4.2 Optimization 2: Discarding Subsumed Supergraphs

The second optimization gives a rule for discarding supergraphs subsumed by some other supergraph. This is safe in the sense that if a subsumed supergraph can yield a counterexample to language inclusion, then also the subsuming one can yield a counterexample. We present an improved version of the subsumption from [1]. The new version uses both $\preceq_{\mathsf{f}}$ and $\preceq_{\mathsf{b}}$ on the $\mathcal{B}$ part of supergraphs instead of $\preceq_{\mathsf{f}}$ only. This allows us to discard significantly more supergraphs than in [1], as illustrated in Section 8.

Definition 6. *We say that a supergraph* $\mathbf{g} = \langle x, g \rangle$ subsumes *a supergraph* $\mathbf{g}' = \langle y, g' \rangle$*, written* $\mathbf{g} \sqsubseteq_{\mathsf{fb}} \mathbf{g}'$*, iff* $y \sqsubseteq_{\mathsf{f}} x$ *and* $g \sqsubseteq_{\mathsf{fb}} g'$*.*

Intuitively, if $y \sqsubseteq_{\mathsf{f}} x$, then x has more power for representing lassoes in $\mathcal{A}$ than y since, by the properties of forward simulation, it has a richer choice of possible forward continuations in $\mathcal{A}$. On the other hand, $g \sqsubseteq_{\mathsf{fb}} g'$ means that g' has more chance of representing lassoes in $\mathcal{B}$ than g: In fact, g' contains edges that have a richer choice of backward continuations (due to the $\preceq_{\mathsf{b}}$ on the left endpoints of the edges) as well as a richer choice of forward continuations (due to the $\preceq_{\mathsf{f}}$ on the right endpoints). Thus, it is more likely for $\mathbf{g}$ than for $\mathbf{g}'$ to lead to a counterexample to language inclusion. This intuition is confirmed by the lemma below, stating the $\sqsubseteq_{\mathsf{fb}}$-monotonicity of RDGT$_{\mathsf{b}}$.

Lemma 9. *For supergraphs* $\mathbf{g}, \mathbf{h} \in S^R$ *and* $\mathbf{g}', \mathbf{h}' \in S$*, if* $\mathbf{g} \sqsubseteq_{\mathsf{fb}} \mathbf{g}'$ *and* $\mathbf{h} \sqsubseteq_{\mathsf{fb}} \mathbf{h}'$*, then* $RDGT_{\mathsf{b}}(\mathbf{g}, \mathbf{h}) \Rightarrow RDGT_{\mathsf{b}}(\mathbf{g}', \mathbf{h}')$*.*

Therefore, no counterexample is lost by testing only $\sqsubseteq_{\mathsf{fb}}$-smaller supergraphs. To show that we can completely discard $\sqsubseteq_{\mathsf{fb}}$-larger supergraphs, we need to show that subsumption is compatible with composition, i.e., that descendants of larger supergraphs are (eventually) subsumed by descendants of smaller ones. Ideally, we would achieve this by showing the following more general fact: For two composable representatives $\mathbf{g}', \mathbf{h}' \in S^R$ that are subsumed by supergraphs $\mathbf{g}$ and $\mathbf{h}$, respectively, the composite supergraph $\mathbf{g} \fatsemi_{\mathsf{b}} \mathbf{h}$ subsumes $\mathbf{g}' \fatsemi_{\mathsf{b}} \mathbf{h}'$. The problem is that subsumption does not preserve composability: Even if $\mathbf{g}', \mathbf{h}'$ are composable, this needs not to hold for $\mathbf{g}, \mathbf{h}$.

We overcome this difficulty by taking into account the specific way supergraphs are generated by the algorithm. Since we only generate new supergraphs by composing old ones on the right with 1-letter minimized supergraphs, we do not need to show that arbitrary composition is $\sqsubseteq_{\mathsf{fb}}$-monotone. Instead, we show that, for representatives $\mathbf{g}, \mathbf{g}' \in S^R$ and a 1-letter minimized supergraph $\mathbf{h}' \in Min(S^1)$, if $\mathbf{g}$ subsumes $\mathbf{g}'$, then there will always be a supergraph $\mathbf{h}$ available which is composable with $\mathbf{g}$ such that $\mathbf{g} \fatsemi_{\mathsf{b}} \mathbf{h}$ subsumes $\mathbf{g}' \fatsemi_{\mathsf{b}} \mathbf{h}'$. Thus, we can safely discard $\mathbf{g}'$ from the rest of the computation.

Lemma 10. *For any* $\mathbf{g}, \mathbf{g}' \in S^R$ *with* $\mathbf{g} \sqsubseteq_{\mathsf{fb}} \mathbf{g}'$ *and* $\mathbf{h}' \in Min(S^1)$ *such that* $\mathbf{g}'$ *and* $\mathbf{h}'$ *are composable, there exists* $\hat{\mathbf{h}} \in Min(S^1)$ *such that for all* $\mathbf{h} \in S^R$ *with* $\mathbf{h} \sqsubseteq_{\mathsf{fb}} \hat{\mathbf{h}}$*,* $\mathbf{g}$ *is composable with* $\mathbf{h}$ *and* $\mathbf{g} \fatsemi_{\mathsf{b}} \mathbf{h} \sqsubseteq_{\mathsf{fb}} \mathbf{g}' \fatsemi_{\mathsf{b}} \mathbf{h}'$*.*

Algorithm with Minimization and Subsumption. We have obtained a modification of the algorithm with minimization. It starts with a subset $Init \subseteq Min(S^1)$ of $\sqsubseteq_{\mathsf{fb}}$-smallest minimized one-letter supergraphs. New supergraphs are generated by $\fatsemi_{\mathsf{b}}$-composition on the right with supergraphs in $Init$, followed by minimization with Min. Generated supergraphs that are $\sqsubseteq_{\mathsf{fb}}$-larger than other generated supergraphs are discarded. The inclusion holds iff all pairs of generated supergraphs that are not discarded pass RDGT$_{\mathsf{b}}$. (An illustration of a run of the algorithm can be found in [2].)

5 Using Forward Simulation Between $\mathcal{A}$ and $\mathcal{B}$

Previously, we showed that some supergraphs can safely be discarded because some $\sqsubseteq_{\mathsf{fb}}$-smaller ones are retained, which preserves the chance to find a counterexample to language inclusion. Our subsumption relation $\sqsubseteq_{\mathsf{fb}}$ is based on forward/backward simulation on $\mathcal{A}$ and $\mathcal{B}$. In order to use forward simulation *between* $\mathcal{A}$ and $\mathcal{B}$, we describe a different reason to discard supergraphs. Generally, supergraphs can be discarded because they can neither find a counterexample to inclusion (i.e., always pass the RDGT) nor generate any supergraph that can find a counterexample. However, the RDGT is asymmetric w.r.t. the left and right supergraph. Thus, a supergraph that is useless (i.e., not counterexample-finding) in the left role is not necessarily useless in the right role (and vice-versa). The following condition C is sufficient for a supergraph to be *useless on the left*. Moreover, C is efficiently computable and compatible with subsumption. Therefore, its use preserves the soundness and completeness of our algorithm.

Definition 7. *For* $\mathbf{g}=\langle\langle p,a,q\rangle,g\rangle \in S$, $\mathsf{C}(\mathbf{g})$ *iff* $p \notin I_{\mathcal{A}} \vee (\exists\langle r,b,s\rangle \in g.\ r \in I_{\mathcal{B}} \wedge q \preceq_{\mathsf{f}}^{\mathcal{AB}} s)$.

The first part $p \notin I_{\mathcal{A}}$ of the condition is obvious because paths witnessing counterexamples to inclusion must start in an initial state. The second part $(\exists\langle r,b,s\rangle \in g.\ r \in I_{\mathcal{B}}, q \preceq_{\mathsf{f}}^{\mathcal{AB}} s)$ uses forward-simulation $\preceq_{\mathsf{f}}^{\mathcal{AB}}$ *between* $\mathcal{A}$ and $\mathcal{B}$ to witness that neither this supergraph nor any other supergraph generated from it will find a counterexample *when used on the left side of the RDGT*. It might still be needed for tests on the right side of the RDGT though. Instead of $\preceq_{\mathsf{f}}^{\mathcal{AB}}$, every relation implying language inclusion would suffice, but (as mentioned earlier) simulation preorder is efficiently computable while inclusion is PSPACE-complete. The following lemma shows the correctness of C.

Lemma 11. $\forall \mathbf{g},\mathbf{h} \in S^R.\ \mathsf{C}(\mathbf{g}) \Rightarrow$ *RDGT*$_{\mathsf{b}}(\mathbf{g},\mathbf{h})$.

C is $\sqsubseteq_{\mathsf{fb}}$-upward-closed and closed w.r.t. right extensions. Hence, it is compatible with subsumption-based pruning of the search space and with the employed incremental construction of supergraphs (namely, satisfaction of the condition is inherited to supergraphs newly generated by right extension with one-letter supergraphs).

Lemma 12. *Let* $\mathbf{g},\mathbf{h} \in S$ *s.t.* $\mathbf{g} \sqsubseteq_{\mathsf{fb}} \mathbf{h}$. *Then* $\mathsf{C}(\mathbf{g}) \Rightarrow \mathsf{C}(\mathbf{h})$.

Lemma 13. *Let* $\mathbf{g} \in S^R$, $\mathbf{h} \in Min(S^1)$ *be composable. Then* $\mathsf{C}(\mathbf{g}) \Rightarrow \mathsf{C}(\mathbf{g} \mathbin{;}_{\mathsf{b}} \mathbf{h})$.

In principle, one could store separate sets of supergraphs for use on the left/right in the RDGT, respectively. However, since all supergraphs need to be used on the right anyway, a simple flag is more efficient. We assign the label L to a supergraph to indicate that it is still useful on the left in the RDGT. If a supergraph satisfies condition C, then the L-label is removed. The algorithm counts the number of stored supergraphs that still carry the L-label. If this number drops to zero, then (1) it will remain zero (by Lemma 13), and

(2) no RDGT will ever find a counterexample: In this case, the algorithm can terminate early and report inclusion. In the special case where forward-simulation holds even between the *initial* states of $\mathcal{A}$ and $\mathcal{B}$, condition C is true for *every* generated supergraph. Thus, all L-labels are removed and the algorithm terminates immediately, reporting inclusion. Of course, condition C can also help in other cases where simulation does not hold between initial states but "more deeply" inside the automata.

The following lemma shows that if some supergraph $\mathbf{g}$ can find a counterexample when used on the left in the RDGT, then at least one of its 1-letter right-extensions can also find a counterexample. Intuitively, the counterexample has the form of a prefix followed by an infinite loop, and the prefix can always be extended by one step. E.g., the infinite words $xy(abc)^\omega$ and $xya(bca)^\omega$ are equivalent. This justifies the optimization in line 15 of our algorithm (cf. [2]).

Lemma 14. *Let* $\mathbf{g},\mathbf{h} \in S^R$. *If* $\neg RDGT_{\mathsf{b}}(\mathbf{g},\mathbf{h})$, *then there exists a* $\sqsubseteq_{\mathsf{fb}}$*-minimal supergraph* $\mathbf{f}$ *in* $Min(S^1)$ *and* $\mathbf{e} \in S^R$ *s.t.* $\neg RDGT_{\mathsf{b}}(\mathbf{g} \mathbin{⨟}_{\mathsf{b}} \mathbf{f},\mathbf{e})$.[4]

6 Metagraphs and a New RDGT

Since many supergraphs share the same graph for $\mathcal{B}$, they can be more efficiently represented by a combined structure that we call a *metagraph*. Moreover, metagraphs allow to define a new RDGT where several $\mathcal{A}$-edges jointly witness a counterexample to inclusion, so that counterexamples can be found earlier than with individual supergraphs.

A metagraph is a structure (X,g) where $X \subseteq E_{\mathcal{A}}$ is a set of $\mathcal{A}$-edges and $g \in G_{\mathcal{B}}$. The metagraph (X,g) represents the set of all supergraphs $\langle x,g\rangle$ with $x \in X$. The L-labels of supergraphs then become labels of the elements of X since the graph g is the same.

We lift basic concepts from supergraphs to metagraphs. For every character $\sigma \in \Sigma$, there is exactly one single-letter metagraph $(E^\sigma_{\mathcal{A}},E^\sigma_{\mathcal{B}})$. Let $M^1 = \{(E^\sigma_{\mathcal{A}},E^\sigma_{\mathcal{B}}) | \ \sigma \in \Sigma\}$. Thus, the set of single-letter metagraphs M^1 represents all single-letter supergraphs in S^1. The function *RightExtend* defines the composition of two metagraphs such that $RightExtend((X,g),(Y,h)) = (X;Y, g \mathbin{⨟}_{\mathsf{b}} h)$, which is the metagraph containing the supergraphs that are $⨟_{\mathsf{b}}$-right extensions of supergraphs contained in (X,g) by supergraphs contained in (Y,h). The L-labels of the elements $z \in X;Y$ are assigned after testing condition C. The function Min_{f} is defined on sets $X \subseteq E_{\mathcal{A}}$ s.t. $Min_{\mathsf{f}}(X)$ contains the $\sqsubseteq_{\mathsf{f}}$-minimal edges of X. If some edges are $\sqsubseteq_{\mathsf{f}}$-equivalent, then $Min_{\mathsf{f}}(X)$ contains just any of them. Let $Min_M(X,g) = (Min_{\mathsf{f}}(X), Min(g))$. Thus, $Min_M(X,g)$ contains exactly one representative of every $\simeq_{\mathsf{fb}}$ equivalence class of the $\sqsubseteq_{\mathsf{fb}}$-minimal supergraphs in (X,g).

It is not meaningful to define subsumption for metagraphs. Instead, we need to remove certain supergraphs (i.e., $\mathcal{A}$-edges) from some metagraph if another metagraph contains a $\sqsubseteq_{\mathsf{fb}}$-smaller supergraph. If no $\mathcal{A}$-edge remains, i.e., $X = \emptyset$ in (X,g), then this metagraph can be discarded. This is the purpose of introducing the function *Clean*: It takes two metagraphs (X,g) and (Y,h), and it returns a metagraph (Z,g) that describes all supergraphs from (X,g) for which there is no $\sqsubseteq_{\mathsf{fb}}$-smaller supergraph in (Y,h). Formally, if $h \sqsubseteq_{\mathsf{fb}} g$, then $x \in Z$ iff $x \in X$ and $\nexists y \in Y$ s.t. $x \sqsubseteq_{\mathsf{f}} y$. Otherwise, if $h \not\sqsubseteq_{\mathsf{fb}} g$, then $Z = X$. Now we define a generalized RDGT on metagraphs.

[4] A slightly modified version, presented in [2], holds for the version of the RDGT mentioned in the footnote on Definition 4.

Algorithm 1. *Inclusion Checking with Metagraphs*

```
Input: BA $\mathcal{A} = (\Sigma, Q_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \delta_\mathcal{A})$, $\mathcal{B} = (\Sigma, Q_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \delta_\mathcal{B})$, and the set $M^1_{\mathcal{A},\mathcal{B}}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.
$Next := \{Min_M((X,g)) \mid (X,g) \in M^1_{\mathcal{A},\mathcal{B}}\}$; $Init := \emptyset$;
while $Next \neq \emptyset$ do
    Pick and remove a metagraph $(X,g)$ from $Next$;
    $Clean_1((X,g), Init)$;
    if $X \neq \emptyset$ then
        $Clean_3(Init, (X,g))$; Add $(X,g)$ to $Init$;
$Processed := \emptyset$; $Next := Init$;
foreach $(X,g) \in Next$ do
    foreach $x \in X$ do
        if $\neg\mathsf{C}(\langle x,g\rangle)$ then label $x$ with $L$
while $Next \neq \emptyset \wedge \exists (X,g) \in Next \cup Processed.\ \exists x \in X.L(x)$ do
    Pick a metagraph $(X,g)$ from $Next$ and remove $(X,g)$ from $Next$;
    if $\neg RDGT^M_\mathsf{b}((X,g),(X,g))$ then return $FALSE$;
    if $\exists (Y,h) \in Processed : \neg RDGT^M_\mathsf{b}((Y,h),(X,g)) \vee \neg RDGT^M_\mathsf{b}((X,g),(Y,h))$ then
    return $FALSE$;
    Create $(X',g)$ from $(X,g)$ by removing the $L$-labels from $X$ and add $(X',g)$ to
    $Processed$;
    foreach $(Y,h) \in Init$ do
        $(Z,f) := Min_M(RightExtend((X,g),(Y,h)))$;
        if $Z \neq \emptyset$ then $Clean_1((Z,f), Next)$;
        if $Z \neq \emptyset$ then $Clean_2((Z,f), Processed)$;
        if $Z \neq \emptyset$ then
            $Clean_3(Next, (Z,f))$; $Clean_3(Processed, (Z,f))$; Add $(Z,f)$ to $Next$;
return $TRUE$;
```

Definition 8. *A pair of sets of $\mathcal{A}$-edges $X, Y \subseteq E_\mathcal{A}$ passes the* forward-downward jumping lasso-finding test*, denoted $LFT_\mathsf{f}(X,Y)$, iff there is an arc $\langle p, a_0, q_0\rangle$ in X (with the L-label) and an infinite sequence of arcs $\langle q'_0, a_1, q_1\rangle, \langle q'_1, a_2, q_2\rangle, \ldots$ in Y s.t. $p \in I_\mathcal{A}$, $q'_i \preceq_\mathsf{f} q_i$ for all $i \geq 0$, and $a_j = 1$ for infinitely many j's.*

Definition 9. *$RDGT^M_\mathsf{b}((X,g),(Y,h))$ iff, whenever $LFT_\mathsf{f}(X,Y)$, then $LFT_\mathsf{b}(g,h)$.*

The following lemma shows the soundness of the new RDGT.

Lemma 15. *Let $(X,g),(Y,h)$ be metagraphs where all contained supergraphs are in S^R. If $\neg RDGT^M_\mathsf{b}((X,g),(Y,h))$, then $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$.*

If there are $x \in X, y \in Y$ s.t. $\neg RDGT_\mathsf{b}(\langle x,g\rangle, \langle y,h\rangle)$, then $\neg RDGT^M_\mathsf{b}((X,g),(Y,h))$, by Definitions 4, 8, and 9. Thus the completeness of the new RDGT follows already from Lemmas 7 and 8. Checking $RDGT^M_\mathsf{b}((X,g),(Y,h))$ can be done very efficiently for large numbers of metagraphs, by using an abstraction technique that extracts test-relevant information from the metagraphs and stores it separately (cf. [2]).

7 The Main Algorithm

Algorithm 1 describes our inclusion testing algorithm. The function *Clean* is extended to sets of metagraphs in the standard way and implemented in procedures $Clean_1$ and $Clean_3$ in which the result overwrites the first argument (the two procedures differ in the role of the first argument, and $Clean_3$ in addition discards empty metagraphs). Lines 1-6 compute the metagraphs which contain the subsumption-minimal 1-letter supergraphs. Lines 7-10 initialize the set *Next* with these metagraphs and assign the correct labels by testing condition C. $L(x)$ denotes that the $\mathcal{A}$-arc x is labeled with L. Lines 11-21 describe the main loop. It runs until *Next* is empty or there are no more L-labels left. In the main loop, metagraphs are tested (lines 13-14) and then moved from *Next* to *Processed* without the L-label (line 15). Moreover, new metagraphs are created and some parts of them discarded by the *Clean* operation (lines 16-21). Extra bookkeeping is needed to handle the case where L-labels are regained by supergraphs in *Processed* in line 19 (see $Clean_2$ in [2]).

Theorem 1. *Algorithm 1 terminates. It returns* TRUE *iff* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

8 Experimental Results

We have implemented the proposed inclusion-checking algorithm in Java (the implementation is available at `http://www.languageinclusion.org/CONCUR2011`) and tested it on automata derived from (1) mutual exclusion protocols [15] and (2) the Tabakov-Vardi model [18]. We have compared the performance of the new algorithm with the one in [1] (which only uses supergraphs, not metagraphs, and subsumption and minimization based on forward simulation on $\mathcal{A}$ and on $\mathcal{B}$), and found it better on average, and, in particular, on difficult instances where the inclusion holds. Below, we present a condensed version of the results. Full details can be found in [2].

In the first experiment, we inject artificial errors into models of several mutual exclusion protocols from [15][5], translate the modified versions into BA, and compare the sequences of program states (w.r.t. occupation of the critical section) of the two versions. For each protocol, we test language inclusion $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ of two variants $\mathcal{A}$ and $\mathcal{B}$. We use a timeout of 24 hours and a memory limit of 4GB. We record the running time and indicate a timeout by ">24h". We compare the algorithm from [1] against its various improvements proposed above. The basic new setting (denoted as "default" in the results) uses forward simulation as in [1] together with metagraphs from Section 6 (and some further small optimizations described in [2]). Then, we gradually add the use of backward simulation proposed in Section 4 (denoted by `-b` in the results) and forward simulation between $\mathcal{A}$ and $\mathcal{B}$ from Section 5 (denoted by `-c`, finally yielding the algorithm of Section 7). We also consider repeated quotienting w.r.t. forward/backward-simulation-equivalence before starting the actual inclusion checking (denoted by `-qr`), while the default does quotienting w.r.t. forward simulation only. In order to better show

[5] The models in [15] are based on guarded commands. We derive variants from them by randomly weakening or strengthening the guard of some commands.

Table 1. Language inclusion checking on mutual exclusion protocols. Forward simulation holds between initial states. The option `-c` is extremely effective in such cases.

Protocol	$\mathcal{A}$		$\mathcal{B}$		Algorithm of [1]	New Algorithm			
	Trans.	States	Trans.	States		default	`-b`	`-b -qr`	`-b -qr -c`
Peterson	33	20	34	20	0.46s	0.39s	0.54	0.61s	0.03s
Phils	49	23	482	161	12h36m	11h3m	7h21m	7h23m	0.1s
Mcs	3222	1408	21503	7963	>24h	2m43s	2m32s	2m49s	1m24s
Bakery	2703	1510	2702	1509	>24h	>24h	>24h	>24h	12s
Fischer	1395	634	3850	1532	4h50m	2m38s	2m50s	27s	3.6s
FischerV2	147	56	147	56	13m15s	5m14s	1m26s	1m1s	0.1s

Table 2. Language inclusion checking on mutual exclusion protocols. Language inclusion holds, but forward simulation does not hold between initial states (we call this category "inclusion"). The new alg. is much better in FischerV3, due to metagraphs. Option `-b` is effective in FischerV4. BakeryV2 is a case where `-c` is useful even if simulation does not hold between initial states.

Protocol	$\mathcal{A}$		$\mathcal{B}$		Algorithm of [1]	New Algorithm			
	Trans.	States	Trans.	States		default	`-b`	`-b -qr`	`-b -qr -c`
FischerV3	1400	637	1401	638	3h6m	45s	10s	11s	7s
FischerV4	147	56	1506	526	>24h	>24h	1h31m	2h12m	2h12m
BakeryV2	2090	1149	2091	1150	>24h	>24h	>24h	>24h	18s

Table 3. Language inclusion checking on mutual exclusion protocols. Language inclusion does not hold. Note that the new algorithm uses a different search strategy (BFS) than the alg. in [1].

Protocol	$\mathcal{A}$		$\mathcal{B}$		Algorithm of [1]	New Algorithm			
	Trans.	States	Trans.	States		default	`-b`	`-b -qr`	`-b -qr -c`
BakeryV3	2090	1149	2697	1506	12m19s	5s	6s	16s	15s
FischerV5	3850	1532	1420	643	7h28m	1m6s	1m47s	39s	36s
PhilsV2	482	161	212	80	1.1s	0.7s	0.8s	1s	1s
PhilsV3	464	161	212	80	1s	0.7s	0.8s	1.2s	1.1s
PhilsV4	482	161	464	161	10.7s	3.8s	4.5s	4.8s	4.8s

the capability of the new techniques, the results are categorized into three classes, according to whether (1) simulation holds, (2) inclusion holds (but not simulation), and (3) inclusion does not hold. See, resp., Tables 1, 2, and 3. On average, the newly proposed approach using all the mentioned options produces the best result.

In the second experiment, we use the Tabakov-Vardi random model[6] with fixed alphabet size 2. There are two parameters, *transition density* (td; average number of tran-

[6] Note that automata generated by the Tabakov-Vardi model are very different from a control-flow graph of a program. They are almost unstructured, and thus on average the density of simulation is much lower. Hence, we believe it is not a fair evaluation benchmark for algorithms aimed at program verification. However, since it was used in the evaluation of most previous works on language inclusion testing, we also include it as one of the evaluation benchmarks.

Table 4. Results of the Tabakov-Vardi experiments on two selected configurations. In each case, we generated 100 random automata and set the timeout to one hour. The new algorithm found more cases with simulation between initial states because the option `-qr` (do fw/bw quotienting repeatedly) may change the forward simulation in each iteration. In the "Hard" case, most of the timeout instances probably belong to the category "inclusion" (Inc).

	Hard: td=2, ad=0.1, size=30		Easy, but nontrivial: td=3, ad=0.6, size=50	
	The Algorithm of [1]	New Algorithm	The Algorithm of [1]	New Algorithm
Sim	1%, 32m42s	2%, 0.025s	13%, 2m5s	21%, 0.14s
Inc	16%, 43m	20%, 30m42s	68%, 26m14s	64%, 6m12s
nInc	49%, 0.17s	49%, 0.2s	15%, 0.3s	15%, 0.3s
TO	34%	29%	4%	0%

sitions per state and alphabet symbol) and *acceptance density* (ad; percentage of accepting states). The results of a complete test for many parameter combinations and automata of size 15 can be found in [2]. Its results can be summarized as follows. In those cases where simulation holds between initial states, the time needed is negligible. Also the time needed to find counterexamples is very small. Only the "inclusion" cases are interesting. Based on the results presented in [2], we picked two configurations (Hard: td=2, ad=0.1, size=30) and (Easy, but nontrivial: td=3, ad=0.6, size=50) for an experiment with larger automata. Both configurations have a substantial percentage of the interesting "inclusion" cases. The results can be found in Table 4.

9 Conclusions

We have presented an efficient method for checking language inclusion for Büchi automata. It augments the basic Ramsey-based algorithm with several new techniques such as the use of weak subsumption relations based on combinations of forward and backward simulation, the use of simulation relations between automata in order to limit the search space, and methods for eliminating redundant tests in the search procedure. We have performed a wide set of experiments to evaluate our approach, showing its practical usefulness. An interesting direction for future work is to characterize the roles of the different optimizations in different application domains. Although their overall effect is to achieve a much better performance compared to existing methods, the contribution of each optimization will obviously vary from one application to another. Such a characterization would allow a portfolio approach in which one can predict which optimization would be the dominant factor on a given problem. In the future, we also plan to implement both the latest rank-based and Ramsey-based approaches in a uniform way and thoroughly investigate how they behave on different classes of automata.

References

1. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.-D., Mayr, R., Vojnar, T.: Simulation subsumption in ramsey-based büchi automata universality and inclusion testing. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 132–147. Springer, Heidelberg (2010)

2. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.D., Mayr, R., Vojnar, T.: Advanced Ramsey-based Büchi Automata Inclusion Testing. Technical report FIT-TR-2011-03, FIT BUT, Czech Republic (2011)
3. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
4. Dill, D., Hu, A., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 255–265. Springer, Heidelberg (1992)
5. Doyen, L., Raskin, J.-F.: Improved Algorithms for the Automata-Based Approach to Model-Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 451–465. Springer, Heidelberg (2007)
6. Etessami, K.: A Hierarchy of Polynomial-Time Computable Simulations for Automata. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 131–131. Springer, Heidelberg (2002)
7. Etessami, K., Wilke, T., Schuller, R.A.: Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. SIAM J. Comp. 34(5) (2005)
8. Fogarty, S.: Büchi Containment and Size-Change Termination. Master's Thesis (2008)
9. Fogarty, S., Vardi, M.Y.: Büchi Complementation and Size-Change Termination. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 16–30. Springer, Heidelberg (2009)
10. Fogarty, S., Vardi, M.Y.: Efficient Büchi Universality Checking. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 205–220. Springer, Heidelberg (2010)
11. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing Simulations on Finite and Infinite Graphs. In: Proc. FOCS 1995. IEEE CS, Los Alamitos (1995)
12. Jones, N.D., Lee, C.S., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. In: Proc. of POPL 2001. ACM SIGPLAN, New York (2001)
13. Kupferman, O., Vardi, M.Y.: Verification of fair transition systems. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 372–384. Springer, Heidelberg (1996)
14. Kupferman, O., Vardi, M.Y.: Weak Alternating Automata Are Not That Weak. ACM Transactions on Computational Logic 2(2), 408–429 (2001)
15. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
16. Sistla, A.P., Vardi, M.Y., Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194, pp. 465–474. Springer, Heidelberg (1985)
17. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS. vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
18. Tabakov, D., Vardi, M.Y.: Model Checking Büchi Specifications. In: Proc. of LATA 2007 (2007)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. of LICS 1986. IEEE Comp. Soc. Press, Los Alamitos (1986)
20. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)

Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations[⋆]

Salvatore La Torre and Margherita Napoli

Dipartimento di Informatica
Università degli Studi di Salerno, Italy

Abstract. A multi-stack pushdown system is a natural model of concurrent programs. The basic verification problems are in general undecidable (two stacks suffice to encode a Turing machine), and in the last years, there have been some successful approaches based on under-approximating the system behaviors. In this paper, we propose a restriction of the semantics of the general model such that a symbol that is pushed onto a stack can be popped only within a bounded number of context-switches. Note that, we allow runs to be formed of unboundedly many execution contexts, we just bound the scope (in terms of number of contexts) of matching push and pop transitions. We call the resulting model a *multi-stack pushdown system with scope-bounded matching relations* (SMPDS). We show that the configuration reachability and the location reachability problems for SMPDS are both PSPACE-complete, and that the set of the reachable configurations is regular, in the sense that there exists a multi-tape finite automaton that accepts it.

1 Introduction

Multi-stack pushdown systems are a natural and well-established model of programs with both concurrency and recursive procedure calls, which is suitable to capture accurately the flow of control. A multi-stack pushdown system is essentially a finite control equipped with one or more pushdown stores. Each store encodes a recursive thread of the program and the communication between the different threads is modelled with the shared states of the finite control.

The class of multi-stack pushdown systems is very expressive. It is well known that two stacks can simulate an unbounded read/write tape, and therefore, a push-down system with two stacks suffices to mimic the behaviour of an arbitrary Turing machine. In the standard encoding, it is crucial for the automaton to move an arbitrary number of symbols from one stack to another and repeat this for arbitrarily many times. To achieve decidability it is thus necessary to break this capability by placing some limitations on the model.

In the last years, the analysis of multi-stack pushdown systems within a bounded number of execution contexts (in each context only one stack is used)

[⋆] This work was partially funded by the MIUR grants FARB 2009-2010 Università degli Studi di Salerno, Italy.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 203–218, 2011.

has been proposed as an effective method for finding bugs in concurrent programs [14]. This approach is justified in practice by the general idea that most of the bugs of a concurrent program are likely to manifest themselves already within few execution contexts (which has also been argued empirically in [13]). Though bounding the number of context-switching in the explored runs does not bound the depth of the search of the state space (the length of each context is unbounded), it has the immediate effect of bounding the interaction among different threads and thus the exchanged information. In fact, the reachability problem with this limitation becomes decidable and is NP-complete [11,14].

In this paper, we propose a decidable notion of multistack pushdown system that does not bound the number of interactions among the different stacks, and thus looks more suitable for a faithful modeling of programs with an intensive interaction between threads. We impose a restriction which is technically an extension of what is done in the bounded context-switching approach but is indeed conceptually very different. We allow an execution to go through an unbounded number of contexts, however recursive calls that are returned can only span for a bounded number of contexts. In other words, we bound the *scope* of the matching push and pop operations in terms of the number of context switches allowed in the between. Whenever a symbol is pushed onto a stack, it is either popped within a bounded number of context switches or never popped. This has the effect that in an execution of the system, from each stack configuration which is reached, at most a finite amount of information can be moved into the other stacks, thus breaking the ability of the multistack pushdown system of simulating a Turing machine. We call the resulting model a *multistack pushdown system with scope-bounded matching relations* (SMPDS).

Bounding the scope of the matching relations to a given number of context-switches instead of the whole computation has some appealing aspects. First, for a same bound k, it covers a reachable space which is at least that covered by the bounded context-switching analysis, and in some cases can be significantly larger. In fact, there are systems such that the whole space of configurations can be explored with a constant bound on the scope while it will require a large number of context switches or even unboundedly many of them (see for example Figure 1). Thus, for systems where the procedure calls do not need to hang for many contexts before returning, the bounded scope analysis covers the behavior explored with the context bounded analysis with smaller values of the bound k, which is a critical parameter for the complexity of the decision algorithms (time is exponential in k in both settings). It is known that looking at the computations as nested words with multiple stack relations, when restricting to k-context computations, the corresponding set has a bounded tree-width (see [12]). Moreover, a bounded context-switch multistack pushdown system can be simulated by a standard pushdown system with just one stack (see [7,10]). For SMPDS instead, any of these results does not seem to work or at least requires a more complex encoding. Finally, SMPDS have a natural and meaningful semantics for infinite computations which allows to observe also infinitely many interactions between the different threads.

In this paper, we tackle the reachability problem for SMPDS, and show that it is PSPACE-complete. Our decision algorithm is an elaborated adaptation of the saturation procedure which is used for the analysis of pushdown sytems [15] and reused in [14] for solving the reachability within a bounded number of context switches. As in [14] we grow the set of reachable configurations as tuples of automata accepting configurations of each stack, and construct the automata by iterating the saturation algorithm from [15] into layers, each for execution context. However, in [14] the construction has a natural limit in the allowed number of contexts which is bounded, while in our setting, we appeal to the bound on the scope of the matching relations and use the automata to represent not all the stack contents but only the portions corresponding to the last k execution contexts.

We assume that the executions of an SMPDS go through rounds of schedule, where in each round all stacks are scheduled exactly once and always according to the same order. To prove our results, we construct a finite graph, that we call the *reachability graph*, whose nodes are n-tuples of k-layered automata along with the n-states of the finite control at which the context-switches have happened in the last round. The edges of the graph link a tuple u to any other tuple v such that executing a whole round, by context-switching at the states listed in u, the stack content of the last k rounds is captured by the tuple of k-layered automata in v. We show that the reachability graph reduces the location reachability for SMPDS to standard reachability in finite graphs. Since its size is exponential in the number of locations, the number of stacks and the number of rounds in a scope, this problem can be decided in polynomial space. The reachability graph can be also used as a general framework where the k-layered automata are combined to obtain an n-tape finite automaton that accept exactly the reachable configurations.

The configuration reachability problem is stated as the problem of deciding if a configuration within a set of target configurations is reachable. The target set is given as a Cartesian product of regular languages over the stack alphabets. The n-tape finite automaton accepting the reachable configurations can be modified to compute the intersection with the target set (this can be done componentwise since the target set is a Cartesian product) and thus the configuration reachability problem can be reduced to check emptiness of an n-tape finite automaton of exponential size, and thus again this can be done in polynomial space. PSPACE-hardness of the considered problems can be shown with standard constructions from the membership of linear bounded automata.

Related Work. In [9], the notion of context is relaxed and the behaviours of multistack pushdown systems are considered within a bounded number of phases, where in each phase only one stack is allowed to execute pop transitions but all the stacks can do push transitions. The reachability problem in this model turns out to be 2ETIME-complete [9,5]. We observe that in each phase an unbounded amount of information can pass from one stack to any other, but still this can be done only a bounded number of times. Thus, in some sense this extension is orthogonal to that proposed in this paper. Moreover, it is simple to verify that the extension of SMPDS where contexts are replaced with phases in the rounds is as powerful as Turing machines.

The notion of bounded-context switching has been successfully used in recent work: model-checking tools for concurrent Boolean programs [6,10,16]; translations of concurrent programs to sequential programs reducing bounded context-switching reachability to sequential reachability [7,10]; model-checking tools for Boolean abstractions of parameterized programs (concurrent programs with unboundedly many threads each running one of finitely many codes) [8]; sequentialization algorithms for parameterized programs [3]; model-checking of programs with dynamic creation of threads [1]; analysis of systems with systems heaps [2], systems communicating using queues [4], and weighted pushdown systems [11].

2 Multistack Pushdown Systems

In this section we introduce the notations and definitions we will use in the rest of the paper. Given two positive integers i and j, $i \leq j$, we denote with $[i, j]$ the set of integers k with $i \leq k \leq j$, and with $[j]$ the set $[1, j]$.

A multi-stack pushdown system consists of a finite control along with one or more pushdown stores. There are three kinds of transitions that can be executed: the system can push a symbol on any of its stacks, or pop a symbol from any of them, or just change its control location by maintaining unchanged the stack contents. For the ease of presentation and without loss of generality we assume that the the symbols used in each stack are disjoint from each other. Therefore, a multi-stack pushdown system is coupled with an n-stack alphabet $\widetilde{\Gamma}_n$ defined as the union of n pairwise disjoint finite alphabets $\Gamma_1, \dots, \Gamma_n$. Formally:

Definition 1. (Multi-stack pushdown system) *A* multi-stack pushdown system *(*Mpds*) with n stacks is a tuple $M = (Q, Q_I, \widetilde{\Gamma}_n, \delta)$ where Q is a finite set of states, $Q_I \subseteq Q$ is the set of initial states, $\widetilde{\Gamma}_n$ is an n-stack alphabet, and $\delta \subseteq (Q \times Q) \cup (Q \times Q \times \Gamma) \cup (Q \times \Gamma \times Q)$ is the transition relation.*

We fix an n-stack alphabet $\widetilde{\Gamma}_n = \cup_{i=1}^{n} \Gamma_i$ for the rest of the paper. A transition (q, q') is an internal transition where the control changes from q to q' and the stack contents stay unchanged. A transition (q, q', γ) for $\gamma \in \Gamma_i$ is a push-transition where the symbol γ is pushed onto stack i and the control changes from q to q'. Similarly, (q, γ, q') for $\gamma \in \Gamma_i$ is a pop-transition where γ is read from the top of stack i and popped, and the control changes from q to q'. A *stack content* w is a possibly empty finite sequence over Γ. A *configuration* of an Mpds M is a tuple $C = \langle q, w_1, \dots, w_n \rangle$, where $q \in Q$ and each w_i is a stack content. Moreover, C is *initial* if $q \in Q_I$ and $w_i = \varepsilon$ for every $i \in [n]$. Transitions between configurations are defined as follows: $\langle q, w_1, \dots, w_n \rangle \rightarrow_M \langle q', w'_1, \dots, w'_n \rangle$ if one of the following holds (M is omitted whenever it is clear from the context):

[Push] there is a transition $(q, q', \gamma) \in \delta$ such that $\gamma \in \Gamma_i$, $w'_i = \gamma \cdot w_i$, and $w'_h = w_h$ for every $h \in ([n] \setminus \{i\})$.

[Pop] there is a transition $(q, \gamma, q') \in \delta$ such that $w_i = \gamma \cdot w'_i$ and $w'_h = w_h$ for every $h \in ([n] \setminus \{i\})$.

[Internal] there is a transition $(q, q') \in \delta$, and $w'_h = w_h$ for every $h \in [n]$.

A *run* of M from C_0 to C_m, with $m \geq 0$, denoted $C_0 \rightsquigarrow_M C_m$, is a possibly empty sequence of transitions $C_{i-1} \rightarrow_M C_i$ for $i \in [m]$ where each C_i is a configuration. A pushdown system (Pds) is an Mpds with just one stack.

Reachability. A *target set of configurations* for M is $S \times \mathcal{L}(A_1) \times \ldots \times \mathcal{L}(A_n)$ such that $S \subseteq Q$ and for $i \in [n]$, $\mathcal{L}(A_i) \subseteq \varGamma_i^*$ is the language accepted by a finite automaton A_i. Given an Mpds $M = (Q, Q_I, \widetilde{\varGamma}_n, \delta)$ and a target set of configurations T, the *reachability problem* for M with respect to target T asks to determine whether there is a run of M from C_0 to C such that C_0 is an initial configuration of M and $C \in T$. The *location reachability* problem for Mpds is defined as the reachability problem with respect to a target set of the form $S \times \varGamma_1^* \times \ldots \times \varGamma_n^*$.

It is well known that the reachability problem for multi-stack pushdown systems is undecidable already when only two stacks are used (two stacks suffice to encode the behavior of a Turing machine) and is decidable in polynomial time (namely, cubic time) when only one stack is used (pushdown systems).

Theorem 1. *The (location) reachability problem is undecidable for* Mpds *and is decidable in cubic time for* Pds*.*

Execution Contexts and Rounds. We fix an Mpds $M = (Q, Q_I, \widetilde{\varGamma}_n, \delta)$. A *context* of M is a run of M where the pop and push transitions are all over the same stack (the only active stack in the context), i.e., a run $C_0 \rightarrow_M C_1 \rightarrow_M \ldots \rightarrow_M C_m$ over the transition rules $\delta \cap ((Q \times Q) \cup (Q \times \varGamma_i \times Q) \cup (Q \times Q \times \varGamma_i))$ for some $i \in [n]$. The *concatenation* of two runs $C \rightsquigarrow_M C'$ and $C' \rightsquigarrow_M C''$ is the run $C \rightsquigarrow_M C' \rightsquigarrow_M C''$. A *round* of M is the concatenation of n contexts $C_{i-1} \rightsquigarrow C_i$ for $i \in [n]$, where stack i is the active stack of $C_{i-1} \rightsquigarrow C_i$. Note that a run without push and pop transitions (and in particular, a run formed of a single configuration) is a context where the active stack can be any of the stacks. Thus, each run of M can be seen as the concatenation of many rounds (and contexts).

Multi-stack Pushdown Systems with Scope-bounded Matching Relations. In the standard semantics of Mpds a pop transition (q, γ, q') can be always executed from q when γ is at the top of the stack. We introduce a semantics that restricts this, in the sense that the pop transitions are allowed to execute only when the symbol at the top of the stack was pushed within the last k rounds, for a fixed k. Thus, each symbol pushed onto the stack can be effectively used only for boundedly many rounds (*scope of the push/pop matching*). We call the resulting model an Mpds *with scope-bounded matching relations* (SMpds, for short). We use the notation k-SMpds when we need to stress the actual bound k on the number of rounds of the scope. Formally:

Definition 2. (MPDS WITH SCOPE-BOUNDED MATCHING RELATIONS) *A multi-stack pushdown system (k-*SMPDS*) with n stacks and k-round scope is a tuple* $M = (k, Q, Q_I, \widetilde{\Gamma}_n, \delta)$ *where* $k \in \mathbb{N}$ *and* $(Q, Q_I, \widetilde{\Gamma}_n, \delta)$ *is as for* MPDS.

For an MPDS $M = (Q, Q_I, \widetilde{\Gamma}_n, \delta)$ we often denote the corresponding SMPDS $(k, Q, Q_I, \widetilde{\Gamma}_n, \delta)$ with (k, M).

To describe the behavior of such systems, we extend the notion of configuration. We assume that when a symbol is pushed onto the stack, it is annotated with the number of the current round. Also we keep track of the current round number and the currently active stack. An *extended configuration* is of the form $\langle r, s, q, w_1, \ldots, w_n \rangle$ $r \in \mathbb{N}$, $s \in [n]$, $q \in Q$ and $w_i \in (\Gamma \times \mathbb{N})^*$. Note that by removing the components r and s, and the round annotation from the stacks, from each extended configuration we obtain a standard configuration. We define a mapping *conf* that maps each extended configuration to the corresponding configuration, that is, $\mathit{conf}(\langle r, s, q, w_1, \ldots, w_n \rangle)$ is the configuration $\langle q, \pi(w_1), \ldots, \pi(w_n) \rangle$, where for each $w_i = (\gamma_1, i_1) \ldots (\gamma_m, i_m)$, with $\pi(w_i)$ we denote the stack content $\gamma_1 \ldots \gamma_m$. An extended initial configuration is an extended configuration $E = \langle r, s, q, w_1, \ldots, w_n \rangle$ such that $r = s = 1$ and $\mathit{conf}(E)$ is an initial configuration.

Transitions between extended configurations are formally defined as follows: $\langle r, s, q, w_1, \ldots, w_n \rangle \mapsto_M \langle r', s', q', w'_1, \ldots, w'_n \rangle$ if one of the following holds (M is omitted whenever it is clear from the context):

[Push] $r' = r$, $s' = s$, $w'_h = w_h$ for every $h \in ([n] \setminus \{s\})$, and there is a transition $(q, q', \gamma) \in \delta$ such that $\gamma \in \Gamma_s$ and $w'_s = (\gamma, r) \cdot w_s$.

[Pop] $r' = r$, $s' = s$, $w'_h = w_h$ for every $h \in ([n] \setminus \{s\})$, and there is a transition $(q, \gamma, q') \in \delta$ such that $w_s = (\gamma, h) \cdot w'_s$ and $h > r - k$.

[Internal] $r' = r$, $s' = s$, and there is a transition $(q, q') \in \delta$ $w'_h = w_h$ for every $h \in [n]$.

[Context-switch] $w'_h = w_h$ for every $h \in [n]$, and if $s = n$ then $r' = r + 1$ and $s' = 1$, otherwise $r' = r$ and $s' = s + 1$.

An *extended run* of a k-SMPDS M from E_0 to E_m, with $m \geq 0$, denoted $E_0 \hookrightarrow_M E_m$, is a possibly empty sequence of transitions $E_{i-1} \mapsto_M E_i$ for $i \in [m]$ where each E_i is an extended configuration. A *run* of a k-SMPDS M from C to C', denoted $C \rightsquigarrow_M C'$, is the projection through *conf* of an extended run $E \hookrightarrow_M E'$ such that $C = \mathit{conf}(E)$ and $C' = \mathit{conf}(E')$. Using this notion of run, we define the reachability and the location reachability problems for SMPDS as for MPDS. Given a run $\rho = C' \rightsquigarrow_M C$ where $C = \langle q, w_1, \ldots, w_n \rangle$, and an integer $m \geq 0$, with $\mathit{Last}_\rho(C, m)$ we denote the tuple $(y_1, \ldots, y_n)$ where for $i \in [n]$, y_i a prefix of w_i and contains the symbols pushed onto stack i in the last m rounds of ρ and not yet popped.

Example 1. Figure 1 gives a 2-stack MPDS M with stack alphabets $\Gamma_1 = \{a\}$ and $\Gamma_2 = \{b, c\}$. The starting location of M is q_0. A typical execution of the system M from the initial location q_0 starts pushing a on stack 1, and then b on stack 2. Thus, M iteratively pushes a on stack 1 and c on stack 2, until it starts popping

$M = (\{q_i | i \in [0,5]\}, \{q_0\}, \Gamma_1 \cup \Gamma_2, \delta) \quad \Gamma_2 = \{a\}, \ \Gamma_2 = \{b, c\}$

$\delta = \{ (q_0, q_1, a), (q_1, q_2, b), (q_2, q_3, a), (q_3, q_2, c), (q_2, c, q_4), (q_2, b, q_5), (q_4, c, q_4), (q_4, b, q_5) \}$

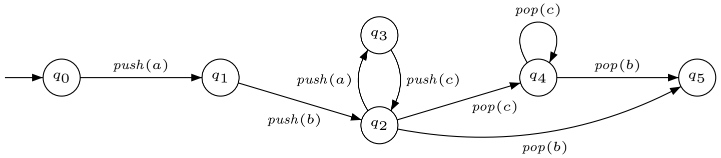


Fig. 1. The Mpds M from Example 1 and its graphical representation

c from stack 2. When all c's are popped out, it can also pop the b and finally reach location q_5 with stack 2 empty. Observe that each iteration of pushes of a's and c's (resp. b's) is a round, and assuming k of such iterations, a run ρ as described above can be split into at least k-rounds. Thus, it cannot be captured by any run of the SMpds (h, M) for any $h < k$. However, denoting with ρ' the prefix of ρ up to the first pop transition, clearly ρ' is a run of $(1, M)$. □

3 Reachability within a Fixed Number of Contexts and Phases

Bounded-context Switching Reachability. A *k-round run* of M is the concatenation of k rounds. The *reachability problem within k rounds* is the restriction of the reachability problem to the sole k-round runs of M. It is defined as the problem of determining whether there is a k-round run of M starting from an initial configuration and reaching a target configuration in T.

Theorem 2. *[11,14] The (location) reachability problem within k rounds for* Mpds *is NP-complete.*

Bounded-phase Reachability. A *phase* of M is a run of M where the pop transitions are all from the same stack (pushes onto any of the stacks are allowed within the same phase). Exploring all the runs of a system obtained as the concatenation of k phases ensures a better coverage of the state space compared to k-contexts reachability. In fact, a k-phase run can be formed of an arbitrary number of contexts (for example, a run that iterates k times a push onto stack 1 and a push onto stack 2 is a $2k$-context one, while it uses only one phase). On the other side, the resulting reachability problem has higher complexity.

Theorem 3. *[9,5] The location reachability problem within k phases for* Mpds *is* 2Etime*-complete.*

Coverage of the State Space in the Different Notions of Reachability. We compare the different reachability problems we have introduced in terms of coverage

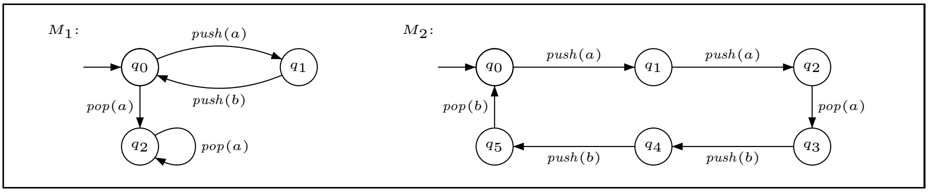


Fig. 2. Graphical representation of the MPDS M_1 and M_2

of the reachable state space of a multi-stack pushdown system. We start observing that for the MPDS M from Example 1, for each $k \geq 2$, the configuration $\langle q_4, a^k, c^{k-1}b\rangle$ is reachable in M within at least k rounds and is also reachable in the SMPDS $(1, M)$. Moreover, for each $k \geq 1$, the configuration $\langle q_5, a^k, \varepsilon\rangle$ is reachable in the SMPDS (k, M) and is not reachable in any of the SMPDS (h, M) for $h < k$. Therefore, the reachable set of a k-SMPDS strictly covers the set of configurations reachable within k-rounds, and thus provides a more careful approximation of the reachable configurations of MPDS. Formally, the following result holds.

Lemma 1. *Let $M = (Q, Q_I, \widetilde{\Gamma}_n, \delta)$ be an* MPDS.

If a configuration C' is reachable from C in (k, M), then C' is also reachable from C in M. Vice-versa, there is an MPDS *M' such that for each $k \in \mathbb{N}$ there is a reachable configuration C that is not reachable in the* SMPDS *(k, M).*

If a configuration C' is reachable from C in M within k rounds, then C' is also reachable from C in (k, M). Vice-versa, there is an MPDS *M' such that for each $k \in \mathbb{N}$ there is a reachable configuration C of (k, M') that is not reachable within k rounds.*

Now, fix $\Gamma_1 = \{a\}$, $\Gamma_2 = \{b\}$. Let M_1 be the 2-stack MPDS from Figure 2. Since the only pop transitions are from the same stack, any run of M_1 is 1-phase. It is simple to see that a configuration $C_k = \langle q_2, \varepsilon, b^k\rangle$ for $k \in \mathbb{N}$, is only reachable with a run of at least k rounds and moreover in the last round all the a's pushed onto stack 1 should be readable in order to pop all them out. Thus, C_k is not reachable in the SMPDS (h, M) for any $h < k$. Moreover, let M_2 be the 2-stack MPDS from Figure 2. It is simple to see that any run of M_2 is also a run of $(1, M_2)$. However, a configuration $C_k = \langle q_0, a^k, b^k\rangle$ for $k \in \mathbb{N}$ is not reachable with a run with less than $2k$ phases.

Therefore, from the above arguments and the given definitions, we get that the notion of reachability for SMPDS is not comparable with the notion of reachability up to k-phases. More precisely, we get the following result.

Lemma 2. *There is an* MPDS *M such that any reachable configuration can be reached within one phase, and for each $k \in \mathbb{N}$ there is a configuration C that is not reachable in the* SMPDS *(k, M).*

There is an MPDS *M such that any reachable configuration can be reached also in $(1, M)$, and for any $k \in \mathbb{N}$ there is a configuration C that is not reachable within k phases.*

4 Location Reachability for SMpds

In this section, we present a decision algorithm that solves the location reachability problem, and address its computational complexity. The main step in our algorithm consists of constructing a finite graph such that solving the considered decision problem equals to solving a standard reachability problem on this graph, that we thus call the reachability graph. The nodes of the reachability graph are tuples of n automata, one for each stack and each accepting the words formed by the symbols pushed onto the corresponding stack in the last k rounds of a run of the k-SMpds. Any such component automaton is structured into layers, which are added incrementally one on the top of the lower ones essentially by applying the saturation procedure from [15]. We use the reachability graph in Section 5 as a main component in the construction of a multi-tape finite automaton accepting the reachable configurations of a given SMpds.

For the rest of this section we fix a k-SMpds $M = (k, Q, Q_I, \widetilde{\Gamma}_n, \delta)$ and $\Gamma = \cup_{i=1}^{n} \Gamma_i$. We start defining k-layered automata. We assume that the reader is familiar with the basic concepts on finite automata and graphs.

k-layered Automata. A k-layered automaton A of M is essentially a finite automaton structured into $(k+1)$ layers whose set of states contains k copies of each $q \in Q$ (each copy belonging to a different layer from 1 through k) along with a new state q_F which is the sole final state and the sole state of layer 0. The input alphabet is Γ_h for some h, and the set of transitions contains only transitions of the form (s, γ, s') where the layer of s' is not larger than the layer of s. Moreover, there are no transitions leaving from q_F and every state is either isolated or connected to q_F. Formally, we have:

Definition 3. (k-layered automaton) *A k-layered finite automaton A of M over Γ_h is a finite automaton $(S, \Gamma_h, \delta, S_0)$, where $h \in [n]$, Γ_h is the input alphabet and:*

- *$S = \cup_{i=0}^{k} S_i$ is the set of states where $S_0 = \{q_F\}$ and $S_i = \{\langle q, i\rangle \mid q \in Q\}$, for $i \in [k]$ (for $i \in [0, k]$, S_i denotes the layer i of A);*
- *$\delta \subseteq S \times (\Gamma_h \cup \{\varepsilon\}) \times S$ is the transition relation and contains transitions of the form (s, γ, s') such that $s \in S_i$, $s' \in S_j$, $i > 0$ and $i \geq j$;*
- *for each state $s \in S$, either there is a run from s to q_F or s is isolated (i.e., there are no transitions involving s).*

For $t \in [0, k]$, S_t is called the top layer *if t is the largest i such that there is at least a state of layer i which is connected to q_F (t is referred to as the* top-layer index*) and A is* full *if its top layer is S_k. The language accepted by A from a top-layer state $\langle q, t\rangle$, for $t > 0$, is denoted $\mathcal{L}(A, q)$. Moreover, $\mathcal{L}(A, q_F)$ denotes the language $\{\epsilon\}$ accepted A from q_F when the top-layer index is* 0 .

Note that two k-layered automata over the same alphabet Γ_h may differ only on the set of transitions and the only k-layered automaton over the alphabet Γ_h of top-layer index 0 is the one having no transitions. In the following, we often refer to a state $\langle q, i\rangle$ of a layered automaton as the copy of q in layer i .

We define the following transformations on k-layered automata. Let A be a k-layered automaton A with alphabet Γ_h and top-layer index t.

With *DownShift*(A) we denote the k-layered automaton A' obtained from A by shifting the transitions one layer down, i.e., the set of transitions of A' is the smallest set such that if $(\langle q, j\rangle, \gamma, s)$ is a transition of A with $j > 1$, then $(\langle q, j-1\rangle, \gamma, s')$ is a transition of A' where s' is q_F, if $s \in S_0 \cup S_1$, and is $\langle q', i-1\rangle$, if $s = \langle q', i\rangle$ for some $i \in [2, k]$ (note that if S_t is the top layer of A, with $t > 1$, then S_{t-1} is the top layer of A').

With *Add*(A, s, s') we denote the k-layered automaton A' obtained from A by adding the transition (s, ε, s').

With *Saturate*(A) we denote the k-layered automaton A' obtained by applying to A the saturation procedure from [15] with respect to the internal transitions and the push and pop transitions involving stack h, and such that the new transitions that are added are all leaving from the top-layer states. Namely, let $t > 0$ be the top layer index (if $t = 0$, *Saturate* does nothing), the saturation procedure consists of repeating the following steps until no more transitions can be added (we let $\gamma \in \Gamma_h$ in the following):

- for an internal transition $(q, q') \in \delta$: $(\langle q', t\rangle, \varepsilon, \langle q, t\rangle)$ is added to set of transitions provided that $\langle q, t\rangle$ is connected to q_F;
- for a push-transition $(q, q', \gamma) \in \delta$: $(\langle q', t\rangle, \gamma, \langle q, t\rangle)$ is added to set of transitions provided that $\langle q, t\rangle$ is connected to q_F;
- for a pop-transition $(q, \gamma, q') \in \delta$: $(\langle q', t\rangle, \varepsilon, \langle q'', i\rangle)$, with $i \leq t$, is added to the set of transitions provided that there is a run of A' from $\langle q, t\rangle$ to $\langle q'', i\rangle$ over γ (note that such a run may contain an arbitrary number of ε-transitions; also, $\langle q'', i\rangle$ is not isolated, and thus, connected to q_F by definition).

Note that all the transitions which cross layers, that get added in the above saturation procedure, are ε-transitions. Moreover, we recall that a similar procedure is given in [15] for constructing a finite automaton accepting all the configurations of a PDS P which are reachable starting from those accepted by a given finite automaton R. The iterated application of the saturation procedure over the execution contexts of an MPDS is already exploited in [14]. However, only runs with a constant bounded number of context switches are considered there and thus it is sufficient to iterate k times. Here, we need to iterate more. The essence of our algorithm is captured by the following definition.

Let A be a k-layered automaton with top layer index t and q be a state of M, define *Successor*(A, q, q') as follows, where q' is a state of M, if $t > 0$, and $q' = q_F$, if $t = 0$. When A is full, *Successor*(A, q, q') denotes the automaton that is obtained from A by first shifting the transitions one layer down, then adding to the resulting automaton an ε-transition from $\langle q, k\rangle$ to $\langle q', k-1\rangle$ and then applying the saturation procedure. Formally: *Successor*(A, q, q') = *Saturate*(*Add*(*DownShift*$(A), \langle q, k\rangle, \langle q', k-1\rangle$))). Moreover, if $\langle q', k-1\rangle$ is connected to q_F then *Successor*(A, q, q') is a full k-layered automaton.

In the other case, i.e., if A is not full, *Successor*(A, q, q') denotes the automaton obtained as before except that no shifting is needed now. Formally, if $t > 0$ then *Successor*(A, q, q') = *Saturate*(*Add*$(A, \langle q, t+1\rangle, \langle q', t\rangle$)). Moreover, if $\langle q', t\rangle$ is connected to q_F then $\mathit{Successor}(A, q, q')$ is a k-layered automaton with top-layer index $(t+1)$. Finally, if $t = 0$, $\mathit{Successor}(A, q, q_F) = \mathit{Saturate}(\mathit{Add}(A, \langle q, 1\rangle, q_F))$ and is a k-layered automaton of top-layer index 1.

The Reachability Graph. In this section, we construct a finite graph that summarizes the computations of an SMPDS. In particular, each vertex stores, as tuples of k-layered automata, the portions of the stack contents that have been pushed onto the stacks in the last k rounds of any run leading to it. In each vertex is also stored the sequence of the states entered when the context switches occur in the last round. Each edge then summarizes the effects of the execution of an entire round on the information stored in the vertices.

The *reachability graph* $\mathcal{G}_M = (V_M, E_M)$ of M is defined as follows. The set of vertices V_M contains tuples of the form $(q_0, A_1, q_1, \ldots, A_n, q_n)$ where $q_0 \in Q$, each A_i is a k-layered automaton on alphabet Γ_i and for each $i \in [n]$, if the top-layer index of A_i is 0, then $q_i = q_F$, otherwise, $q_i \in Q$ is such that the copy of q_i in the top layer of A_i is not isolated (and thus connected to q_F). The set of edges E_M contains an edge from $(q_0, A_1, q_1, \ldots, A_n, q_n)$ to $(q'_n, A'_1, q'_1, \ldots, A'_n, q'_n)$ if $A'_i = \mathit{Successor}(A_i, q'_{i-1}, q_i)$, for $i \in [n]$ and denoting $q'_0 = q_0$ (note that in each vertex that has in-going edges, the first and the last components coincide).

Let A^0 be the k-layered automaton without any transition. We refer to each vertex of the form $(q, A^0, q_F, \ldots, A^0, q_F)$ such that $q \in Q_I$ as an *initial vertex* of $\mathcal{G}_M$ and denote it with in_q. Observe that since $q_F \notin Q$, $q \neq q_F$ always holds in the above tuple and thus each initial vertex has no in-going edges.

The following lemma relates the reachability within an SMPDS M with the reachability within $\mathcal{G}_M$.

Lemma 3. *Let M be a k-SMPDS. There is a path of m edges in $\mathcal{G}_M$ starting from an initial vertex in_q and ending at a vertex $v = (q_0, A_1, q_1, \ldots, A_n, q_n)$ if and only if there is a run ρ of m rounds in M starting from the initial configuration $\langle q, \varepsilon, \ldots, \varepsilon\rangle$ and ending at a configuration $C = \langle q_0, w_1, \ldots, w_n\rangle$ such that for $i \in [n]$, $y_i \in \mathcal{L}(A_i, q_i)$ where $(y_1, \ldots, y_n) = \mathrm{Last}_\rho(C, \min\{k, m\})$.*

Proof. We only sketch the proof for the "only if" part, the "if" part being similar. The proof is by induction on the length m of the path in $\mathcal{G}_M$. The base of the induction ($m = 0$) follows from the fact that there is a one-to-one correspondence between an initial vertex in_q of $\mathcal{G}$ and an initial configuration $\langle q, \varepsilon, \ldots, \varepsilon\rangle$ of M. Now, assume by induction hypothesis that the statement holds for $m \geq 0$. Thus, for any path of m edges from in_q to $v = (q_0, A_1, q_1, \ldots, A_n, q_n)$, there is a corresponding m-rounds run ρ of M from $\langle q, \varepsilon, \ldots, \varepsilon\rangle$ to $C = \langle q_0, w_1, \ldots, w_n\rangle$ such that for $i \in [n]$, $y_i \in \mathcal{L}(A_i, q_i)$ where $(y_1, \ldots, y_n) = \mathit{Last}_\rho(C, \min\{k, m\})$. Let $w_i = z'_i z''_i$ for $i \in [n]$ where $(z'_1, \ldots, z'_n) = \mathit{Last}_\rho(C, \min\{k-1, m\})$ (clearly, $z''_i = \varepsilon$ if $m < k$). Consider now an edge of $\mathcal{G}_M$ from v to $v' = (q'_0, A'_1, q'_1, \ldots, A'_n, q'_n)$. From the definition of $\mathcal{G}_M$ we get that $q'_0 = q'_n$, $A'_1 = \mathit{Successor}(A_1, q_0, q_1)$, and $A'_i = \mathit{Successor}(A_i, q'_{i-1}, q_i)$ for $i \in [2, n]$. Now, by a standard proof by induction on the number of applications of the rules of the saturation procedure, it is possible to show that, denoting $C_0 = C$ and for $i \in [n]$, there are $y'_i \in \mathcal{L}(A'_i, q'_i)$ and contexts $C_{i-1} \leadsto_M C_i$ where the stack i is active, and such that $y'_i z''_i$ is the content of stack i in C_i and q'_i is the location of C_i. Thus, $C \leadsto_M C_1 \ldots \leadsto_M C_n$ is a round and $C_n = \langle q'_n, y'_1 z''_1, \ldots, y'_n z''_n \rangle$ (recall that in each context $C_{i-1} \leadsto_M C_i$ only stack i can be updated). Therefore, appending this round as a continuation of run ρ, we get an $(m+1)$-rounds run with the properties stated in the lemma. □

Let $Q^T \subseteq Q$ be a target set of locations of M. By the above lemma, we can solve the reachability problem for SMpds by checking if any vertex $(q, A_1, q_1, \ldots, A_n, q_n)$, with $q \in Q^T$, can be reached in $\mathcal{G}_M$ from an initial vertex. Observe that the number of different k-layered automata for an alphabet Γ_i is at most $\chi_i = 2^{k|Q|+k|\Gamma_h||Q|^2+k^2|Q|^2}$. Since $\mathcal{G}$ has exactly $|Q_I|$ initial vertices, and all the other reachable vertices have the first and the last component that coincide, the number of vertices of $\mathcal{G}_M$ is at most $|Q_I| + \chi^n |Q|^n$ where $\chi = \max\{\chi_i \mid i \in [n]\}$, and thus exponential in the number of stacks, in the number of rounds and in the number of locations. Since we can explore the graph $\mathcal{G}_M$ on-the-fly, the proposed algorithm can be implemented in space polynomial in all the above parameters. With fairly standard constructions, it is possible to reduce the membership problem for a Turing machine to the location reachability problem for both a 2-stack k-SMpds and an n-stack 1-SMpds, thus showing Pspace-hardness in both n and k. Therefore, by Lemma 3 we get:

Theorem 4. *The location reachability problem for* SMpds *is* Pspace-*complete, and hardness can be shown both with respect to the number of stacks and the number of rounds.*

5 Configuration Reachability for SMpds

Regularity of the Reachable Configurations. Fix an SMpds $M = (k, Q, Q_I, \widetilde{\Gamma}_n, \delta)$. For $q \in Q$, with Reach_q we denote the set of tuples $(w_1, \ldots, w_n)$ such that $(q, w_1, \ldots, w_n)$ is configuration which is reachable from an initial configuration.

Observe that in general this set may not be expressible as a finite union of the Cartesian product of regular sets. For example, consider the 2-stack 1-SMpds M_1 with $\Gamma_1 = \{a\}$, $\Gamma_2 = \{b\}$, $Q_I = \{q_0\}$ and set of transitions $\delta = \{(q_0, q_1, a), (q_1, q_0, b)\}$. The only initial configuration is $\langle q_o, \varepsilon, \varepsilon \rangle$. It is simple to verify that $\mathit{Reach}_M(q_0) = \{(a^n, b^n) \mid n \geq 0\}$ that clearly cannot be expressed as finite union of the Cartesian product of regular languages.

In this section, we show that the sets Reach_q are recognized by an n-tape finite automaton. An *n-tape finite automaton* (n-FA) A is $(S, I, \Sigma, \Delta, F)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, Σ is a finite set of input symbols, $F \subseteq S$ is the set of final states, and $\Delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times [n] \times S$ is the set of transitions. The meaning of a transition rule (s, τ, i, s') is that for a state s the control of A can move to a state s' on reading the symbol τ from tape i. Every time a symbol τ is read from a tape the corresponding head moves to next symbol (clearly, if $\tau = \varepsilon$, the head does not move). A tuple $(w_1, \ldots, w_n)$

is accepted by A if there is a run from $s \in I$ to $s' \in F$ that consumes the words $w_1, \ldots, w_n$ (one on each tape).

Now, we construct an n-FA R_q recognizing $Reach_q$ for a location q of M. A main part of this automaton is the reachability graph $\mathcal{G}_M$ definend in Section 4. In particular, the automaton R_q starts from a vertex $(q, A_1, q_1, \ldots, A_n, q_n) \in V_M$ and then moves backwards along a path up to an initial vertex of G_M. We recall that from Lemma 3, each edge on a path of $\mathcal{G}_M$ corresponds to a round of a run of M and thus moving backwards along a path corresponds to visiting a run of $\mathcal{G}_M$ round-by-round starting from the last round (*round-switch mode*). Therefore, on each visited vertex v, R_q can read the symbols (if any) that, along the explored run, are pushed onto a stack in the current round and then never popped out, and this can be done by simulating just the top layers of the k-layered automata of v that are required to move into this round (*simulation mode*).

To implement these two main modes of the automaton R_q, we store in each state a vertex v of $\mathcal{G}_M$ (the currently explored vertex) and a tuple of the states which are currently visited for each k-layered automaton A of v. We also couple each such state with a count-down counter that maintains the number of rounds that the corresponding A will stay idle before executing the top layer transitions (when this counter is 0, A gets executed). Such counters are used to synchronize the two modes of R_q. When all the counters are non-zero, then R_q moves backwards in $\mathcal{G}_M$, and decrements all the counters, and this is repeated until some counters are 0. As long as there is a counter d set to 0, R_q can execute for the corresponding k-layered automaton either a top-layer transition or an ε-transition to a state q in a lower layer. In both cases, the state coupled with d gets updated accordingly to the taken transition, and only in the second case d is set to the difference between the top layer and the layer of q. The automaton R_q accepts when it enters a state where all the counters are 0, the coupled states are all q_F (the final state of each k-layered automaton), and the current vertex of G is an initial vertex.

Formally, the n-FA $R_q = (S_q, I_q, \widetilde{\Gamma}_n, \Delta_q, F_q)$ is defined as follows.

- The set of states S_q of R_q contains tuples of the form $(v, q_1, d_1, \ldots, q_n, d_n)$ where $v = (p_n, A_1, p_1, \ldots, A_n, p_n) \in V_M$ and for $i \in [n]$, $q_i \in Q \cup \{q_F\}$ and $d_i \in [0, k]$.
- The set of initial states I_q contains states of the form $(v, q_1, 0, \ldots, q_n, 0)$ where $v = (q_n, A_1, q_1, \ldots, A_n, q_n)$ and $q_n = q$.
- The set of final states F_q contains states of the form $(v, q_F, 0, \ldots, q_F, 0)$ where v is an intial vertex of $\mathcal{G}_M$.
- Denoting $s = (v, q_1, d_1, \ldots, q_n, d_n)$ and $s' = (v', q'_1, d'_1, \ldots, q'_n, d'_n)$, where $v = (p_n, A_1, p_1, \ldots, A_n, p_n)$ and $v' = (p'_n, A'_1, p'_1, \ldots, A'_n, p'_n)$, the set of transition Δ_q contains tuples (s, τ, h, s') such that $s \notin F_q$ and one of the following cases applies (in the following description, if a component of s' is not mentioned then it is equal to the same component of s):

  [simulate within the top layer] $d_h = 0$ and $(\langle q_h, t\rangle, \tau, \langle q'_h, t\rangle)$ is a transition of A_h where t is the top-layer index of A_h;

  [simulate exit from the top layer] $d_h = 0$, $d'_h = t - t'$, $(\langle q_h, t\rangle, \tau, \langle q'_h, t'\rangle)$ is a transition of A_h, where t is the top-layer index of A_h and $t' < t$;

[round-switch] for $i \in [n]$, $d_i > 0$ and $d'_i = d_i - 1$, there is an edge from v' to v in $\mathcal{G}_M$.

Observe that Δ_q is such that in any run of R_q whenever a component q_i of a visited state $(v, q_1, d_1, \ldots, q_n, d_n)$ becomes q_F, then it stays q_F up to the end of the run.

With a standard proof by induction, by Lemma 3, we get that R_q accepts the language $Reach_q$. Therefore, we have the following theorem.

Theorem 5. *For each $q \in Q$, the set* Reach_q *is accepted by an n-*FA*.*

Reachability of SMPDS. Fix an SMPDS $M = (k, Q, Q_I, \widetilde{\Gamma}_n, \delta)$ and a set of configurations $T = P \times \mathcal{L}(B_1) \times \ldots \times \mathcal{L}(B_n)$, where $P \subseteq Q$.

By Theorem 5, the reachability problem for SMPDS can be reduced to checking the emptiness of $\cup_{q \in P}(Reach_q \cap L)$ where $L = \mathcal{L}(B_1) \times \ldots \times \mathcal{L}(B_n)$. Denoting with $A_i \times B_i$ the standard cross product construction synchronized on the input symbols (ε transitions can be taken asynchronously), the construction of R_q given above can be modified such that in the simulation mode it tracks the behaviors of $A_i \times B_i$ instead of just the k-layered automaton A_i. Denote with R_q^T the resulting n-FA. We observe that in R_q^T, the simulation of each B_i starts from the initial states, and then the B_i-component gets updated only in the simulation mode in pair with the couped k-layered automaton. For the lack of space we omit the explicit construction of R_q^T.

The number of states of each R_q^T is at most $|V_M|\,(|Q|+1)^n(k+1)^n\,\chi^n$, where χ is the maximum over the number of states of $B_1, \ldots, B_n$. Recall that the number of vertices $|V_M|$ of $\mathcal{G}_M$ is exponential in n, $|Q|$ and k. Thus, the number of states of R_q^T is also exponential in the same parameters. Again, we can explore on-the-fly the state space of each R_q^T, thus we can check the emptiness of R_q^T in polynomial space, and in time exponential in n, $|Q|$ and k. Since each instance of the location reachability is also an instance of the general reachability problem, by Theorems 4 and 5 we get:

Theorem 6. *The reachability problem for* SMPDS *is* PSPACE*-complete, and hardness can be shown both with respect to the number of stacks and the number of rounds.*

6 Conclusion and Future Work

We have introduced a decidable restriction of multistack pushdown systems that allows unboundedly many context switches. Bounding the scope of the matching relations of push and pop operations on each stack has the effect of bounding the amount of information that can flow out of a stack configuration into the other stacks in the rest of the computation. For the resulting model we have shown that the set of reachable configurations is recognized by a multitape finite automaton, and that location and configuration reachability can be solved essentially by searching a graph of size exponential in the number of control states, stacks and rounds in a scope.

We see mainly two future directions in this research. We think that it would be interesting to experiment the effectiveness of the verification methodology based on our approach, by implementing our algorithms in a verification tool and compare them with competing tools. If on one side the considered reachability problem has a theoretically higher complexity compared to the case of bounded context-switching, on the other side smaller values of the bound on the number of context switches are likely to suffice for several systems. Moreover, our model presents the right features for studying the model-checking of concurrent software with respect to properties that concern non-terminating computations or require to explore unboundedly many contexts.

References

1. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
2. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
3. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs, http://www.dia.unisa.it/dottorandi/parlato/papers/sequ-parameterized.pdf
4. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
5. La Torre, S., Madhusudan, P., Parlato, G.: An infinite automaton characterization of double exponential time. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 33–48. Springer, Heidelberg (2008)
6. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222 (2009)
7. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
8. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
9. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170 (2007)
10. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
11. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
12. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: POPL, pp. 283–294 (2011)

13. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
14. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. TACAS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
15. Schwoon, S.: Model-Checking Pushdown Systems. Ph.D. thesis, Technische Universität München (2002)
16. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded java programs. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)

Granularity and Concurrent Separation Logic

Jonathan Hayman

Computer Laboratory, University of Cambridge

Abstract. When defining the semantics of shared-memory concurrent programming languages, one conventionally has to make assumptions about the atomicity of actions such as assignments. Running on physical hardware, these assumptions can fail to hold in practice, which puts in question reasoning about their concurrent execution. We address an observation, due to John Reynolds, that processes proved sound in concurrent separation logic are separated to an extent that these assumptions can be disregarded, so judgements remain sound even if the assumptions on atomicity fail to hold. We make use of a Petri-net based semantics for concurrent separation logic with explicit representations of the key notions of ownership and interference. A new characterization of the separation of processes is given and is shown to be stronger than existing race-freedom results for the logic. Exploiting this, sufficient criteria are then established for an operation of refinement of processes capable of changing the atomicity of assignments.

1 Introduction

When giving the semantics of concurrent shared-memory programming languages, one has to choose a level of *granularity* for primitive actions. For example, an assignment $x := [y] + [y]$ which assigns to x twice the value held in the location pointed-to by y might be considered to be a primitive action that occurs in one step of execution. On the other hand, depending on the compiler and the system architecture, the assignment might be split in two, in effect running $x := [y]; x := x + [y]$. In a sequential setting, this is of little significance, but in a concurrent setting the second interpretation can give rise to additional behaviour. Suppose, for example, that the variable z points to the same heap location, ℓ say, as y; we say that y and z are *aliases*. If we run the programs above in parallel with the program $[z] := [z] + 1$ which increments the value at ℓ, it can be seen that $(x := [y] + [y]) \parallel ([z] := [z] + 1)$ always yields an even value in x, but $(x := [y]; x := x + [y]) \parallel ([z] := [z] + 1)$ terminates with an odd value in x if the assignment $[z] := [z] + 1$ occurs after $x := [y]$ but before $x := x + [y]$.

The key feature of the example above is that there is a *data-race*, *i.e.* an attempt to concurrently access the memory location ℓ. One of the key properties enforced by concurrent separation logic [8] is that every parallel process owns a disjoint region of memory and each process only accesses locations that it owns, from which it follows that proved processes are race-free. In [10], John Reynolds argues that race-freedom as enforced by the logic means that issues of granularity

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 219–234, 2011.

can be disregarded; this has come to be known as *Reynolds' conjecture*. Due to the possibility of aliasing and the vital richness of the logic in allowing the transfer of ownership between processes, soundness of the logic and race-freedom are certainly non-trivial to prove, and led to the pioneering proof by Brookes [2] based on action traces. However, the intricacy of the powerful trace-based model stymied attempts to provide a formal proof of Reynolds' conjecture.

With the general goal of connecting concurrent separation logic and independence models for concurrency, in [7,6] it was shown how to define a Petri-net based semantics for programming languages and, based on this, a semantics for the logic was developed. A key feature of independence models such as Petri nets is that they support notions of *refinement*: semantic operations to allow the provision of more accurate specifications of actions without affecting overall system behaviour [5]. General properties that are intrinsically linked to the independence of events connect the refined process back to the original process. In [7,6], a form of refinement was given that could be applied to change the granularity assumed in the net semantics, and examples of its use were given. However, this did not fully address Reynolds' conjecture: there was no formal proof that the constraints governing when the refinement operation could be applied were always met for proved processes. This is tackled in this paper by establishing conditions on subprocesses and their refinements based on their *footprints*. In doing so, we reveal a new, stronger characterization of race-freedom arising from separation logic, which has the interesting side-effect of eliminating certain key examples of incompleteness of the logic.

Recently, Ferreira *et al.* have developed a new 'parameterized' operational semantics for programs and used it to show that a wide range of relaxations of assumptions of the memory model, including granularity, do not introduce extra behaviour for race-free processes [4]. However, not only does the net semantics presented here provide a valuable alternative perspective, with its direct account of ownership inherited from [7,6], but it also shows how the Petri-net model directly supports reasoning about issues such as granularity: there is no need to extend the model, refinement being a native operation on nets, and so there is no need to provide a new proof of soundness of the logic as there is in [4]. This is part of a general programme of research aimed at demonstrating the use of independence models for concurrency, such as Petri nets, in the semantics of programming languages and showing how they natively support the study of a wide range of important aspects such as race-freedom, separation, granularity, weak memory models and memory-optimisation.

2 Syntax

In this section, we present the syntax of the programming language to be considered, and in the following section we present its Petri-net semantics. These sections are as in [6,7] with the exception of equivalence at the end of Section 2 and Lemmas 1 and 2 in Section 3.

The programs that we consider operate on a *heap*, a finite partial map from a subset of heap locations Loc to values Val. We reserve the symbols $\ell, m, n, \ell', \ldots$ to range over heap locations and $h, h', \ldots$ to range over heaps. In a heap, a location can either be allocated, in which case the partial function is defined at that location, or be unallocated, in which case the partial function is undefined.

$$\mathrm{Heap} = \mathrm{Loc} \rightharpoonup_{\mathrm{fin}} \mathrm{Val}$$

Heap locations can point to other heap locations, so we have Loc ⊆ Val.

We now introduce a simple language for concurrent heap-manipulating programs. Its terms, ranged over by t, follow the grammar

$$\begin{array}{rll} t & ::= & \alpha \mid \texttt{alloc}(\ell) \mid \texttt{dealloc}(\ell) \mid t_1;t_2 \mid t_1 \parallel t_2 \mid \alpha_1.t_1 + \alpha_2.t_2 \\ & \mid & \texttt{while}\ b\ \texttt{do}\ t\ \texttt{od} \mid \texttt{with}\ r\ \texttt{do}\ t\ \texttt{od} \end{array}$$

where α ranges over *heap actions*, b ranges over *Boolean guards* and r ranges over *resources.*

Heap actions represent arbitrary forms of action on the heap that neither allocate nor deallocate locations. For every action α, we assume that we are given a set

$$\mathcal{A}\,[\![\alpha]\!] \subseteq \mathrm{Heap} \times \mathrm{Heap}$$

such that $(h_1, h_2) \in \mathcal{A}\,[\![\alpha]\!]$ implies $\mathrm{dom}(h_1) = \mathrm{dom}(h_2)$. The set $\mathcal{A}\,[\![\alpha]\!]$ represents all the ways in which the action α can behave; the interpretation is that α can occur in a heap h if there exists $(h_1, h_2) \in \mathcal{A}\,[\![\alpha]\!]$ s.t. (the graph of) h_1 is contained in h, yielding a new heap in which the values held at locations in $\mathrm{dom}(h_1)$ are updated according to h_2. For example, the semantics of an assignment $\ell \mathrel{\texttt{:=}} v'$ is

$$\mathcal{A}\,[\![\ell \mathrel{\texttt{:=}} v']\!] = \{(\{(\ell, v)\}, \{(\ell, v')\}) \mid v \in Val\}.$$

For any initial value v of ℓ, the heap can be updated to v' at ℓ. Further examples of action, such as pointer manipulation, are presented in [7,6].

Boolean guards are heap actions that can occur only if the property that they represent holds. A full logic is given in [7,6], but examples include:

- $\texttt{false}$, the action that can never occur (so $\mathcal{A}\,[\![\texttt{false}]\!] = \emptyset$),
- $\texttt{true}$, the action that can always occur (so $\mathcal{A}\,[\![\texttt{true}]\!] = \{(\emptyset, \emptyset)\}$),
- $\ell \mathrel{\texttt{?=}} v$, the action that proceeds only if ℓ holds value v, and
- $[\ell] \mathrel{\texttt{!=}} v$, the action that proceeds only if ℓ points to some location m that does not hold value v.

Allocation of heap locations can only occur through the term $\texttt{alloc}(\ell)$, which allocates an unused location and makes the existing location ℓ point to it, and deallocation can only occur through the term $\texttt{dealloc}(\ell)$, which deallocates the location pointed at by ℓ. There are primitives for iteration, sequential composition and parallel composition, and there is a form of guarded sum $\alpha_1.t_1 + \alpha_2.t_2$ which is a process that can run the process t_1 if the action α_1 can occur and can run the process t_2 if the action α_2 can occur. We sometimes use the notation $\texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{endif}$ for the term $b.t_1 + \neg b.\texttt{true}$.

Finally, critical regions can be specified through the construct `with` r `do` t `od` indexed by *resources* r drawn from a set Res. This construct enforces the property that no two processes can concurrently be inside a critical region protected by the same resource r. This is implemented by recording a set of available resources; a critical region protected by r may only be entered if r is available, and whilst the process is in the critical region, the resource is marked as unavailable.

We define equivalence on terms to be the least congruence such that:

$$(t_1;t_2);t_3 \equiv t_1;(t_2;t_3) \qquad (t_1 \parallel t_2) \parallel t_3 \equiv (t_1 \parallel t_2) \parallel t_3 \qquad t_1 \parallel t_2 \equiv t_2 \parallel t_1.$$

A key tool in what follows shall be *term contexts*, terms with a single 'hole' denoted $-$. We use the symbols $k, k', \ldots$ to range over term contexts.

$$\begin{array}{rcl} k & ::= & - \mid k;t \mid t;k \mid k \parallel t \mid t \parallel k \mid \alpha_1.k + \alpha_2.t \mid \alpha_1.t + \alpha_2.k \\ & & \mid \texttt{while}\, b \,\texttt{do}\, k \,\texttt{od} \mid \texttt{with}\, r \,\texttt{do}\, k \,\texttt{od} \end{array}$$

We denote by $k[t]$ the term obtained by substituting the term t for the hole in k. Term contexts and equivalence will be used together to discuss the subprocesses of terms; for example, $\alpha;\beta$ is a subprocess of $\alpha;(\beta;\gamma)$ since there is a context, namely $-;\gamma$, such that $\alpha;(\beta;\gamma) \equiv (-;\gamma)[\alpha;\beta]$. Note the essential rôle here of equivalence as opposed to syntactic equality.

3 Petri-Net Semantics

Petri nets represent the behaviour of processes as collections of events that affect regions of local state called conditions. The particular form of net that we give semantics over is nets without multiplicity in which contact inhibits the occurrence of events — cf. the 'basic' nets of [12].

We place some additional structure on nets to form what we call *embedded nets*, which shall be used to define the semantics of programs. An embedded net is a tuple $(\mathbf{C}, \mathbf{S}, E, \mathit{pre}, \mathit{post}, I, T)$, where $\mathbf{C}$ is the set of *control* conditions, $\mathbf{S}$ is the set of *state* conditions, disjoint from $\mathbf{C}$, and $I, T \subseteq \mathbf{C}$ are the *initial* and *terminal* control conditions, respectively. We require that $(\mathbf{C} \cup \mathbf{S}, E, \mathit{pre}, \mathit{post})$ forms a Petri-net with pre- and post-condition maps *pre* and *post*. As such, embedded nets are Petri nets equipped with a partition of their conditions into control and state conditions, alongside subsets of control conditions to indicate the initial and terminal control states of the process.

Any marking M of an embedded net can correspondingly be partitioned into (c, s) where $c = M \cap \mathbf{C}$ and $s = M \cap \mathbf{S}$. We write ${}^{\mathbf{C}}e$ for the control conditions that are preconditions to an event e, namely $\mathit{pre}(e) \cap C$, and define notation for the post-control conditions $e^{\mathbf{C}}$ and pre- and post-state conditions ${}^{\mathbf{S}}e$ and $e^{\mathbf{S}}$ similarly. We say that two embedded nets are *isomorphic* if there exists a bijection between their conditions and events that preserves the pre- and postconditions of events and preserves initial and terminal control markings. An isomorphism is said to be *state-preserving* if the restriction of the bijection to state conditions is the identity.

Table 1. Event notations

	Event e	${}^{\mathrm{S}}e$	e^{S}
Heap action	$\mathsf{act}_{(c,c')}(h,h')$	h	h'
Allocation	$\mathsf{alloc}_{(c,c')}(\ell,v,\ell',v')$	$\{(\ell,v)\}$	$\{(\ell,\ell'),(\ell',v'),\mathsf{curr}(\ell')\}$
Deallocation	$\mathsf{dealloc}_{(c,c')}(\ell,\ell',v')$	$\{(\ell,\ell'),(\ell',v'),\mathsf{curr}(\ell')\}$	$\{(\ell,\ell')\}$
Enter CR	$\mathsf{acq}_{(c,c')}(r)$	$\{r\}$	$\emptyset$
Leave CR	$\mathsf{rel}_{(c,c')}(r)$	$\emptyset$	$\{r\}$

For all the above notations, ${}^{\mathbf{C}}e = c$ *and* $e^{\mathbf{C}} = c'$*.*

The embedded net representing a term t is denoted $\mathcal{N}\,[\![t]\!]$ with initial control conditions $\mathrm{Ic}(t)$ and terminal control conditions $\mathrm{Tc}(t)$. The inductive definition is presented in [7,6]. The sets of control and state conditions are defined as follows:

Definition 1. *The control conditions* $\mathbf{C}$ *and state conditions* $\mathbf{S}$ *are defined as:*

- $\mathbf{C}$ *is ranged-over by* a *and follows the grammar*

$$a, a' ::= \mathsf{i} \mid \mathsf{t} \mid 1{:}a \mid 2{:}a \mid (a, a')$$

- $\mathbf{S} = \mathrm{Res} \cup (\mathrm{Loc} \times \mathrm{Val}) \cup \{\mathsf{curr}(\ell) \mid \ell \in \mathrm{Loc}\}$

A marking of state conditions $s \subseteq \mathbf{S}$ has $r \in s$ if the resource r is available; the heap holds value v at ℓ if $(l, v) \in s$; and the location ℓ has been allocated if $\mathsf{curr}(\ell) \in s$. Only certain markings of state conditions are sensible, namely those that are finite, satisfying $\mathsf{curr}(\ell) \in s$ iff there exists v s.t. $(\ell, v) \in s$, and if $(\ell, v), (\ell, v') \in s$ then $v = v'$. We call such markings (state) *consistent*. Given a consistent marking of state conditions s, we denote by $\mathsf{hp}(s)$ the heap in s, *i.e.* (the graph of) a partial function with finite domain:

$$\mathsf{hp}(s) = \{(\ell, v) \mid (\ell, v) \in s\}$$

Notations for the kinds of event that might be present in the net $\mathcal{N}\,[\![t]\!]$ are given in Table 1. The set of events for any process shall be extensional: any event is fully described just by its sets of pre- and postconditions. Two operations on events viewed in this way will be of particular use. The first prefixes a 'tag' onto the control conditions of an event. For any event e, the event $1 : e$ (and similarly $2{:}e$) is defined to have exactly the same effect as e on state conditions, ${}^{\mathrm{S}}(1{:}e) = {}^{\mathrm{S}}e$ and $(1{:}e)^{\mathrm{S}} = e^{\mathrm{S}}$, but using the tagged control conditions:

$${}^{\mathbf{C}}(1{:}e) = \{1{:}a \mid a \in {}^{\mathbf{C}}e\} \qquad (1{:}e)^{\mathbf{C}} = \{1{:}a \mid a \in e^{\mathbf{C}}\}$$

The second operation is used to 'glue' two nets together, for example forming $\mathcal{N}\,[\![t_1; t_2]\!]$ by pairing initial conditions of $\mathcal{N}\,[\![t_2]\!]$ with terminal conditions of $\mathcal{N}\,[\![t_1]\!]$. Given a set of control conditions $c \subseteq \mathbf{C}$ and a set $P \subseteq \mathbf{C} \times \mathbf{C}$, define

$$\begin{aligned} P \lhd c &= \{(a, x) \in P \mid a \in c\} \cup \{a \in c \mid \nexists x.(a, x) \in P\} \\ P \rhd c &= \{(x, a) \in P \mid a \in c\} \cup \{a \in c \mid \nexists x.(x, a) \in P\} \end{aligned}$$

This notation is applied to events e, yielding events $P \lhd e$ and $P \rhd e$ with

$$\begin{array}{rl} {}^{\mathbf{S}}(P \lhd e) = {}^{\mathbf{S}}(P \rhd e) = {}^{\mathbf{S}}e & (P \lhd e)^{\mathbf{S}} = (P \rhd e)^{\mathbf{S}} = e^{\mathbf{S}} \\ {}^{\mathbf{C}}(P \lhd e) = P \lhd ({}^{\mathbf{C}}e) & (P \lhd e)^{\mathbf{C}} = P \lhd (e^{\mathbf{C}}) \\ {}^{\mathbf{C}}(P \rhd e) = P \rhd ({}^{\mathbf{C}}e) & (P \rhd e)^{\mathbf{C}} = P \rhd (e^{\mathbf{C}}) \end{array}$$

The definitions of tagging and gluing extend to sets of events in the obvious way (for example, $1{:}E = \{1{:}e \mid e \in E\}$).

3.1 Net Contexts and Substitution

The net semantics for terms can be extended to give a net semantics for term contexts, giving what we call a *net context.* A net context simply specifies a control-point at which an embedded net may be placed.

Definition 2. *A* net context *is an embedded net with a distinguished event denoted* $[-]$ *such that* ${}^{\mathbf{S}}[-] = \emptyset = [-]^{\mathbf{S}}$.

The inductive definition of the net semantics of terms is extended in the obvious way to define a semantics for term contexts, denoted $\mathcal{N}[\![k]\!]$: the interpretation of the term context $-$ is the net context $\mathcal{N}[\![-]\!]$ with a single event $[-]$ with $pre([-]) = \{\mathsf{i}\}$ and $post([-]) = \{\mathsf{t}\}$.

Given a net context K and an embedded net N, we now define an embedded net representing the substitution of N for the hole $[-]$ in K. This is simply obtained by using tagging to force the events of K and N to be disjoint, removing the artificial 'hole' event from K and then 'gluing' N in the appropriate place.

Definition 3. *Let the embedded net* $N = (\mathbf{C}, \mathbf{S}, E_N, pre_N, post_N, I_N, T_N)$ *and let the net context* $K = (\mathbf{C}, \mathbf{S}, E_K, pre_K, post_K, I_K, T_K)$*. Define the subsets of control conditions*

$$P_{\mathsf{init}} = 1{:}pre_K([-]) \times 2{:}I_N \qquad P_{\mathsf{term}} = 1{:}post_K([-]) \times 2{:}T_N.$$

The embedded net $K[N] = (\mathbf{C}, \mathbf{S}, E, pre, post, I, T)$ *is defined as*

$$\begin{array}{c} I = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd 1{:}I_K \qquad T = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd 1{:}T_K \\ E = (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \lhd (1{:}(E_K \setminus \{[-]\})) \cup (P_{\mathsf{init}} \cup P_{\mathsf{term}}) \rhd 2{:}E_N \end{array}$$

An example is shown in Figure 1 (eliding the unaffected state conditions). Note that the conditions $P_{\mathsf{init}} = \{(1:c, 2:w), (1:c, 2:x), (1:d, 2:w), (1:d, 2:x)\}$ are marked when the initial control point of N is reached. Generally, we say that the subprocess N in $K[N]$ is *initialized* in the marking of control conditions c when $P_{\mathsf{init}} \subseteq c$.

Net contexts and net substitution connect with term contexts and substitution through the following lemma.

Lemma 1. *There is a state-preserving isomorphism of embedded nets*

$$\gamma_{k,t} : \mathcal{N}[\![k[t]]\!] \cong \mathcal{N}[\![k]\!]\,[\mathcal{N}[\![t]\!]].$$

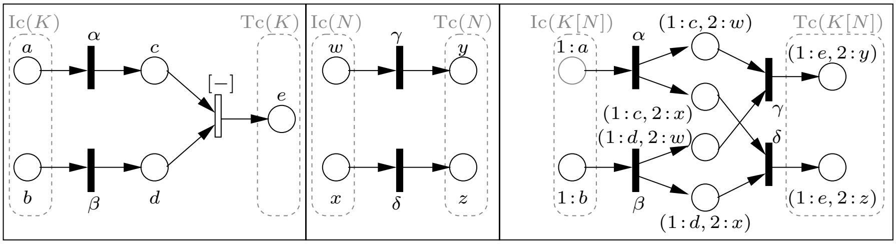


Fig. 1. Application of context K to net N, yielding $K[N]$

It now becomes possible to determine when a subprocess t in $k[t]$ becomes active. Given a marking c of control conditions in $\mathcal{N}\llbracket k[t]\rrbracket$, let $\gamma_{k,t}c$ denote the subset of control conditions of $\mathcal{N}\llbracket k\rrbracket\left[\mathcal{N}\llbracket t\rrbracket\right]$ obtained as the image of the set c under $\gamma_{k,t}$. We shall say that t *is initialized in* c if, under the isomorphism, the set of conditions P_{init} is marked; that is, if $P_{\mathsf{init}} \subseteq \gamma_{k,t}c$. The semantics of terms ensures that if t is initialized in c, no condition inside t is marked and no condition corresponding to a terminal condition of t is marked.

Lemma 2. *For any initial marking* $(\mathrm{Ic}(k[t]), s_0)$ *of* $\mathcal{N}\llbracket k[t]\rrbracket$*, if* (c, s) *is reachable and* $\gamma_{k,t}^{-1}P_{\mathsf{init}} \subseteq c$ *then* $c = \gamma_{k,t}^{-1}P_{\mathsf{init}} \cup 1{:}c_1$ *for some set of control conditions* c_1*.*

The property is shown by consideration of the *control nets* described in [7,6].

4 Concurrent Separation Logic

Concurrent separation logic is a Hoare-style system designed to provide partial correctness judgements about concurrent heap-manipulating programs. We refer the reader to [8] for a full introduction to the logic.[1]

In this section, we briefly summarize the essential parts of the Petri-net model presented in [7,6], to which we refer the reader for the full definition of the syntax and semantics of the logic. A selection of rules is presented in Figure 2. The key judgement is $\Gamma \vdash \{\varphi\}t\{\psi\}$, which has the following interpretation:

> *If initially* φ *holds of the heap defined at the locations* owned *by the process, then, after* t *runs to completion,* ψ *holds of the heap defined at the locations owned by the process; during any such run, the process only accesses locations that it owns and preserves* invariants *in* Γ*.*

At the core of separation logic is the separating conjunction, $\varphi_1 \star \varphi_2$. A heap h satisfies this formula, written $h \models \varphi_1 \star \varphi_2$, if h can be partitioned into disjoint subheaps h_1 and h_2 such that $h_1 \models \varphi_1$ and $h_2 \models \varphi_2$.

The *environment* Γ associates an *invariant* to every resource free in t. An invariant is a *precise* heap formula: a formula χ is said to be precise if given any

[1] Here, we make two simplifications to the logic that are orthogonal to our results: we do not distinguish stack and heap variables and we do not give a rule for declaration of new resources (so the environment Γ is fixed through any derivation).

$$\frac{\begin{array}{r}\forall h \models \varphi : \forall (h_1, h_2) \in \mathcal{A}[\![\alpha]\!] : \qquad\qquad\qquad \\ \forall h' \supseteq h : (h_1 \subseteq h' \implies h_1 \subseteq h) \\ \& \ (h_1 \subseteq h \implies (h \setminus h_1) \cup h_2 \models \psi)\end{array}}{\Gamma \vdash \{\varphi\}\alpha\{\psi\}} \qquad \frac{\begin{array}{c}\Gamma \vdash \{\varphi_1\}t_1\{\psi_1\} \\ \Gamma \vdash \{\varphi_2\}t_2\{\psi_2\}\end{array}}{\Gamma \vdash \{\varphi_1 \star \varphi_2\}t_1 \parallel t_2\{\psi_1 \star \psi_2\}}$$

$$\frac{\Gamma, w : \chi \vdash \{\varphi \star \chi\}t\{\psi \star \chi\}}{\Gamma, w : \chi \vdash \{\varphi\}\texttt{with}\ w\ \texttt{do}\ t\ \texttt{od}\{\psi\}}$$

Fig. 2. Selected rules of concurrent separation logic

heap h, there is at most one heap $h_0 \subseteq h$ such that $h_0 \models \chi$. When a process enters a critical region, it gains ownership of the part of the heap that satisfies the invariant. When it leaves the critical region, it is required to have restored the invariant and it loses ownership of the associated part of the heap. This is reflected in the rule for critical regions. As seen in [8], ownership of locations can be transferred between processes using critical regions. This provides vital power to the logic but introduces subtlety to the notion of ownership since the set of locations that the process owns changes as it executes.

4.1 Interference and Ownership Nets

In order to demonstrate the correctness of the rule for parallel composition, we shall reason about the process running in the presence of arbitrary processes that act only on locations that they are seen to own. Central to this interpretation is a formal treatment of ownership, in which locations are partitioned into three disjoint sets: locations that are owned by the process, locations that are used to satisfy the invariants of available resources and locations that are owned by other processes.

The first stage in the creation of the formal model is the definition of an *interference net*, a net to simulate the behaviour of the arbitrary concurrently-executing processes on the shared state. The constraints on their behaviour are represented through the presence of explicit conditions to represent ownership in the system. Notation to describe the kinds of event present in the interference net is given in Table 2.

Definition 4. *The set of* ownership conditions *is defined to be the set* $\mathbf{W} = \{\omega_{\mathsf{proc}}(x), \omega_{\mathsf{inv}}(x), \omega_{\mathsf{oth}}(x) \mid x \in \mathrm{Loc} \cup \mathrm{Res}\}$. *The* interference net *for* Γ *has conditions* $\mathbf{S} \cup \mathbf{W}$ *and events called* interference events*:*

- $\overline{\mathsf{act}}(h_1, h_2)$ *for all heaps* h_1 *and* h_2 *such that* $\mathrm{dom}(h_1) = \mathrm{dom}(h_2)$
- $\overline{\mathsf{alloc}}(\ell, v, \ell', v')$ *and* $\overline{\mathsf{dealloc}}(\ell, \ell', v')$ *for all* ℓ *and* ℓ' *and values* v *and* v'
- $\overline{\mathsf{acq}}(r, h)$ *and* $\overline{\mathsf{rel}}(r, h)$ *for all* $r \in \mathrm{dom}(\Gamma)$ *and* h *such that* $h \models \chi$*, for* χ *the unique formula such that* $r : \chi \in \Gamma$

We use the symbol u to range over interference events and use w to range over markings of ownership conditions. A marking $\sigma = (s, w)$ of the interference net is said to be *consistent* if σ is consistent and, for each $z \in \mathrm{Loc} \cup$

Table 2. Interference events

Abbreviation	Preconditions	Postconditions
$\overline{\mathsf{act}}(h_1, h_2)$	$h_1 \cup \{\omega_{\mathsf{oth}}(\ell) \mid \ell \in \mathrm{dom}(h_1)\}$	$h_2 \cup \{\omega_{\mathsf{oth}}(\ell) \mid \ell \in \mathrm{dom}(h_1)\}$
$\overline{\mathsf{alloc}}(\ell, v, \ell', v')$	$\{\omega_{\mathsf{oth}}(\ell), (\ell, v)\}$	$\{\omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(\ell'), \mathsf{curr}(\ell'), (\ell, \ell'), (\ell', v')\}$
$\overline{\mathsf{dealloc}}(\ell, \ell', v')$	$\{\omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(\ell'), \mathsf{curr}(\ell'), (\ell, \ell'), (\ell', v')\}$	$\{\omega_{\mathsf{oth}}(\ell), (\ell, \ell')\}$
$\overline{\mathsf{acq}}(r, h)$	$\{\omega_{\mathsf{inv}}(r), r\} \cup h \cup \{\omega_{\mathsf{inv}}(\ell) \mid \exists v.(\ell, v) \in h\}$	$\{\omega_{\mathsf{oth}}(r)\} \cup h \cup \{\omega_{\mathsf{oth}}(\ell) \mid \exists v.(\ell, v) \in h\}$
$\overline{\mathsf{rel}}(r, h)$	$\{\omega_{\mathsf{oth}}(r)\} \cup h \cup \{\omega_{\mathsf{oth}}(\ell) \mid \exists v.(\ell, v) \in h\}$	$\{\omega_{\mathsf{inv}}(r), r\} \cup h \cup \{\omega_{\mathsf{inv}}(\ell) \mid \exists v.(\ell, v) \in h\}$

Res, if either $z \in \mathrm{Res}$ or $\mathsf{curr}(z) \in s$ then there is precisely one condition in $\{\omega_{\mathsf{proc}}(z), \omega_{\mathsf{inv}}(z), \omega_{\mathsf{oth}}(z)\} \cap w$; otherwise, if $z \in \mathrm{Loc}$ and $\mathsf{curr}(z) \notin s$, we require that the set $\{\omega_{\mathsf{proc}}(z), \omega_{\mathsf{inv}}(z), \omega_{\mathsf{oth}}(z)\} \cap w$ be empty. As such, a consistent marking assigns precisely one ownership state to every current location and resource.

The interference net for an environment Γ describes the potential behaviour of other processes that can take place on the state. For example, the interference event $\overline{\mathsf{act}}(h_1, h_2)$ can update the heap only if the locations operated-on are seen as owned by 'other' processes. The interpretation of a judgement $\Gamma \vdash \{\varphi\}t\{\psi\}$ shall therefore consider the net for t running in parallel with the interference net. However, additionally, the symmetry in the rule for parallel composition requires that the behaviour of t can be seen as interference when considering other processes; we establish this by *synchronization* of $\mathcal{N}[\![t]\!]$ with the interference net for Γ. We begin by defining with which interference events an event e of $\mathcal{N}[\![t]\!]$ can synchronize:

– the event $\mathsf{act}_{(c,c')}(h_1, h_2)$ can synchronize with $\overline{\mathsf{act}}(h_1, h_2)$,

– the event $\mathsf{alloc}_{(c,c')}(\ell, v, \ell', v')$ can synchronize with $\overline{\mathsf{alloc}}(\ell, v, \ell', v')$,

– the event $\mathsf{dealloc}_{(c,c')}(\ell, \ell', v')$ can synchronize with $\overline{\mathsf{dealloc}}(\ell, \ell', v')$,

– the event $\mathsf{acq}_{(c,c')}(r)$ can synchronize with $\overline{\mathsf{acq}}(r, h)$ for any h, and

– the event $\mathsf{rel}_{(c,c')}(r)$ can synchronize with $\overline{\mathsf{rel}}(r, h)$ for any h.

Suppose that two events synchronize, e from the process and u from the interference net. The event u is the event that would fire in the net for the other parallel process to simulate the event e. Let $e \cdot u$ be the event formed by taking the union of the preconditions (and, respectively, postconditions) of e and u, other than using $\omega_{\mathsf{proc}}(\ell)$ in place of $\omega_{\mathsf{oth}}(\ell)$, and similarly $\omega_{\mathsf{proc}}(r)$ in place of $\omega_{\mathsf{oth}}(r)$.

Definition 5. *The* ownership net $\mathcal{W}[\![t]\!]_\Gamma$ *is the net with conditions* $\mathbf{C} \cup \mathbf{S} \cup \mathbf{W}$ *and all events that are either events u from the interference net for Γ or synchronized events $e \cdot u$ where e is an event of $\mathcal{N}[\![t]\!]$ and u is an interference event such that e and u can synchronize.*

Markings of ownership nets are tuples (c, s, w) where c, s and w are the markings of, respectively, control, state and ownership conditions. We say that (c, s, w) is consistent if (s, w) is consistent. It is shown in [7,6] that the property of markings being consistent is preserved as processes execute.

4.2 Soundness

The formulation of the ownership net permits a fundamental understanding of when a process acts in a way that would not be seen as interference when considering other processes, by failing to respect ownership or to restore invariants.

Definition 6 (Violating marking). *We say that a consistent marking* (c, s, w) *of* $\mathcal{W}[\![t]\!]_\Gamma$ *is* violating *if there exists an event* e *of* $\mathcal{N}[\![t]\!]$ *that has concession in the marking* (c, s) *but there is no event* u *from the interference net such that* u *synchronizes with* e *and* $e \cdot u$ *has concession in* (c, s, w)*.*

We are now ready to turn to soundness of judgements. Given a state s and environment Γ, let $\mathrm{inv}(\Gamma, s)$ denote the separating conjunction of formulae χ_r s.t. $r \in R$ and $r : \chi_r \in \Gamma$; so $\mathrm{inv}(\Gamma, s)$ is a formula that is satisfied by a heap that can be split into separate parts, each of which satisfies the invariant for a distinct available resource.

Definition 7. *Let* s *be a state containing heap* h*. For any* $L \subseteq \mathrm{Loc}$*, let* $h \restriction L$ *denote the restriction of* h *to* L*, so* $h \restriction L = \{(\ell, v) \mid \ell \in L \text{ and } v \in \mathrm{Val} \text{ and } (\ell, v) \in s\}$*. The marking* (c, s, w) *is said to* satisfy φ in Γ *if* (c, s, w) *is consistent,* $h \restriction \{\ell \mid \omega_{\mathsf{inv}}(\ell) \in w\} \models \mathrm{inv}(\Gamma, s)$*, and* $h \restriction \{\ell \mid \omega_{\mathsf{proc}}(\ell) \in w\} \models \varphi$*.*

We now present soundness of the system: see [7,6] for a proof and also formal results connecting this down to the behaviour of the original process $\mathcal{N}[\![t]\!]$.

Theorem 1. *If* $\Gamma \vdash \{\varphi\}t\{\psi\}$ *then, for any* s *and* w *such that* $(\mathrm{Ic}(t), s, w)$ *satisfies* φ *in* Γ*, no violating marking is reachable from* $(\mathrm{Ic}(t), s, w)$ *in* $\mathcal{W}[\![t]\!]_\Gamma$ *and if* $(\mathrm{Tc}(t), s', w')$ *is reachable from* $(\mathrm{Ic}(t), s, w)$ *then* $(\mathrm{Tc}(t), s', w')$ *satisfies* ψ *in* Γ*.*

5 Separation

In this section, we use the ownership semantics described above to capture how the subprocesses of any proved term can be separated from their environment. In [7,6], the characterization was based solely on independence of events; here, we provide a stronger result based on ownership.

First, we extend the construction of the ownership net to contexts, yielding ownership nets $\mathcal{W}[\![k]\!]_\Gamma$ consisting of events that are either interference events from the interference net for Γ, synchronized events as described above or the hole event $[-]$ drawn from $\mathcal{N}[\![k]\!]$. Note that Lemma 2 extends straightforwardly to ownership nets: Given an initial marking $(\mathrm{Ic}(k[t]), s_0, w_0)$ of an ownership net $\mathcal{W}[\![k[t]]\!]_\Gamma$, any reachable marking (c, s, w) such that t is initialized in c satisfies $c = \gamma_{k,t}^{-1} P_{\mathsf{init}} \cup 1 : c_1$ for some (necessarily unique) subset of control conditions c_1.

Central to characterizing how a term t and context k can be separated is the ability to split their ownership. Let w, w_1 and w_2 be markings of ownership conditions. Then w_1 and w_2 form an *ownership split* of w if for all $z \in \mathrm{Loc} \cup \mathrm{Res}$:

$$\begin{gathered}\omega_{\mathsf{proc}}(z) \in w \iff \omega_{\mathsf{proc}}(z) \in w_1 \cup w_2 \\ \omega_{\mathsf{oth}}(z) \in w \iff \omega_{\mathsf{oth}}(z) \in w_1 \cap w_2 \\ \omega_{\mathsf{inv}}(z) \in w \iff \omega_{\mathsf{inv}}(z) \in w_1 \iff \omega_{\mathsf{inv}}(z) \in w_2\end{gathered}$$

Definition 8 (Separability and subprocess race-freedom). *With respect to an environment Γ, say that k and t are* separable *from (s_0, w_0) if, for any marking (c, s, w) reachable from $(\mathrm{Ic}(k[t]), s_0, w_0)$ in $\mathcal{W}[\![k[t]]\!]_\Gamma$ such that t is initialized in c, there exist w_1 and w_2 forming an ownership split of w satisfying*

- *no violating marking is reachable from (c_1, s, w_1) in $\mathcal{W}[\![k]\!]_\Gamma$ by events excluding $[-]$, where c_1 is the set such that $c = \gamma^{-1}_{k,t} P_{\mathsf{init}} \cup 1\!:\!c_1$, and*
- *no violating marking is reachable from $(\mathrm{Ic}(t), s, w_2)$ in $\mathcal{W}[\![t]\!]_\Gamma$.*

Say that a term t_0 is subprocess race-free *from (s_0, w_0) if, for all k and t such that $t_0 \equiv k[t]$, it is the case that k and t are separable from (s_0, w_0).*

Intuitively, if a marking is encountered in $\mathcal{W}[\![k[t]]\!]_\Gamma$ in which t is initialized, that k and t are separable means that the ownership of the heap and resources can be partitioned between t and k in such a way that the allocation of resources to t is sufficient that it never acts on anything that it does not own, and the allocation of resources to k is sufficient that any action that k can perform prior to the completion of t is constrained to be on the locations that k owns.

Theorem 2. *If $\Gamma \vdash \{\varphi\}t\{\psi\}$ then t_0 is subprocess race-free from any (s_0, w_0) that satisfies φ in Γ.*

5.1 Strength of Race-Freedom

Subprocess race-freedom is unusual in its use of ownership; as we shall see, this is intimately related to the logic and shall be central to our constraint on refinement. We first study its position in a hierarchy of race-freedom properties.

An interesting alternative candidate is to say that a term t_0 is *AAD race-free* from marking (s_0, w_0) if, for any context k and any term t that is either a heap action, an allocation or a deallocation command such that $t_0 \equiv k[t]$, then k and t are separable from (s_0, w_0). (A full treatment of this would extend contexts to allow the hole to occur at guards in loops and the sum.)

AAD race-freedom is a stronger property than the more familiar notions of race-freedom [2,7,6] which require that no two actions can occur concurrently on the same memory location. For example, consider the process t defined as $\ell := 0$ and context k defined as $- \parallel \texttt{alloc}(m); \texttt{dealloc}(m); \texttt{if}\ m = \ell\ \texttt{then}\ \ell := 1\ \texttt{endif}$ (the Boolean $m = \ell$ passes only if m is a pointer to ℓ). From any initial state in which ℓ and m are owned by $k[t]$, it is easy to see that the allocation command can never allocate ℓ so there is never any concurrent access of any memory location. However, these processes are not AAD race-free. To see this, we would certainly have to give ownership of ℓ to the assignment $\ell := 0$. Consequently, in the net $\mathcal{W}[\![k]\!]_\emptyset$, an interference event could occur in which ℓ is deallocated followed by allocation of ℓ by the allocation command and subsequently by the assignment of 1 to the location ℓ which is not owned by the process.

Subprocess race-freedom is an even more discriminating condition than AAD race-freedom. Consider the net $\mathcal{W}[\![k'[t']]\!]_\emptyset$ where

$$t' = \quad n := 0; \texttt{dealloc}(p)$$
$$k' = \quad - \parallel \texttt{alloc}(\ell); \texttt{while}\ (\ell\ \texttt{!=}\ m)\ \texttt{do}\ \texttt{alloc}(\ell)\ \texttt{od}; n := 1$$

running from an initial state with heap $\{(\ell,0),(m,0),(n,0),(p,m)\}$ and ownership marking $\{\omega_{\mathsf{proc}}(\ell),\omega_{\mathsf{proc}}(m),\omega_{\mathsf{proc}}(n),\omega_{\mathsf{proc}}(p)\}$. Going through each action, it can be verified that $k'[t']$ is AAD race-free; the key is that $n := 1$ can only happen after $\texttt{dealloc}(p)$. However, $k'[t']$ is not subprocess race-free: ownership of n, p and m must be given to t', which means that when considering the context $\mathcal{W}\,[\![k']\!]_{\emptyset}$, it becomes possible for an interference event deallocating m to occur, then re-allocation of m by k' followed by assignment to the unowned location n.

Interestingly, the most important known examples of the incompleteness of concurrent separation logic all involve processes that are not subprocess race-free according to the definition here, so one may hope that this tighter form of race-freedom gives new insight towards (relative) completeness.

6 Footprints and Refinement

We have seen that any context k and term t that form a proved process can be separated from suitable initial states. We now introduce an operation of refinement of t by some other process t'. Of course, this is only permitted when the interaction of t and t' with k is restricted; in particular, we shall restrict to processes that do not have critical regions or allocate or deallocate locations.

Definition 9. *A term z is* static *if it follows the grammar*

$$z ::= \alpha \mid z_1; z_2 \mid \alpha_1.z_1 + \alpha_2.z_2 \mid z_1 \parallel z_2 \mid \texttt{while}\, b \,\texttt{do}\, z \,\texttt{od}.$$

The key goal is to show that, for static terms z and z', if $k[z']$ runs from a suitable initial state to a terminal state, there should be a corresponding run of $k[z]$ from the initial state to the same terminal state.

Two constraints shall be necessary when considering whether a static term z' can replace z in a context k. The first is that if z' runs from an initial state s to a terminal state s' then z can also run from s to s'. For any term t, write $t : s \Downarrow s'$ if the marking $(\mathrm{Tc}(t), s')$ is reachable from $(\mathrm{Ic}(t), s)$ in $\mathcal{N}\,[\![t]\!]$. We shall require that if $z' : s \Downarrow s'$ then $z : s \Downarrow s'$.

The second constraint is that, running from any state s, the locations that z' accesses are all locations that z might access. To justify this, consider the following example. It is easy to see that there are no s and s' such that $\ell \mathbin{?=} 0; \ell \mathbin{?=} 1 : s \Downarrow s'$, so the first constraint for using $\ell \mathbin{?=} 0; \ell \mathbin{?=} 1$ to replace $\texttt{false}$ in the process $\texttt{false} \parallel \ell := 1$ would be met. However, the resulting process $\ell \mathbin{?=} 0; \ell \mathbin{?=} 1 \parallel \ell := 1$ has more behaviour, so the refinement is unsound. It should be ruled-out because the command $\texttt{false}$ accesses no locations whereas $\ell \mathbin{?=} 0; \ell \mathbin{?=} 1$ accesses ℓ.

The locations that might be accessed by z are called its *footprint*, which we now capture as the least allocation of ownership of the heap to the process that ensures that no violation is encountered. The notion of footprint has intricacies, but since our interest is in the footprint of static terms, we can avoid many of them. In particular, we do not need to consider ownership of invariants; we shall say that a marking of ownership conditions w is *invariant-empty* if there exists no $z \in \mathrm{Loc} \cup \mathrm{Res}$ such that $\omega_{\mathsf{inv}}(z) \in w$.

Let w and w' be invariant-empty markings of ownership conditions consistent with some state s. Define $w \leq w'$ if $\omega_{\mathsf{proc}}(z) \in w$ implies $\omega_{\mathsf{proc}}(z) \in w'$ for all $z \in \mathrm{Loc} \cup \mathrm{Res}$. For two invariant-empty ownership markings w and w' consistent with the state s, define

$$w \sqcap_s w' = \begin{array}{c} \{\omega_{\mathsf{proc}}(z) \mid \omega_{\mathsf{proc}}(z) \in w \text{ and } \omega_{\mathsf{proc}}(z) \in w'\} \\ \cup\{\,\omega_{\mathsf{oth}}(z) \mid \omega_{\mathsf{oth}}(z) \in w \text{ or } \omega_{\mathsf{oth}}(z) \in w'\,\} \end{array}.$$

It is easy to see that this is consistent and is a least upper bound of w and w' w.r.t. the partial order $\leq$ over invariant-empty ownership markings consistent with s.

Lemma 3. *Let z be a static term. For any s and invariant-empty w and w' such that both (s, w) and (s, w') are consistent, if no violating marking is reachable from either $(\mathrm{Ic}(z), s, w)$ or $(\mathrm{Ic}(z), s, w')$ in $\mathcal{W}\,[\![z]\!]_{\emptyset}$ then no violating marking is reachable from $(\mathrm{Ic}(z), s, w \sqcap_s w')$ in $\mathcal{W}\,[\![t]\!]_{\emptyset}$.*

Recalling that the marking s must be finite, it follows immediately from this lemma that, for any static term z and state s, there exists a *least* (according to the order $\leq$) invariant-empty marking of ownership conditions w consistent with s such that no violating marking is reachable in $\mathcal{W}\,[\![z]\!]_{\emptyset}$ from $(\mathrm{Ic}(z), s, w)$. The locations that must be owned by the process form the footprint of t:

$$\mathsf{footprint}(z, s) = \{\ell \mid \omega_{\mathsf{proc}}(\ell) \in w\}$$

The restriction to static terms in Lemma 3 is important: there are examples of non-static processes for which this property fails. For example, consider the state with heap $\{(k, 0), (l, 0), (m, 0)\}$ and term `while` m `!=` k `do alloc(`m`) od;` ℓ `:=` 1. No violating marking is reachable from either of the initial ownership markings $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{proc}}(\ell), \omega_{\mathsf{oth}}(k)\}$ or $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{proc}}(k)\}$ but a violating marking is reachable from their l.u.b., $\{\omega_{\mathsf{proc}}(m), \omega_{\mathsf{oth}}(\ell), \omega_{\mathsf{oth}}(k)\}$.

We now give a key result, that footprint-respecting refinements give rise to no additional behaviour.

Theorem 3. *Let z and z' be static terms such that, for all states s and s':*

$$z' : s \Downarrow s' \implies z : s \Downarrow s' \qquad \textit{and} \qquad \mathsf{footprint}(z', s) \subseteq \mathsf{footprint}(z, s)$$

Let k be a context such that k and z are separable from (s_0, w_0). Then:

- *k and z' are separable from (s_0, w_0),*
- *if no violating marking is reachable in $\mathcal{W}\,[\![k[z]]\!]_{\Gamma}$ from (s_0, w_0) then no violating marking is reachable in $\mathcal{W}\,[\![k[z']]\!]_{\Gamma}$, and*
- *if the terminal marking $(\mathrm{Tc}(k[z']), s, w)$ is reachable from $(\mathrm{Ic}(k[z']), s_0, w_0)$ in $\mathcal{W}\,[\![k[z']]\!]_{\Gamma}$ then the terminal marking $(\mathrm{Tc}(k[z]), s, w)$ is reachable from $(\mathrm{Ic}(k[z]), s_0, w_0)$ in $\mathcal{W}\,[\![k[z]]\!]_{\Gamma}$.*

The proof proceeds similarly to that for 'non-interfering substitutions' in [7,6], using the fact that any two consecutive occurrences of events whilst z' is active will be independent if one event is from k and one is from z'.

We now show how the property of being subprocess race-free is preserved under the forms of refinement described above.

Theorem 4. *Let z and z' be static terms such such that, for all states s and s':*

- *$z' : s \Downarrow s'$ implies $z : s \Downarrow s'$ and $\mathsf{footprint}(z', s) \subseteq \mathsf{footprint}(z, s)$, and*
- *for any w such that (s, w) is consistent, if z is subprocess race-free from (s, w) then z' is subprocess race-free from (s, w), both taking the environment to be empty.*

For any context k, environment Γ and consistent (s_0, w_0), if $k[z]$ is subprocess race-free from (s_0, w_0) then $k[z']$ is subprocess race-free from (s_0, w_0).

Together, Theorems 3 and 4 show how the validity of judgements is preserved by footprint-preserving refinements of static subprocesses.

We conclude by giving an example of the kind of refinement that is permitted, showing how any static subterm can be refined to its effect. For a static term z define the action $\mathsf{collapse}_z$ as

$$\mathcal{A}\,[\![\mathsf{collapse}_z]\!] \overset{\text{def}}{=} \left\{ (\mathsf{hp}(s), \mathsf{hp}(s')) \;\middle|\; \begin{array}{r} z : s \Downarrow s' \text{ and for all } s_0 \text{ s.t. } s \subseteq s_0 : \\ \mathsf{footprint}(z, s_0) = \mathrm{dom}(\mathsf{hp}(s)) \end{array} \right\}.$$

The action is formed of minimal heaps that represent the fault-avoiding (*i.e.* sufficient that z never accesses an unallocated location) big-step semantics of z. It is easy to see that the conditions for $\mathsf{collapse}_z$ replacing any static subterm z in any context k are met, so the obtained semantics is related to the original semantics by Theorems 3 and 4.

An adaptation of this would be to define two actions, one representing the start of z and the other the end of z. The events of the 'start' action record the part of the heap to be modified by z and the 'end' events would perform the update, yielding a semantics for proved processes following that in [10].

7 Conclusions and Related Work

We have seen how a Petri-net semantics for concurrent separation logic can be used to prove that its judgements are insensitive to the granularity assumed of primitive actions. In particular, through net and term contexts, an interpretation of the subprocesses of programs was introduced and it was shown how ownership can be split between any subprocess and its context in such a way that neither exceeds the constraints imposed by ownership. The issue of granularity was then addressed by showing that if the footprint of a refinement of a subprocess does not exceed the original footprint, the validity of judgements is preserved.

We have seen how refinements of static terms can begin to yield a semantics along the lines of Reynolds' model [10], with static terms being replaced by 'start' and 'end' actions. His model, however, treats races as 'catastrophic'; we directly prove that they cannot occur. This may be of use when considering refinements of separable parts of racy programs. Related to Reynolds' model is Brookes' footstep trace model [1], in which sequences of actions in individual traces are 'collapsed' to their effect. The goal of the footstep model is to move towards logical full abstraction. However, work presented there (and Reynolds') is critically different from that presented here in that no general connection is shown between the behaviour of processes following change in the granularity of actions and the original processes. The model in [1] also bypasses the important issue of interaction between concurrent processes through allocation or deallocation of memory; it assumes that all allocated memory locations are 'fresh', whereas in real implementations the opposite is often the case. Reynolds' model, on the other hand, has no allocation or deallocation at all.

As discussed in the introduction, in [4] it has recently been shown how a range of assumptions relating both to granularity and other aspects of 'relaxed' memory models can be ignored for programs proved in separation logic. The key difference between their model and this one is that the net-based semantics *natively* permits refinements, whereas their model relies on an additional 'parameterization' relation in the semantics. This leads to them having to re-prove concurrent separation logic sound. However, their constraints on their parameterization relations under which the validity of judgements is preserved are similar to those seen here, so it would be interesting to provide a full connection with their work, in particular to consider how the net model can be applied to prove sound the other forms of memory relaxation described there.

There are a number of areas worthy of further investigation, one of which is the extension of refinement beyond static terms. It seems as though a treatment of refinement of terms involving allocation and deallocation would require a more subtle interpretation of footprint, perhaps along the lines of that presented in [9]. More broadly, the net model here and the abstract semantics for concurrent separation logic [3] deserve connection, and thereon, for example, to RGSep [11].

Acknowledgements. I would like to thank Glynn Winskel, Matthew Parkinson and the anonymous referees for useful comments, and gratefully acknowledge the support of the ERC Advanced Grant ECSYM and the Leverhulme Trust.

References

1. Brookes, S.: A grainless semantics for parallel programs with shared mutable data. In: Proc. MFPS XXI. ENTCS (2005)
2. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science 375(1-3) (2007)
3. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proc. LICS 2007. IEEE Press, Los Alamitos (2007)
4. Ferreira, R., Feng, X., Shao, Z.: Parameterized memory models and concurrent separation logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 267–286. Springer, Heidelberg (2010)
5. van Glabbeek, R.J., Goltz, U.: Equivalence notions for concurrent systems and refinement of actions. In: Kreczmar, A., Mirkowska, G. (eds.) MFCS 1989. LNCS, vol. 379. Springer, Heidelberg (1989)
6. Hayman, J.M.: Petri net semantics. Ph.D. thesis, University of Cambridge, Computer Laboratory, available as Technical Report UCAM-CL-TR-782 (2009)
7. Hayman, J.M., Winskel, G.: Independence and concurrent separation logic. Logical Methods in Computer Science 4(1) (2008); special issue for LICS 2006

8. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
9. Raza, M., Gardner, P.: Footprints in local reasoning. Logical Methods in Computer Science 5(2), 1–27 (2009)
10. Reynolds, J.C.: Toward a grainless semantics for shared-variable concurrency. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 35–48. Springer, Heidelberg (2004)
11. Vafeiadis, V., Parkinson, M.: A marriage of rely/Guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
12. Winskel, G., Nielsen, M.: Models for concurrency. In: Handbook of Logic and the Foundations of Computer Science, vol. 4, pp. 1–148. Oxford University Press, Oxford (1995)

Tractable Reasoning in a Fragment of Separation Logic

Byron Cook[1,3], Christoph Haase[2], Joël Ouaknine[2],
Matthew Parkinson[1], and James Worrell[2]

[1] Microsoft Research Cambridge, UK
[2] Department of Computer Science, University of Oxford, UK
[3] Department of Computer Science, Queen Mary University of London, UK

Abstract. In 2004, Berdine, Calcagno and O'Hearn introduced a fragment of separation logic that allows for reasoning about programs with pointers and linked lists. They showed that entailment in this fragment is in coNP, but the precise complexity of this problem has been open since. In this paper, we show that the problem can actually be solved in polynomial time. To this end, we represent separation logic formulae as graphs and show that every satisfiable formula is equivalent to one whose graph is in a particular normal form. Entailment between two such formulae then reduces to a graph homomorphism problem. We also discuss natural syntactic extensions that render entailment intractable.

1 Introduction

Separation logic (SL) [11,14] is an extension of Hoare logic to reason about pointer manipulating programs. It extends the syntax of assertions with predicates describing shapes of memory; aliasing and disjointness can be concisely expressed within these shapes. This extended assertion languages allows elegant and concise hand written proofs of programs that manipulate dynamically allocated data structures. However, generating such proofs in an automated fashion is constrained by the undecidability of separation logic [14]. For that reason, in recent years research has been concentrating on finding decidable fragments of this logic, see e.g. [2,5].

In this paper, we study the SL fragment presented in [2]. This fragment allows for reasoning about structural integrity properties of programs with pointers and linked lists. In [2], the decidability of checking validity of entailments in this logic has been shown. Entailment is the problem to decide whether, given two separation logic assertions α and α', α' holds in every memory model in which α holds. Decidability was shown in model-theoretic terms and by providing a complete syntactic proof theory for this fragment. Based on these theoretical results, Berdine, Calcagno and O'Hearn [3] later developed the tool Smallfoot. This tool decides entailments via a syntactic proof search using the proof theory, however in the worst case an exponential number of proofs have to be explored. The tool demonstrated that SL could be used to automatically verify memory safety of linked list and tree manipulating programs. Based on the

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 235–249, 2011.

success of SMALLFOOT, this approach has been extended to allow automatic inference of specifications of systems code [1,4], to reason about object-oriented programs [7,12], and even to reason about non-blocking concurrent programs [3]. But fundamentally all these tools are based on the same style of syntactic proof theory.

The precise computational complexity of checking entailments was not fully answered in [2]. The authors show that a memory model disproving an entailment is polynomial in the size of the input, thus giving a coNP algorithm. As we are going to show in this paper, entailment can actually be decided in *polynomial time*. To this end, we take a fundamentally different approach to [2]: Instead of reasoning syntactically about formulae, we represent them as graphs in a particular normal form and then compute a homomorphism between those graphs to prove that an entailment holds. It is well-known [8] that computing graph homomorphisms is an NP-complete problem, however our graphs in normal form enjoy some special structural properties that allow one to compute homomorphisms in polynomial time.

This paper is structured as follows: In Section 2 we formally introduce our SL fragment, graphs and the decision problems that we consider. Section 3 then shows how we can compute in polynomial time from a given assertion a graph in normal form that represents the same set of models of the formula. We then show in Section 4 that a homomorphism between graphs in normal form witnesses an entailment, and that such a homomorphism can be computed in polynomial time. Section 5 deals with syntactic extensions that make entailment coNP-hard.

Due to space constraints, we do not present all algorithms and proofs in the main part of this paper, they can however be found in an extended version [6]. Moreover, we assume the reader to be familiar with basic notions and concepts of separation logic. For a comprehensive introduction to separation logic, see [14].

2 Preliminaries

Let Vars and V be countably infinite sets of *variables* and *nodes*. We assume some fixed total order $<$ on Vars and for any finite $S \subseteq \mathsf{Vars}$, denote by $min(S)$ the unique $x \in S$ such that $x \leq y$ for all $y \in S$.

The syntax of our assertion language is given by the following grammar, where x ranges over Vars:

$$\begin{aligned}
expr &::= x && (\mathit{expressions})\\
\phi &::= expr = expr \mid expr \neq expr \mid \phi \wedge \phi && (\mathit{pure\ formulae})\\
\sigma &::= expr \mapsto expr \mid \mathsf{ls}(expr, expr) \mid \sigma * \sigma && (\mathit{spatial\ forumlae})\\
\alpha &::= (\phi; \sigma) && (\mathit{assertions})
\end{aligned}$$

Subsequently, we call formulae of our assertion language *SL-formulae*. An example of an SL-formula is $\alpha = (x \neq y; \mathsf{ls}(x, y) * y \mapsto z)$. It describes memory models

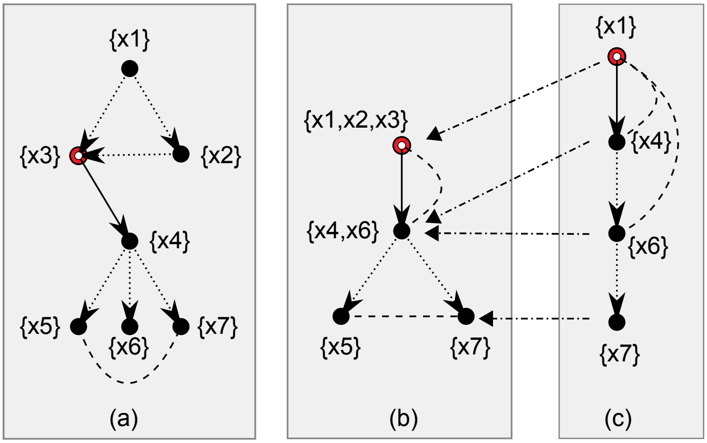


Fig. 1. Three SL-graphs, where l-edges are dotted arrows, p-edges solid arrows and d-edges dashed lines. Nodes are labelled with the variables next to them. The graphs (b) and (c) are in normal form, where (b) is obtained by reducing (a). The arrows from (c) to (b) depict a homomorphism.

in which the value of the stack variable x is not equal to the value of the stack variable y, and in which the heap can be separated into two disjoint segments such that in one segment there is a linked list from the heap cell whose address is the value of x to the heap cell whose address is the value of y, and where in the other segment the latter heap cell points to the heap cell whose address is z. We denote by $|\phi|$ the *size* of a pure formula and by $|\sigma|$ the size of a spatial formula, which is in both cases the number of symbols used to write down the formula. Given an assertion $\alpha = (\phi; \sigma)$, the size of α is $|\alpha| \overset{\text{def}}{=} |\phi| + |\sigma|$. By ϵ, we subsequently denote the empty spatial assertion of size zero.

Remark 1. The SL fragment considered in [2] also contains *nil* as an expression. This does however not give more expressiveness, since we can introduce a designated variable *nil* and implicitly join $nil \mapsto nil$ to every spatial assertion to obtain the same effect.

The semantics of SL-formulae is given in terms of SL-graphs, which we define to be a special class of directed graphs. Throughout this paper, SL-graphs will also be used to represent SL-formulae.

Definition 2. *An* SL-graph *G is either $\bot$ or $(V_b, V_r, E_l, E_p, E_d, \ell)$ such that*

- $V_b, V_r \subseteq_{fin} V$, $V_b \cap V_r = \emptyset$, $V_{b,r} \overset{def}{=} V_b \cup V_r$;
- $E_l \subseteq V_{b,r} \times V_{b,r}$;
- $E_p \subseteq V_r \times V_{b,r}$ *and for every* $v \in V_r$, $E_p(v)$ *is defined;*
- $E_d \subseteq \{\{v, w\} : v, w \in V_{b,r}, v \neq w\}$;
- $\ell : \mathsf{Vars} \rightharpoonup_{fin} V_{b,r}$

Table 1. Semantics of the assertion language, where $\mathcal{I}$ is an SL interpretation

$$\begin{aligned}
\mathcal{I} \models x = y &\iff \ell^{\mathcal{I}}(x) = \ell^{\mathcal{I}}(y) \\
\mathcal{I} \models x \neq y &\iff \ell^{\mathcal{I}}(x) \neq \ell^{\mathcal{I}}(y) \\
\mathcal{I} \models \phi_1 \wedge \phi_2 &\iff \mathcal{I} \models \phi_1 \text{ and } \mathcal{I} \models \phi_2 \\
\mathcal{I} \models x \mapsto y &\iff \exists v, w \in V^{\mathcal{I}}_{b,r}.V^{\mathcal{I}}_r = \{v\}, E^{\mathcal{I}}_p = \{(v,w)\}, \ell^{\mathcal{I}}(x) = v, \ell^{\mathcal{I}}(y) = w \\
\mathcal{I} \models \mathsf{ls}(x,y) &\iff \exists n \in \mathbb{N}.\mathcal{I} \models \mathsf{ls}^n(x,y) \\
\mathcal{I} \models \mathsf{ls}^0(x,y) &\iff \ell^{\mathcal{I}}(x) = \ell^{\mathcal{I}}(y) \text{ and } V^{\mathcal{I}}_r = \emptyset \\
\mathcal{I} \models \mathsf{ls}^{n+1}(x,y) &\iff \exists z \notin \mathit{dom}(\ell^{\mathcal{I}}), v \in V.\mathcal{I}[\ell/\ell[z \mapsto v]] \models x \mapsto z * \mathsf{ls}^n(z,y) \\
\mathcal{I} \models \sigma_1 * \sigma_2 &\iff \exists \mathcal{I}_1, \mathcal{I}_2.\mathcal{I} = \mathcal{I}_1 * \mathcal{I}_2, \mathcal{I}_1 \models \sigma_1, \mathcal{I}_2 \models \sigma_2 \\
\mathcal{I} \models (\phi;\sigma) &\iff \mathcal{I} = \mathcal{I}_1 * \mathcal{I}_2, \mathcal{I}_1 \models \phi \text{ and } \mathcal{I}_1 \models \sigma, \text{ where } \mathcal{I} \models \epsilon \text{ for all } \mathcal{I}
\end{aligned}$$

An SL interpretation *is an SL-graph where $E_l = \emptyset$, E_p is functional and $E_d = \{\{v,w\} : v, w \in V_{b,r}, v \neq w\}$.*

An SL-graph $\bot$ indicates an inconsistent SL-graph. The set $V_{b,r}$ of *nodes* of an SL-graph partitions into sets V_b and V_r, where we refer to nodes in V_b as *black nodes* and to those in V_r as *red nodes*. We call E_p the set of *pointer edges (p-edges)*, E_l the set of *list edges (l-edges)*, E_d is the set of *disequality edges (d-edges)* and ℓ the *variable labelling function*. For convenience, $E_{p,l}$ denotes the set $E_p \cup E_l$. Given a node $v \in V$, we set $\mathit{vars}(v) \overset{\text{def}}{=} \{x \in \mathsf{Vars} : \ell(x) = v\}$ and $\mathit{var}(v) \overset{\text{def}}{=} \mathit{min}(\mathit{vars}(v))$. We sometimes wish to alter one component of a graph and, e.g., write $G[E_p/E'_p]$ to denote the graph $G' = (V_b, V_r, E'_p, E_l, E_d, \ell)$.

Example 3. Figure 1 shows three examples of SL-graphs. Subsequently, we identify nodes of an SL-graph with any of the variables they are labelled with. Graph (a) has an l-edge from the black node x_1 to the red node x_3, depicted by a dotted arrow. The latter node has a p-edge to the black node x_4, depicted by a solid arrow. Moreover, there is a d-edge between x_5 and x_7, depicted by a dashed line.

In the remainder of this paper, we denote an SL interpretation by $\mathcal{I}$ and usually denote the components of an interpretation with superscript $\mathcal{I}$, e.g., we write $V^{\mathcal{I}}_b$ to denote the black nodes of an interpretation $\mathcal{I}$. Given SL interpretations $\mathcal{I}, \mathcal{I}', \mathcal{I}''$, we define $\mathcal{I} = \mathcal{I}' * \mathcal{I}''$ if, and only if, $V^{\mathcal{I}}_r = V^{\mathcal{I}'}_r \uplus V^{\mathcal{I}''}_r$, $V^{\mathcal{I}'}_b = V^{\mathcal{I}}_b \cup V^{\mathcal{I}''}_r$, $V^{\mathcal{I}''}_b = V^{\mathcal{I}}_b \cup V^{\mathcal{I}'}_r$, $E^{\mathcal{I}}_p = E^{\mathcal{I}'}_p \uplus E^{\mathcal{I}''}_p$, and $\ell^{\mathcal{I}} = \ell^{\mathcal{I}'} = \ell^{\mathcal{I}''}$. The semantics of our assertion language is presented in Table 1. We call $\mathcal{I}$ a *model* of α if $\mathcal{I} \models \alpha$.

Remark 4. In [2], the semantics of SL-formulae is given in terms of heaps and stacks. In our setting, we can view the red nodes of an interpretation as the set of allocated heap cells, $E^{\mathcal{I}}_p$ as a representative of the contents of heap cells and $\ell^{\mathcal{I}}$ as the stack. Black nodes then correspond to dangling locations. Moreover, our semantics differs in that we employ the *intuitionistic* model of separation logic [14] and that the semantics of lists is *imprecise*. We will discuss the relationship to the semantics given in [2] in Section 4.

The decision problems of interest to us are *satisfiability* and *entailment*. Given an assertion α, we say α *is satisfiable* if there exists a model $\mathcal{I}$ such that $\mathcal{I} \models \alpha$. Given two assertions α_1 and α_2, we say α_1 *entails* α_2 if for any SL interpretation $\mathcal{I}$, whenever $\mathcal{I} \models \alpha_1$ then $\mathcal{I} \models \alpha_2$. We write $\alpha_1 \models \alpha_2$ if α_1 entails α_2, and $\alpha_1 \equiv \alpha_2$ if $\alpha_1 \models \alpha_2$ and $\alpha_2 \models \alpha_1$.

Given an SL-graph G, we now define its *corresponding assertion* $\alpha(G)$. If $G = \bot$ then $\alpha(G) \overset{\text{def}}{=} (x \neq x; \epsilon)$, i.e., an unsatisfiable SL-formula. Otherwise, the assertion $\alpha(G)$ corresponding to G is defined as follows, where we use an indexed separation operator:

$$\phi(G) \overset{\text{def}}{=} \bigwedge_{\substack{v \in V_{b,r} \\ x,y \in vars(v)}} x = y \wedge \bigwedge_{\{v,w\} \in E_d} var(v) \neq var(w),$$

$$\sigma(G) \overset{\text{def}}{=} \left(*_{(v,w) \in E_p} var(v) \mapsto var(w)\right) * \left(*_{(v,w) \in E_l} \mathsf{ls}(var(v), var(w))\right),$$

$$\alpha(G) \overset{\text{def}}{=} (\phi(G), \sigma(G)).$$

We define the *size* of an SL-graph G as $|G| \overset{\text{def}}{=} |\alpha(G)|$.

Example 5. Graph (b) of Figure 1 corresponds to the assertion $(x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_6 \wedge x_1 \neq x_4 \wedge x_5 \neq x_7; x_1 \mapsto x_4 * \mathsf{ls}(x_4, x_5) * \mathsf{ls}(x_4, x_7))$, where we have omitted superfluous equalities.

We now give some technical definitions about paths in SL-graphs. Given a relation $E \subseteq V \times V$, a *v-w path in E of length n* is a sequence of nodes $\pi : v_1 \cdots v_{n+1}$ such that $v_1 = v$, $v_{n+1} = w$ and $(v_i, v_{i+1}) \in E$ for all $1 \leq i \leq n$. We write $|\pi|$ to denote the length of π. The *edges traversed by* π is defined as $edges(\pi) \overset{\text{def}}{=} \{(v_i, v_{i+1}) : 1 \leq i \leq n\}$. Two paths π_1, π_2 are *distinct* if $edges(\pi_1) \cap edges(\pi_2) = \emptyset$. If $v \neq w$, we call a v-w path *loop-free* if $v_i \neq v_j$ for all $1 \leq i \neq j \leq n+1$. We write $v \rightsquigarrow_p w$, $v \rightsquigarrow_l w$ and $v \rightsquigarrow_{p,l} w$ if there exists a v-w path in E_p, E_l respectively $E_{p,l}$. Moreover, we write $v \rightarrow_p w$, $v \rightarrow_l w$ and $v \rightarrow_{p,l} w$ if $(v,w) \in E_p$, $(v,w) \in E_l$ respectively $(v,w) \in E_{p,l}$. Given a set of edges E, $V(E)$ denotes the set $V(E) \overset{\text{def}}{=} \{v : \exists w.(v,w) \in E \text{ or } (w,v) \in E\}$. As usual, E^* denotes the reflexive and transitive closure of E. For $e = (v,w) \in E$, we define $E^*(e) \overset{\text{def}}{=} \{u : (w,u) \in E^*\} \cup \{v\}$.

The challenging aspect in giving a polynomial time algorithm to decide entailment is that our logic is *non-convex*. As has already been observed in [2], given $\alpha = (y \neq z; \mathsf{ls}(x,y) * \mathsf{ls}(x,z))$, for any model $\mathcal{I}$ of α we have $\mathcal{I} \models (x = y; \epsilon)$ or $\mathcal{I} \models (x = z; \epsilon)$. However there are models $\mathcal{I}_1, \mathcal{I}_2$ of α such that $\mathcal{I}_1 \not\models (x = y; \epsilon)$ and $\mathcal{I}_2 \not\models (x = z; \epsilon)$. Non-convexity often makes computing entailment coNP-hard for logics that contain predicates for describing reachability relations on graphs, e.g., in fragments of XPath or description logics [13,10]. However, in our SL fragment we obtain tractability through the SL-graph normal form we develop in the next section and the fact that variable names only occur at exactly one node in an SL-graph, which fully determines a graph homomorphism if it exists.

3 A Normal Form of SL-Graphs

In this section, we show that given an assertion α we can compute in polynomial time an SL-graph G in a *normal form* such that $\alpha \equiv \alpha(G)$. This normal form serves three purposes: First, it makes implicit equalities and disequalities from α explicit. Second, an SL-graph in normal form has the structural property that if there is a loop-free path between two distinct vertices then there is exactly one such path. Third, any SL-graph $G \neq \bot$ in normal form can be transformed into an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \alpha(G)$, thus showing that satisfiability in our SL fragment is in polynomial time.

First, we show how given a pure formula ϕ we can construct a corresponding graph G_ϕ such that $(\phi, \epsilon) \equiv \alpha(G_\phi)$. Let $\{x_1, \ldots, x_m\} \subseteq \mathsf{Vars}$ be the set of all variables occurring in ϕ, and let $\{[e_1], \ldots, [e_n]\}$ be the set of all equivalence classes of variables induced by ϕ, i.e., $x, y \in [e_i]$ if, and only if, ϕ implies $x = y$. Let $V_b \overset{\text{def}}{=} \{v_1, \ldots, v_n\} \subseteq V$; $\ell(x) \overset{\text{def}}{=} v_i$ if, and only if, $x \in [e_i]$; and $E_d \overset{\text{def}}{=} \{\{v_i, v_j\} : \exists x, y \in \mathsf{Vars}.x \in [e_i], y \in [x_j] \text{ and } x \neq y \text{ occurs in } \phi\}$. If there is a singleton set in E_d then set $G_\phi \overset{\text{def}}{=} \bot$, otherwise $G_\phi \overset{\text{def}}{=} (V_b, \emptyset, \emptyset, \emptyset, E_d, \ell)$. The following lemma can now easily be verified.

Lemma 6. *Let ϕ be a pure formula. There exists a polynomial time computable SL-graph G_ϕ such that $\alpha(G_\phi) \equiv (\phi, \epsilon)$.*

Next, we show how to deal with spatial assertions. When processing spatial assertions and transforming SL-graphs into normal form, we need to manipulate SL-graphs. The two operations we perform on them are *merging nodes* and *removing edges*. Due to space constraints, we relegate details of the algorithms that implement these operations to the extended version of this paper [6].

Algorithm $\textsc{Merge}(G, v, w)$ takes an SL-graph G as input and merges the node w into node v by adding all labels from w to the labels of v and appropriately updating E_l, E_p and E_d. Moreover, the algorithm makes sure that if either $v \in V_r$ or $w \in V_r$ then $v \in V_r$ in the returned graph. If both $v, w \in V_r$ or $\{v, w\} \in E_d$ then $\textsc{Merge}(G, v, w)$ returns $\bot$. Thus, $\textsc{Merge}$ is characterised as follows: If $\alpha(G) = (\phi; \sigma)$, $v, w \in V_{b,r}$, $x = \mathit{var}(v)$ and $y = \mathit{var}(w)$ then $\alpha(\textsc{Merge}(G, v, w)) \equiv (\phi \wedge x = y; \sigma)$.

Algorithm $\text{L}\textsc{Remove}(G, (v, w))$ takes an SL-graph G as input and removes the l-edge (v, w) from G. Likewise, $\text{P}\textsc{Remove}(G, (v, w))$ removes a p-edge from G and, if necessary, moves v from V_r to V_b. Both algorithms can be characterised as follows: If $\alpha(G) = (\phi; \sigma * \mathsf{ls}(x, y))$, $v, w \in V_{b,r}$, $x = \mathit{var}(v)$ and $y = \mathit{var}(w)$ then $\alpha(\text{L}\textsc{Remove}(G, (v, w))) \equiv (\phi; \sigma)$. If $\alpha(G) = (\phi; \sigma * x \mapsto y))$ then $\alpha(\text{P}\textsc{Remove}(G, (v, w))) \equiv (\phi; \sigma)$, where v, w, x and y are as before. As an abbreviation, we introduce $\text{L}\textsc{ReMerge}(G, (v, w))$ and $\text{P}\textsc{ReMerge}(G, (v, w))$ which first remove an l- respectively p-edge (v, w) from G and then merge w into v.

Finally, Algorithm $\textsc{Apply}(G, \sigma)$ takes an SL-graph G and a single spatial assertion $\sigma \in \{x \mapsto y, \mathsf{ls}(x, y)\}$ as input and outputs an SL-graph G' such that if $\alpha(G) = (\phi; \sigma')$ then $\alpha(G') \equiv (\phi; \sigma' * \sigma)$. Again, the concrete algorithm can be

Algorithm 1. Reduce

```
Require: $G$
  while $G$ is not reduced do
    case split on violated condition at node $v$
    // conditions are as in Table 2
    // node names below refer in each case to the corresponding case in Lemma 13
    case (i): return $\bot$
    case (ii): $G = \text{LReMerge}(G, (v, w'))$
    case (iii): $G = \text{LReMerge}(G, (v, w''))$
    case (iv):  $G = \text{Merge}(G', v, w)$
  end while
  return  $G$
```

found in the extended version due to space limitations, but it is not difficult to construct such an algorithm that runs in polynomial time. Some extra care has to be taken if an l-edge is added that is already present in G, since $(\phi; \sigma * \mathsf{ls}(x, y) * \mathsf{ls}(x, y)) \equiv (\phi \wedge x = y; \sigma * \mathsf{ls}(x, y))$. By combining all algorithms considered in this section, we obtain the following lemma.

Lemma 7. *Let α be an SL-graph. Then there exists a polynomial-time algorithm that computes an SL-graph G such that $\alpha \equiv \alpha(G)$.*

We now move towards defining the normal form of an SL-graph and show that any SL-graph can be transformed into one in normal form such that their corresponding assertions are equivalent. A key concept of the normal form is that of a persistent set of edges.

Definition 8. *Let G be an SL-graph, a set of edges $E \subseteq E_{p,l}$ is* persistent *if $V(E) \cap V_r \neq \emptyset$ or there are $v, w \in V(E)$ such that $\{v, w\} \in E_d$.*

For example, let e_1 be the l-edge from x_4 to x_5 and e_2 the l-edge from x_4 to x_7 of graph (a) in Figure 1. Neither $\{e_1\}$ nor $\{e_2\}$ is persistent, but $\{e_1, e_2\}$ is as there is a d-edge between x_5 and x_7. Intuitively, the idea behind the definition is as follows: Suppose we are given an SL-graph G with $(v, w) \in E_l$ such that $E = E^*_{p,l}(v, w)$ is persistent. Then in any model $\mathcal{I}$ of $\alpha(G)$ for $v' = \ell^{\mathcal{I}}(var(v))$, we have $v' \in V^{\mathcal{I}}_r$ since v' must have an outgoing p-edge as the persistence property enforces that there is a p-edge in E or that not all variable names occurring in E are mapped to v' in $\mathcal{I}$. Moreover, if v has a further outgoing l-edge (v, w') then $\ell^{\mathcal{I}}(var(w')) = v$ since v can only have one outgoing p-edge in $\mathcal{I}$. For graph (a) in Figure 1, this means that x_6 becomes equivalent to x_4 in any model of the corresponding SL-formula. Thus persistency allows us to make some implicit equalities in G explicit.

Definition 9. *An SL-graph G is reduced if $G = \bot$ or if it fulfils the conditions in Table 2.*

The definition of a reduced SL-graph is the first step towards the normal form of SL-graphs. Table 2 consists of four conditions, and the idea is that if any of

Table 2. Conditions for an SL-graph G to be reduced

(i) if $v \in V_r$ then $|E_p(v)| = 1$
(ii) if $v \rightarrow_{p,l} w$ such that $E^*_{p,l}(v, w)$ is persistent then $E_l(v) \subseteq \{w\}$
(iii) if $v \rightarrow_l w_1$ and $v \rightarrow_l w_2$ such that $E^*_{p,l}(v, w_1) \cup E^*_{p,l}(v, w_2)$ is persistent then $E_l(v) \subseteq \{w_1, w_2\}$
(iv) there are no distinct loop-free v-w paths π_1, π_2 in E_l.

those conditions is violated by an SL-graph G then we can make some implicit facts explicit. Clearly, if (i) is violated then $\alpha(G)$ is unsatisfiable as the spatial part of $\alpha(G)$ consists of a statement of the form $x \mapsto y * x \mapsto z$. If (ii) or (iii) is violated then by the previous reasoning any further outgoing l-edge can be collapsed into v. Condition (iv) contributes to making sure that between any two different nodes there is at most one loop-free path, as can be seen by the following lemma.

Lemma 10. *Let $G \neq \bot$ be a reduced SL-graph, v, w be distinct nodes in $V_{b,r}$ and $\pi : v \rightsquigarrow_{l,r} w$ a loop-free path. Then π is the unique such loop-free path.*

Proof. To the contrary, assume that there are two different loop-free v-w paths π_1, π_2. Then there are nodes v', w' such that there are distinct v'-w' paths π'_1 and π'_2 that are segments of π_1 respectively π_2, where at least one of π_1 or π_2 is of non-zero length. If $v' = w'$ then this contradicts to π_1 or π_2 being loop-free. Thus, assume $v' \neq w'$. If both π'_1, π'_2 are l-paths then this contradicts to G being reduced, as condition (iv) is violated. Otherwise, if π'_1 reaches a red node then $\mathit{edges}(\pi'_1)$ is persistent and hence v' has one outgoing edge, contradicting to π'_1 and π'_2 being distinct. The case when π'_2 reaches a red node is symmetric.

It is easy to see that deciding whether a graph G is reduced can be performed in polynomial time in $|G|$. In order to transform an arbitrary SL-graph into a reduced SL-graph, Algorithm Reduce just checks for a given input G if any condition from Table 2 is violated. If this is the case, the algorithm removes edges and merges nodes, depending on which condition is violated, until G is reduced. We will subsequently prove Reduce to be correct. First, we provide two technical lemmas that will help us to prove correctness. They allow us to formalise our intuition about persistent sets of edges. Due to space constraints, we omit the proof of the following lemma.

Lemma 11. *Let G be an SL-graph and $v, w, w' \in V_{b,r}$ such that $x = \mathit{var}(v)$, $y = \mathit{var}(w)$, $v \rightsquigarrow_l w$, and let $\mathcal{I}$ be a model of $\alpha(G)$. Then the following holds:*

(i) if $\ell^{\mathcal{I}}(y) \in V_r^{\mathcal{I}}$ then $\ell^{\mathcal{I}}(x) \in V_r^{\mathcal{I}}$; and
(ii) if $v \rightsquigarrow_l w'$ and $\{w, w'\} \in E_d$ then $\ell^{\mathcal{I}}(x) \in V_r^{\mathcal{I}}$.

Lemma 12. *Let $\alpha = (\phi, \sigma)$ and $x \in \mathsf{Vars}$ be such that for all models $\mathcal{I}$ of α, $\ell^{\mathcal{I}}(x) \in V_r^{\mathcal{I}}$. Then for all $y \in \mathsf{Vars}$ and $\alpha' = (\phi, \sigma * \mathsf{ls}(x, y)), \alpha'' = (\phi \wedge x = y, \sigma)$, we have $\alpha' \equiv \alpha''$.*

Proof. We clearly have that $\alpha'' \models \alpha'$. For the other direction, let $\mathcal{I}'$ be a model of α'. By definition, there are $\mathcal{I}_1, \mathcal{I}_2$ such that $\mathcal{I}' = \mathcal{I}_1 * \mathcal{I}_2$, $\mathcal{I}_1 \models (\phi; \sigma)$ and $\mathcal{I}_2 \models (\phi; \mathsf{ls}(x, y))$. By assumption, $\ell^{\mathcal{I}_1}(x) \in V_r^{\mathcal{I}_1}$ and hence $\ell^{\mathcal{I}_2}(x) \notin V_r^{\mathcal{I}_2}$. Consequently, $\ell^{\mathcal{I}_2}(x) = \ell^{\mathcal{I}_2}(y)$. Hence $\ell^{\mathcal{I}'}(x) = \ell^{\mathcal{I}'}(y)$, which yields $\mathcal{I}' \models (\phi \wedge x = y; \sigma)$.

We are now prepared to show the correctness of Reduce. Each case in the lemma below captures a violated condition from Table 2 and shows that the manipulation performed by Reduce is sound and correct.

Lemma 13. *Let G be an SL-graph,*

(i) if there is $v \in V_r$ such that $|E_p(v)| > 1$ then $\alpha(G)$ is unsatisfiable;
(ii) if there are $v, w, w' \in V_{b,r}$, $x, y \in \mathsf{Vars}$ such that $v \rightarrow_{p,l} w$, $v \rightarrow_l w'$, $x = var(v)$, $y = var(w')$, $E^*_{p,l}(v, w)$ is persistent and $\alpha(G) = (\phi, \sigma * \mathsf{ls}(x, y))$ then $\alpha(G) \equiv (\phi \wedge x = y; \sigma)$;
(iii) if there are $v, w, w', w'' \in V_{b,r}$, $x, y \in \mathsf{Vars}$ such that $v \rightarrow_l w$, $v \rightarrow_l w'$, $v \rightarrow_l w''$, $x = var(v)$, $y = var(w'')$, $E^*_{p,l}(v, w) \cup E^*_{p,l}(v, w')$ is persistent and $\alpha(G) = (\phi, \sigma * \mathsf{ls}(x, y))$ then $\alpha(G) \equiv (\phi \wedge x = y; \sigma)$;
(iv) if there are $v, w \in V_b$, $x, y \in \mathsf{Vars}$ such that $x = var(v)$, $y = var(w)$, $\alpha(G) = (\phi, \sigma)$ and there are distinct loop-free v-w l-paths π_1, π_2 in E_l then $\alpha(G) \equiv (\phi \wedge x = y; \sigma)$.

Proof. Case (i): Let $x = var(v)$; we have that there are $y, z \in \mathsf{Vars}$ such that $(\phi; \sigma * x \mapsto y * x \mapsto z)$, which clearly is unsatisfiable.

Case (ii): We show that for all models $\mathcal{I}$ of $\alpha(G)$, $\ell^{\mathcal{I}}(x) \in V_r$. The statement then follows from Lemma 12. If there is $u \in V(E^*_{p,l}(v, w)) \cap V_r$ then by Lemma 11(i) we have $x \in V_r^{\mathcal{I}}$. Otherwise, if there are $u, u' \in V(E^*_{p,l}(v, w))$ such that $\{u, u'\} \in E_d$ then Lemma 11(ii) gives $x \in V_r^{\mathcal{I}}$.

Case (iii): Again, we show that for all models $\mathcal{I}$ of $\alpha(G)$, $\ell^{\mathcal{I}}(x) \in V_r$. The statement then follows from Lemma 12. It is sufficient to consider the case in which there are $u, u' \in V_{b,r}$ such that $w \rightsquigarrow_l u$, $w \rightsquigarrow_l u'$ and $\{u, u'\} \in E_d$ as all other cases are subsumed by (ii). But then, Lemma 11(ii) again yields $x \in V_{b,r}^{\mathcal{I}}$.

Case (iv): Let $\pi_1 = vw_1 \cdot \pi_1'$ and $\pi_2 = vw_2 \cdot \pi_2'$ be v-w paths. Thus, $w_1 \neq w_2$ and hence $m \overset{\text{def}}{=} |\pi_1| + |\pi_2| \geq 3$. We show the statement by induction on m. For $m = 3$, the statement follows from a similar reasoning as in Lemma 12. For the induction step, let $m > 3$ and $\mathcal{I}$ be model of $\alpha(G)$. Let $y_1 = var(w_1)$ and $y_2 = var(w_2)$, we have that $\alpha(G) = (\phi; \sigma * \mathsf{ls}(x, y_1) * \mathsf{ls}(x, y_2))$ and consequently $\mathcal{I} \models \sigma * \mathsf{ls}^{n_1}(x, y_1) * \mathsf{ls}^{n_2}(x, y_2)$ for some $n_1, n_2 \in \mathbb{N}$. If $n_1 = 0$ then $\mathcal{I} \models G'$, where $G' = \text{LReMerge}(G, (v, w_1))$ and the induction hypothesis yields $\ell^{\mathcal{I}}(x) = \ell^{\mathcal{I}}(y)$. The case $n_2 = 0$ follows symmetrically.

Lemma 14. *Let G, G' be SL-graphs such that $G' =$ Reduce(G). Then G' is reduced and $\alpha(G) \equiv \alpha(G')$. Moreover,* Reduce *runs in polynomial time on any input G.*

Proof. Clearly, Reduce only returns graphs that are reduced. Moreover, Lemma 13 shows that in every iteration equivalent graphs are generated and hence

$\alpha(G) \equiv \alpha(G')$. Regarding the complexity, checking if G is reduced can be performed in polynomial time in $|G|$. Removing edges and merging nodes in the **while** body can also be performed in polynomial time. Moreover, the size of G strictly decreases after each iteration of the **while** body. Hence the **while** body is only executed a polynomial number of times.

A nice property of reduced SL-graphs is that they allow to easily construct a model of their corresponding SL-formulae.

Lemma 15. *Let $G \neq \bot$ be a reduced SL-graph and $v, w \in V_{b,r}$ such that $v \neq w$. Then $\alpha(G)$ has a model $\mathcal{I}$ such that $\ell^{\mathcal{I}}(var(v)) \neq \ell^{\mathcal{I}}(var(w))$ and for all $x, y \in \mathsf{Vars}$, $\ell(x) = \ell(y)$ implies $\ell^{\mathcal{I}}(x) = \ell^{\mathcal{I}}(y)$.*

Proof. We sketch how G can iteratively be turned into a desired model $\mathcal{I}$. Suppose w is reachable from v and let π be the loop-free path from v to w. First, we replace any l-edge occurring on π by *two* consecutive p-edges. For all nodes $v' \neq w$ along π that have further outgoing l-edges, we merge all nodes reachable via l-paths from v' into v' and remove the connecting l-edges. If v is reachable from w via a loop-free path π', we apply the same procedure to π'. Finally, we iterate the following procedure: if there is a node u with more than one outgoing l-edge, we fix an l-edge e and merge all nodes reachable from u via the remaining l-edges different from e into u and remove the connecting l-edges. We then replace e with two new consecutive p-edges. Once this procedure has finished, we obtain an SL-graph containing no l-edges that can be turned into an interpretation $\mathcal{I}$. It is easily checked that $\mathcal{I}$ is a model of $\alpha(G)$ and $\ell^{\mathcal{I}}(var(v)) \neq \ell^{\mathcal{I}}(var(w))$.

Theorem 16. *Satisfiability of SL-formulae is decidable in polynomial time.*

Remark 17. When we expand l-edges in the proof of Lemma 15, we replace them by two consecutive p-edges. When we consider entailment in the next section, this will make sure that we obtain a model in which $\mathsf{ls}(x, y)$ holds, but $x \mapsto y$ does not hold. This corresponds to the observation made in [2] that in order to find a counter-model of an entailment, each l-edge has to be expanded at most to length two.

Finally, we can now define our normal form. An SL-graph is in normal form if it is reduced and if its set of disequalities is maximal. Note that in particular any interpretation $\mathcal{I}$ is an SL-graph in normal form.

Definition 18. *Let G be an SL-graph. Then G is in* normal form *if G is reduced and for all $v, w \in V_{b,r}$ such that $\alpha(G) = (\phi; \sigma)$, $x = var(v)$ and $y = var(w)$, whenever $(\phi \wedge x = y; \sigma)$ is unsatisfiable then $\{v, w\} \in E_d$.*

Proposition 19. *For any SL-formula α, there exists a polynomial time computable SL-graph G in normal form such that $\alpha \equiv \alpha(G)$.*

Proof. Given an assertion $\alpha = (\phi; \sigma)$, by Lemma 7 we can construct an SL-graph G' such that $\alpha(G') \equiv \alpha$. Applying Reduce to G' yields a reduced graph G''

such that $\alpha(G') \equiv \alpha(G'')$. In order to bring G'' into normal form, we check for each of the polynomially many pairs $v, w \in V_{b,r}$ if REDUCE returns $\perp$ on input MERGE(G'', v, w). If this is the case, we add $\{v, w\}$ to E_d, which finally gives us the desired graph G. As argued before, all constructions can be performed in polynomial time.

We close this section with an example. Graph (b) in Figure 1 is in normal form and obtained from the graph (a) by applying REDUCE. Graph (a) violates condition (iii) as $\{(\ell(x_4), \ell(x_5)), (\ell(x_4), \ell(x_7)\}$ is persistent, which results in REDUCE merging x_6 into x_4. Moreover, the graph also violates condition (iv) since there are two distinct l-paths from x_1 to x_3. Hence, REDUCE merges x_1 and x_3 and then removes all newly obtained outgoing l-edges from x_3 due to a violation of condition (ii). Finally, $\{(\ell(x_3), \ell(x_4))\}$ is added to E_d in order to obtain graph (b) as merging the nodes x_3 and x_4 and applying REDUCE results in an inconsistent graph.

4 Computing Entailment via Homomorphisms between SL-Graphs in Normal Form

In this section, we show that entailment between SL-formulae can be decided by checking the existence of a graph homomorphism between their corresponding SL-graphs in normal form. Throughout this section, we will assume that all SL-formulae considered are satisfiable and all SL-graphs $G \neq \perp$, since deciding entailment becomes trivial otherwise, and checking for satisfiability can be done in polynomial time.

A homomorphism is a mapping between the nodes of two SL-graphs that, if it exists, preserves the structure of the source graph in the target graph. In the definition of a homomorphism, we make use of the property of SL-graphs in normal form that between any disjoint nodes there is at most one loop-free path connecting the two nodes. For nodes $v \neq w$, we denote this path by $\pi(v, w)$ if it exists. If $v = w$ then $\pi(v, w)$ is the zero-length path $\pi(v, w) \stackrel{\text{def}}{=} v$.

Definition 20. *Let G, G' be SL-graphs in normal form. A mapping $h : V_{b,r} \to V'_{b,r}$ is a* homomorphism *from G to G' if the homomorphism conditions from Table 3 are satisfied.*

Given a mapping h, it is easy to see that checking whether h is a homomorphism can be performed in polynomial time in $|G| + |G'|$. The goal of this section is to prove the following proposition, which gives us the relationship between homomorphisms and entailment.

Proposition 21. *Let G, G' be SL-graphs in normal form. Then $\alpha(G') \models \alpha(G)$ if, and only if, there exists a homomorphism h from G to G'.*

Before we begin with formally proving the proposition, let us discuss its validity on an intuitive level. Suppose there is a homomorphism from G to G'. Condition

Table 3. Conditions for a homomorphism h from G to G'

(i) $\mathit{vars}(v) \subseteq \mathit{vars}(h(v))$
(ii) if $\{v, w\} \in E_d$ then $\{h(v), h(w)\} \in E'_d$
(iii) if $v \to_p w$ then $h(v) \to'_p h(w))$
(iv) if $v \to_l w$ then $h(v) \rightsquigarrow'_{p,l} h(w)$
(v) for all $v_1 \to_{p,l} w_1$ and $v_2 \to_{p,l} w_2$ such that $(v_1, w_1) \neq (v_2, w_2)$, $\mathit{edges}(\pi(h(v_1), h(w_1))) \cap \mathit{edges}(\pi(h(v_2), h(w_2))) = \emptyset$
(vi) if $v, w \in V_r$ and $v \neq w$ then $h(v) \neq h(w)$

(i) makes sure that for any node v of G its image under h is labelled with at least the same variables. If this were not the case, we could easily construct a counter-model of $\alpha(G')$ disproving entailment. Likewise, condition (ii) ensures that whenever two nodes are required to be not equivalent, the same is true for the two nodes under the image of h. Since G' is in normal form, merging the two nodes in the image of h would otherwise be possible since E'_d is maximal. Condition (iii) requires that whenever there is a p-edge between any two nodes v, w, such an edge also exists in G'. Again, it is clear that if this were not the case we could construct a counter-model $\mathcal{I}$ of $\alpha(G')$ such that there is no p-edge between $\ell^{\mathcal{I}}(\mathit{var}(v))$ and $\ell^{\mathcal{I}}(\mathit{var}(w))$. Condition (iv) is of a similar nature, but here we allow that there is a whole path between $h(v)$ and $h(w)$. In condition (v), we require that the paths obtained from the image of two disjoint edges do not share a common edge in G'. If this were the case, we could construct a model of $\alpha(G')$ in which separation is violated. Finally, condition (vi) makes sure that no two different nodes from V_r are mapped to the same node. This condition is needed to handle p-edges of the form (v, v), which may not be covered by condition (v). We now proceed with formally proving Proposition 21. First, the following lemma shows the relationship between models and homomorphisms and that homomorphisms can be composed.

Lemma 22. *Let G, G', G'' be SL-graphs in normal form and $\mathcal{I}$ an interpretation. Then the following holds:*

(i) let $h : V_{b,r} \to V^{\mathcal{I}}_{b,r}$ be such that for all $v \in V_{b,r}$, $h(v) \overset{def}{=} \ell^{\mathcal{I}}(\mathit{var}(v))$; then $\mathcal{I} \models \alpha(G)$ if, and only if, h is a homomorphism from G to $\mathcal{I}$; and
(ii) given homomorphisms h', h'' from G' to G respectively G'' to G'; then $h \overset{def}{=} h'' \circ h'$ is a homomorphism from G'' to G.

The proof of the lemma is rather technical but not difficult and deferred to the extended version of this paper. Proposition 21 now is a consequence of the following lemma. Note that the homomorphism is fully determined by G and G'.

Lemma 23. *Let G, G' be SL-graphs in normal form and let $h : V_{b,r} \to V'_{b,r}$ be defined as $h(v) \overset{def}{=} \ell'(\mathit{var}(v))$ for all $v \in V_{b,r}$. Then $\alpha(G') \models \alpha(G)$ if, and only if, h is a homomorphism from G to G'.*

Proof. ("$\Leftarrow$") Let h be a homomorphism from G to G' and $\mathcal{I}$ be such that $\mathcal{I} \models \alpha(G')$. By Lemma 22(i), there exists a homomorphism h' from G' to $\mathcal{I}$. By Lemma 22(ii), $h'' \overset{\text{def}}{=} h' \circ h$ is a homomorphism from G to $\mathcal{I}$. Consequently, Lemma 22(i) yields $\mathcal{I} \models \alpha(G)$.

("$\Rightarrow$") Due to space constraints, we defer the full proof of this direction to the appendix. The direction is shown via the contrapositive by constructing a counter-model depending on which homomorphism condition from Table 3 is violated, as discussed earlier in this section.

We can now combine all results of this paper so far. Given satisfiable SL-formulae α and α', by Proposition 19 we can compute in polynomial time SL-graphs G and G' in normal form such that $\alpha \equiv \alpha(G)$ and $\alpha' \equiv \alpha(G')$. Next, we can compute in polynomial time a mapping h from $\alpha(G')$ to $\alpha(G)$ and check in polynomial time whether h is a homomorphism. By the previous lemma, this then is the case if, and only if, $\alpha \models \alpha'$.

Theorem 24. *Entailment between SL-formulae is decidable in polynomial time.*

An example of a homomorphism can be found in Figure 1. The arrows from graph (c) to graph (b) depict a homomorphism witnessing an entailment between the corresponding formulae of the graphs.

As promised in Section 2, we are now going to briefly discuss the differences between the semantic models in [2] and this paper. In [2], $\mathcal{I} \models \alpha = (\phi; \sigma)$ if, and only if, $\mathcal{I} \models \phi$ and $\mathcal{I} \models \sigma$, i.e., the semantics is non-intuitionistic. Informally speaking, in our semantics models can contain more red nodes than actually required. It is not difficult to see that the concept of SL-graphs in normal form can be adopted to non-intuitionistic semantics, in fact most proofs carry over straight forwardly. However, the homomorphism conditions need some adjustments. One basically needs to add extra conditions that ensure that when h is a homomorphism from G to G', all edges from G' are covered by h. These extra conditions ensure that no model of $\alpha(G')$ can contain extra red nodes that, informally speaking, do not get used up by $\alpha(G)$. Moreover, some further adjustments have to be made in order to deal correctly with precise list semantics. The details are messy and deferred to a full version of this paper.

5 Syntactic Extensions Leading to Intractability

As stated in Section 2, due to the non-convexity, it is rather surprising that entailment in our fragment is decidable in polynomial time. In this section, we briefly discuss syntactic extensions that render satisfiability or entailment intractable. It turns out that even small syntactic extensions make computing entailment intractable.

First, we consider additional Boolean connectives in pure and spatial formulae. Allowing disjunction in pure formulae with the standard semantics makes entailment coNP-hard, since implication between Boolean formulae is coNP-hard. Less obviously, allowing conjunction in spatial assertions makes satisfiability NP-hard and thus entailment coNP-hard. To be more precise, suppose we allow assertions of the form $\alpha = (\phi; \sigma_1 \wedge \sigma_2)$, where σ_1 and σ_2 are spatial formulae and $\mathcal{I} \models (\phi; \sigma_1 \wedge \sigma_2)$ if, and only if, $\mathcal{I} \models (\phi; \sigma_1)$ and $\mathcal{I} \models (\phi; \sigma_2)$. We reduce from graph three colourability (3Col), which is known to be NP-complete [8]. Given an instance $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of 3Col with $\mathcal{V} = \{v_1, \ldots, v_n\}$, we construct an assertion α such that $\mathcal{G}$ can be coloured with three colours if, and only if, α is satisfiable. We set $\alpha \overset{\text{def}}{=} (\phi, \sigma)$, where $\phi \overset{\text{def}}{=} \bigwedge_{(v_i,v_j)\in\mathcal{E}} x_i \neq x_j$ and $\sigma \overset{\text{def}}{=} y_1 \mapsto y_2 * y_2 \mapsto y_3 \wedge \bigwedge_{v_i \in \mathcal{V}} \mathsf{ls}(y_1, x_i) * \mathsf{ls}(x_i, y_3)$. Let us sketch the correctness of our reduction. The first conjunct of σ ensures that any model of α contains a list of three nodes that are successively labelled with the variable names y_1, y_2 and y_3. The remaining conjuncts enforce that for any $v_i \in \mathcal{V}$, some y_j-node is additionally labelled with the variable name x_i. Our intention is that y_j is additionally labelled with x_i in a model of α if v_i is coloured with colour j in a three-colouring induced by that model. We use ϕ to enforce that that two labels x_i, x_k are not placed on the node labelled with the same y_j if v_i and v_k are adjacent in $\mathcal{G}$, i.e., they must have a different colour in the induced three colouring. Hence $\mathcal{G}$ can be three coloured if, and only if, α is satisfiable. The coNP-hardness of entailment then follows from the fact that α is satisfiable if, and only if, $\alpha \not\models (x \neq x; \epsilon)$.

Finally, we briefly discuss allowing existentially quantified variables in assertions. An example of such an assertion is $\alpha = \exists y.(x \neq y; x \mapsto y)$, where $\mathcal{I} \models \alpha$ if $\mathcal{I}$ can be extended to $\mathcal{I}'$ in which some node of $\mathcal{I}$ is labelled with y such that $\mathcal{I}' \models (x \neq y; x \mapsto y)$. It is easily seen that satisfiability in this extended fragment is still decidable in polynomial time by just dropping the existential quantifier. However, it follows from recent results that entailment becomes coNP-hard [9].

6 Conclusion

In this paper, we have studied the complexity of entailment in a fragment of separation logic that includes pointers and linked lists. Despite the non-convexity of this logic, we could show that entailment is computable in polynomial time. To this end, we showed that for any SL-formula we can compute in polynomial time a corresponding SL-graph in a particular normal form which has an equivalent corresponding SL-formula. Moreover, we showed that deciding entailment between two SL-formulae then reduces to checking for the existence of a homomorphism between their associated SL-graphs in normal form. A key advantage was that the homomorphism, if it exists, is uniquely determined by the SL-graphs, and that checking the homomorphism conditions can be performed in polynomial time. Moreover, we discussed how minor syntactic extensions to our fragment lead to intractability of the entailment problem.

Acknowledgements. We would like to thank Nikos Gorogiannis for interesting and helpful discussions and the anonymous referees for helpful suggestions on how to improve the presentation of the paper.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Space invading systems code. In: Logic-Based Program Synthesis and Transformation, pp. 1–3. Springer, Heidelberg (2009)
5. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 108–119. Springer, Heidelberg (2001)
6. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tracatable reasoning in a fragment of separation logics (full version). Technical report, University of Oxford (2011), http://www.cs.ox.ac.uk/people/christoph.haase/sl.pdf
7. Distefano, D., Parkinson, M.: jstar: towards practical verification for java. In: OOPSLA 2008, pp. 213–226. ACM, New York (2008)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
9. Gorogiannis, N., Kanovich, M., O'Hearn, P.: The complexity of abduction for separated heap abstraction. In: SAS 2011. Springer, Heidelberg (to appear, 2011)
10. Haase, C., Lutz, C.: Complexity of subsumption in the EL family of description logics: Acyclic and cyclic tboxes. In: ECAI 2008, pp. 25–29. IOS Press, Amsterdam (2008)
11. Ishtiaq, S.S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. In: POPL 2001, pp. 14–26. ACM, New York (2001)
12. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report 520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
13. Miklau, G., Suciu, D.: Containment and equivalence for an XPath fragment. In: PODS 2002, pp. 65–76. ACM, New York (2002)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002. IEEE Computer Society Press, Los Alamitos (2002)

On Locality and the Exchange Law for Concurrent Processes

C.A.R. Hoare[1], Akbar Hussain[2], Bernhard Möller[3], Peter W. O'Hearn[2], Rasmus Lerchedahl Petersen[2], and Georg Struth[4]

[1] Microsoft Research Cambridge
[2] Queen Mary University of London
[3] Universität Augsburg
[4] University of Sheffield

Abstract. This paper studies algebraic models for concurrency, in light of recent work on Concurrent Kleene Algebra and Separation Logic. It establishes a strong connection between the Concurrency and Frame Rules of Separation Logic and a variant of the exchange law of Category Theory. We investigate two standard models: one uses sets of traces, and the other is state-based, using assertions and weakest preconditions. We relate the latter to standard models of the heap as a partial function. We exploit the power of algebra to unify models and classify their variations.

1 Introduction

The theory of concurrency is composed of a bewildering variety of models founded on diverse concepts. The general outlook in this paper is one of unification. Wouldn't it be wonderful if we could find a more general theory that accepted the diverse models as instances? Might it be possible to have a comprehensive treatment of concurrency in which shared memory (in weak or strong forms), message passing (in its numerous forms), interleaving and independence models were seen to be part of the same theory with the same core axioms, specialised in different ways, rather than founded on disparate models or calculi? Ideally, such a unifying theory would even connect up the models, program logics, and operational semantics. Of course, unification has long been an aim in concurrency theory, with many translations between models aiding understanding.

With such lofty general aims, we state right away that this paper is but a modest attempt to contribute to such a programme. We in no way have a grand unifying theory at the moment. But we are probing, and as a result altering, algebraic concepts by comparing them with concrete models, in the hope that evolution to a general theory is eventually possible.

This paper builds on previous work on Concurrent Kleene Algebra (CKA, [8]), where an algebra mixing primitives for concurrent ($*$) and sequential ($;$) composition was introduced, generalizing the Kleene algebra of sequential programs [9]. The standout feature of these algebras is the presence of an ordered version of the exchange law from 2-categories or bi-categories

$$(p * r) ; (q * s) \sqsubseteq (p \, ; q) * (r \, ; s)$$

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 250–264, 2011.

This law makes immediate sense in an interleaving model of concurrency. Suppose you have a trace $t = t_1; t_2$ where t_1 is an interleaving of two traces t_p and t_r, and t_2 of t_q and t_s. Then t is certainly also an interleaving of $t_p; t_q$ and $t_r; t_s$. But, as was shown in [8], the law also validates proof rules which originally arose in a completely different model of concurrency, based on separation of resources, the Concurrency and Frame Rules from Concurrent Separation Logic [10,3],

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}} \qquad \frac{\{P\}\, C\, \{Q\}}{\{P * F\}\, C\, \{Q * F\}}.$$

These rules are derived at once from the algebraic structure of CKA, meaning that the modular reasoning that these rules support could conceivably be applicable in a wide variety of circumstances.

This is a remarkable situation. The concurrency and Frame Rules in Concurrent Separation Logic (CSL) have historically relied on somewhat subtle locality properties of the semantics of programs [11,4], which say that programs access resources in a circumscribed fashion. But these laws flow extremely easily in a CKA, and examples of CKAs have been described which do not have anything that looks like locality conditions on resource access. Furthermore, in [8] the validity of Hoare logic laws is referred to, slightly tongue in cheek, as due to a 'cheat', of identifying assertions with programs. Our starting point was simply to understand: what is going on here? Is the easy validity of these laws in CKA a sleight of hand? Or, do they have the same import as in CSL (but generalized)?

We consider another concrete model, the Resource Model, based on the primitive of resource separation used in CSL. This model is built from predicate transformers, and is equipped with notions of parallel and sequential composition. It is directly a model of Concurrent Separation Logic. We show that two algebraic interpretations of Hoare triples coincide, in the model, with the usual interpretation of pre/post specs for predicate transformers. This shows that there is no cheating or sleight of hand: the various logical laws of Hoare and Separation Logic 'mean the same thing' when the algebra is instantiated to this standard and independently existing model of Separation Logic.

An interesting point is that we find some, but not all, of the structure of a CKA in the Resource Model. This leads us to propose a notion of 'locality bimonoid', which is derived from algebraic considerations as well as computational considerations concerning concurrency and resources. The basic structure is that of a pair of ordered monoids on the same underlying poset, linked by the exchange law. Locality is expressed using the equation $f = f * \mathsf{skip}$, where skip is the unit of the sequencing operator ;. We show that this provides a simpler and much more general alternate characterization of a semantic definition of locality used in work on Separation Logic. We end up with a situation where there is an algebra satisfying exchange but not locality, and this equation serves as a healthiness condition (in the sense of Dijkstra [5] and He/Hoare [7]) used to distinguish a subalgebra of local elements, which we call the *local core*. We also find, in contrast with our initial expectation, that the exchange law does not itself rely on locality: the non-local algebra admits exchange, and as a consequence it validates the Concurrency but not the Frame Rule of CSL.

In addition to the aforementioned works on CKA and CSL, we mention other prior related work on the exchange law in concurrency. The structure of two monoids with shared unit and exchange has been called 'concurrent monoid' in [8]. This structure was studied earlier by Gischer in the 1980s [6], and then by Bloom and Ésik [2] in the 1990s. Gischer showed that pomsets or series parallel posets are models of concurrent monoids, and that series parallel posets form the free ordered algebras in that variety. Bloom and Ésik showed that languages with shuffle form models of concurrent monoids. The relation of this paper to these works is similar to the relation to the CKA work: we introduce a more general structure, locality bimonoid, which has a concurrent monoid as the local core, and we connect this structure to a standard model of pre/post specs, complementing the models in [6,2,8] based on traces.

All calculational proofs in this paper have been formally verified by automated theorem proving in Isabelle; they can be found online [1].

2 Two Models

Before moving to algebra, we describe two concrete models that we will refer to throughout the paper.

The Trace Model. Let A be a set. Then *Traces* is the set of finite sequences over A. The powerset $P(\mathit{Traces})$ will be the carrier set of this algebra. We have the following two binary operations on $P(\mathit{Traces})$

1. $T_1 * T_2$ is the set of interleavings of traces in T_1 with traces in T_2;
2. $T_1 \,; T_2$ is the set of concatenations of traces in T_1 with traces in T_2.

The set $\{\epsilon\}$ functions as a unit for both ; and $*$, where ϵ is the empty sequence. The exchange law in this model was already stated in [2]. $P(\mathit{Traces})$ is a prototypical CKA and even a quantale [8]. This means, among other things, that it is a complete lattice in which $*$ and ; distribute through $\bigsqcup$, and that both $*$ and ; have a left and right zero (the empty set, $\emptyset$).

The Resource Model. Let $(\Sigma, \bullet, u)$ be a partial, commutative monoid, given by a partial binary operation $\bullet$ and unit u where the unity, commutativity and associativity laws hold. Here equality means that both sides are defined and equal, or both are undefined. In examples we will often refer to the partial monoid of heaps, where Σ is the set of finite partial functions from naturals to naturals, $\Sigma = N \rightharpoonup_f N$, with $\bullet$ being the partial operation of union of functions with disjoint domains and u being the empty function. The domain of a heap h is denoted by $\mathit{dom}(h)$.

The powerset $P(\Sigma)$ has an ordered total commutative monoid structure $(*, \mathit{emp})$ given by

$$\begin{aligned} p * q &= \{\sigma_0 \bullet \sigma_1 \mid \sigma_0 \# \sigma_1 \wedge \sigma_0 \in p \wedge \sigma_1 \in q\} \\ \mathit{emp} &= \{u\} \end{aligned}$$

where $\sigma_0 \# \sigma_1$ means that $\sigma_0 \bullet \sigma_1$ is defined.

The monotone function space $[P(\Sigma) \rightarrow P(\Sigma)]$ will be the carrier set of the algebra in the Resource Model. These functions represent (backwards) predicate transformers and the algebra has the following operators. Using F_i to range over predicate transformers, and Y, Y_i... to range over subsets of Σ, we define

$$\begin{aligned}(F_1 * F_2)Y &= \bigcup\{F_1Y_1 * F_2Y_2 \mid Y_1 * Y_2 \subseteq Y\}\\ \mathsf{nothing}\ Y &= Y \cap emp\\ (F_1\,;F_2)Y &= F_1(F_2(Y))\\ \mathsf{skip}\ Y &= Y\end{aligned}$$

Here we are overloading $*$, relying on context to disambiguate: sometimes, as on the right of the definition of $F_1 * F_2$, it refers to an operator on predicates, and other times, as on the left of the definition, to an operation on predicate transformers. The definition of the predicate transformer $F_1 * F_2$ was suggested to O'Hearn by Hongseok Yang in 2002, shortly after CSL was introduced (and before it was published). It is motivated by the concurrency proof rule of CSL. The idea is that we start with a postcondition Y, split it into separate assertions Y_1 and Y_2, and apply the Concurrency Rule backwards to get a precondition $F_1Y_1 * F_2Y_2$ for the parallel composition of F_1 and F_2. We union over all such, to obtain the weakest possible precondition.

We use the reverse pointwise ordering $\sqsubseteq$ on $[P(\Sigma) \rightarrow P(\Sigma)]$: $F \sqsubseteq G$ iff $\forall X.\, FX \supseteq GX$. According to this definition, the least element is the function $\lambda X.\Sigma$, corresponding to the weakest (liberal) precondition transformer for divergence. We are thinking of preconditions for partial rather than total correctness here, so we do not insist on Dijkstra's 'law of the excluded miracle' $F\emptyset = \emptyset$.

Note that, in contrast to the Trace Model, the Resource Model has distinct units: `nothing` is the unit of $*$ and `skip` that of ;.

3 Exchange

In this section we introduce the main definitions surrounding the exchange law, we make the connection to program logic, and we classify our two running examples in terms of the introduced notions.

3.1 Algebra

Definition 3.1 (Ordered bisemigroup). An *ordered bisemigroup* $(M, \sqsubseteq, *, ;)$ is a partially ordered set $(M, \sqsubseteq)$ with two monotone compositions $*$ and ;, such that $(M, *)$ is a commutative semigroup and $(M, ;)$ is a semigroup.

Definition 3.2 (Exchange Laws). Over the signature of an ordered bisemigroup $(M, \sqsubseteq, *, ;)$, we consider the following formulas for $p, q, r, s \in M$:

$$\begin{aligned}(p * r)\,;(q * s) &\sqsubseteq (p; q) * (r; s) && \text{(full) exchange}\\ (r * p); q &\sqsubseteq r * (p; q) && \text{small exchange I}\\ p; (q * r) &\sqsubseteq (p; q) * r && \text{small exchange II}\end{aligned}$$

As mentioned in Section 1, it is easy to see that the exchange law holds in the Trace Model. The situation in the Resource Model is subtler, and it will be helpful to exemplify the full law and give counterexamples to the small versions.

Example 3.3 (Exchange in Heap Model). We consider a specialization of the Resource Model, based on heaps. With this model we will use commands

$$[n] := m \quad = \quad \lambda X.\ \{h[n \mapsto l] \mid h \in X, l \in \mathbb{N}, n \in dom(h), h(n) = m\}$$

where $h[n \mapsto l]$ is function update. The command $[n] := m$ updates the memory at location n to have the value m.

With this definition, we can give this example of the exchange law.

$$\begin{array}{cc} ([10] := 55 * [20] := 66)\,;([20] := 77 * [10] := 88) & (\neq \top) \\ \sqsubseteq & \\ ([10] := 55\,;[20] := 77) * ([20] := 66\,;[10] := 88) & (= \top) \end{array}$$

The reason that this holds is that, as there is a race on locations 10 and 20, the lower term is $\top$. The reader might enjoy verifying this in the model. On the other hand, the upper term is not $\top$ since, given a state whose domain contains both 10 and 20, we can split it for the first parallel composition, and again for the second: thus, there is a non-empty precondition for the upper program.

Note that this example also shows that the reverse direction of the exchange law is not valid in the model. ■

Example 3.4 (Invalidity of Small Exchange in Heap Model). To obtain a counterexample to the law $p\,;(q * r) \sqsubseteq (p\,;q) * r$, we choose $p = q = \mathsf{nothing}$ and $r = \mathsf{skip}$. The left hand side is then $\mathsf{nothing}\,;(\mathsf{nothing} * \mathsf{skip}) = \mathsf{nothing}$.

We now observe that in the Resource Model, $\mathsf{nothing}\,;\mathsf{nothing} = \mathsf{nothing}$, so the right hand side simplifies to skip. And clearly it is not the case that $\mathsf{nothing} \sqsubseteq \mathsf{skip}$ in the Resource Model.

The same instantiation yields a counterexample to $(r * p)\,; q \sqsubseteq r * (p\,; q)$. ■

Proposition 3.5. *The Resource Model satisfies the exchange law.*

Proof. Let F_1, F_2, G_1 and G_2 be (monotone) predicate transformers and $X \subseteq \Sigma$. Since the order on predicate transformers is pointwise reverse inclusion, we aim to show

$$((F_1\,; F_2) * (G_1\,; G_2))X \subseteq ((F_1 * G_1)\,;(F_2 * G_2))X$$

The left hand side expands to

$$\bigcup\{F_1(F_2X_F) * G_1(G_2X_G) \mid X_F * X_G \subseteq X\}$$

Now, for every subsplitting $X_F * X_G$ of X, we see that $F_2X_F * G_2X_G$ is a subsplitting of $(F_2 * G_2)X = \bigcup\{F_2A * G_2B \mid A * B \subseteq X\}$, simply because it is one of the elements of the union. So we can overapproximate the left hand side:

$$\bigcup\{F_1(F_2X_F)*G_1(G_2X_G) \mid X_F*X_G \subseteq X\} \subseteq \bigcup\{F_1A*G_1B \mid A*B \subseteq (F_2*G_2)X\}$$

and this overapproximation quickly rewrites to the right hand side:

$$\bigcup\{F_1A{*}G_1B \mid A{*}B \subseteq (F_2{*}G_2)X\} = (F_1{*}G_1)((F_2{*}G_2)X) = (F_1{*}G_1);(F_2{*}G_2)X$$

□

It is a straightforward extension to consider units.

Definition 3.6 (Ordered bimonoid). An *ordered bimonoid* is a structure $(M, \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ such that $(M, \sqsubseteq, *, ;)$ is an ordered bisemigroup and $(M, *, \mathsf{nothing})$ and $(M, ;, \mathsf{skip})$ are monoids.

A first important consequence is

Lemma 3.7. *In an ordered bimonoid with exchange,* $\mathsf{skip} \sqsubseteq \mathsf{nothing}$ *and* $\mathsf{nothing} \sqsubseteq \mathsf{nothing}\,;\mathsf{nothing}$.

It is tempting to require the two units to be identical, but in the Resource Model they are not. Still, when the units are equal we are in a special situation.

Lemma 3.8. *In an ordered bimonoid* $(M, \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ *with shared unit (meaning* $\mathsf{nothing} = \mathsf{skip}$*), the full exchange law implies the small exchange laws.*

3.2 Program Logic

In our algebra we may interpret pre/post specs in two ways.

Definition 3.9. Let $(M, \sqsubseteq, ;)$ be an ordered semigroup and $p, c, q \in M$.

1. The *Hoare triple* $\{p\}\,c\,\{q\}$ is defined by $\{p\}\,c\,\{q\} \;\Leftrightarrow\; p;c \sqsubseteq q$
2. The *Plotkin triple* $p \rightarrow_c q$ is defined by $p \rightarrow_c q \;\Leftrightarrow\; p \sqsupseteq c;q$

The Hoare triple can be thought of in terms of the past. In the Trace Model, it says that if p describes events done before command c is run, then q over-approximates p followed by c. Conversely, the Plotkin triple can be thought of in terms of the future: it says that if p describes a possible future, and q describes the future after c is run, then p over-approximates c followed by q.

In the Resource Model, there is a further interpretation of pre/post specs:

Definition 3.10 (Dijkstra Triple). In the Resource Model, if X is a predicate (a set of states) and F is a predicate transformer, then the relationship $X \subseteq FY$ says that X is a valid precondition for F with post Y:

$$\langle\!\langle X\rangle\!\rangle\, F\, \langle\!\langle Y\rangle\!\rangle \quad\Leftrightarrow\quad X \subseteq FY$$

While this definition can be given for arbitrary predicate transformers, the rule of consequence for F, that $X \subseteq X' \wedge \langle\!\langle X'\rangle\!\rangle\, F\, \langle\!\langle Y'\rangle\!\rangle \wedge Y' \subseteq Y \Rightarrow \langle\!\langle X\rangle\!\rangle\, F\, \langle\!\langle Y\rangle\!\rangle$, holds iff F is monotone.

While the interpretations in Definition 3.9 make sense on their own, it is useful to try to connect them to the ordinary understanding of specs in state-based (rather than history-based) models; we will do that in Section 5.

We now leave the concrete model and the Dijkstra triples, and describe the program logic that results from Definition 3.9.

Lemma 3.11 (Hoare Logic). *In an ordered semigroup the following rules hold:*
Rule of Consequence:

$$\forall p, c, q, p', q'.\ \{p'\}\, c\, \{q'\} \ \wedge\ p \sqsubseteq p' \wedge q' \sqsubseteq q \quad \Rightarrow \quad \{p\}\, c\, \{q\}$$

Sequencing Rule:

$$\forall p, c, r, c', q.\ \{p\}\, c\, \{r\} \ \wedge\ \{r\}\, c'\, \{q\} \quad \Rightarrow \quad \{p\}\, c; c'\, \{q\}$$

If the ordered semigroup is an ordered monoid, then the following holds:

Skip Rule:

$$\forall p.\ \{p\}\, \mathsf{skip}\, \{p\}$$

Furthermore, this lemma also holds with Plotkin in place of Hoare triples, except that the ordering is reversed in the Rule of Consequence.

In a bisemigroup we can obtain proof rules from Separation Logic.

Theorem 3.12 (Separation Logic). *In an ordered bisemigroup,*

1. *The exchange law holds iff we have the*
   Concurrency Rule:
   $$\forall p, c, q, p', c', q'.\ \{p\}\, c\, \{q\} \ \wedge\ \{p'\}\, c'\, \{q'\} \quad \Rightarrow \quad \{p * p'\}\, c * c'\, \{q * q'\}$$
   Furthermore, the same holds true with Plotkin in place of Hoare triples.
2. *The first small exchange law holds iff we have*
   Frame Rule (for Hoare triples):
   $$\forall p, c, q, r.\ \{p\}\, c\, \{q\} \Rightarrow \{p * r\}\, c\, \{q * r\}$$
3. *The second small exchange law holds iff we have*
   Frame Rule (for Plotkin triples):
   $$\forall p, c, q, r.\ p \rightarrow_c q \Rightarrow p * r \rightarrow_c q * r$$

This abstract treatment does not cover all of the rules of Hoare or Separation Logic, notably the conjunction rule and rules for variable renaming and quantifiers, as there is not enough structure to state them.

It is worth remarking that we also have temporal frame rules.

Futuristic Frame Rule (for Plotkin Triples):

$$\forall p, c, q, r.\ p \rightarrow_c q \Rightarrow p; r \rightarrow_c q; r$$

Archaic Frame Rule (for Hoare Triples):

$$\forall p, c, q, r.\ \{p\}\, c\, \{q\} \Rightarrow \{r; p\}\, c\, \{r; q\}$$

We summarize the situation with our two example models as follows.

Proposition 3.13 (Classification).

1. *The Trace Model* $(P(\mathit{Traces}), \sqsubseteq, *, \{\epsilon\}, ;, \{\epsilon\})$ *is an ordered bimonoid with shared unit satisfying the exchange law. Consequently, it satisfies all of the Hoare and Separation Logic rules mentioned in this section.*
2. *The Resource Model* $([P(\Sigma) \rightarrow P(\Sigma)], \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ *is an ordered bimonoid with distinct units, satisfying the full but not small exchange laws. Consequently, it satisfies the rules of Hoare Logic and the Concurrency Rule of Separation Logic, but not the Frame Rule (for Hoare or Plotkin triples).*

The Trace Model is an example of a CKA, defined in [8]. The Resource Model, however, is not. Having two different units causes a difference. Additionally, the model does not have a right zero for ;, an element 0 where $p;0 = 0$ for all p (while all constant transformers are left zeros). The Trace Model does have a right zero, the empty set.

The properties of the Resource Model seem a bit odd at first sight. We have a situation where the Concurrency Rule (which expresses a form of modular reasoning) is present while the Frame Rule (which expresses locality) is absent. We now probe this issue further.

4 Locality

In this section we develop the concept of locality, culminating in an analysis of what we call a 'locality bimonoid' (an ordered bimonoid with exchange where skip is idempotent for $*$, Definition 4.7). In a locality bimonoid we have the Frame Rules exactly for the local elements, and we also have that the local elements form a locality sub-bimonoid where the inclusion is a Galois insertion. Some of the results in this section do not require all of the data of a locality bimonoid, so we build up to the definition as we go along.

4.1 Local Elements

In the Resource Model, a transformer F is called local if it can be described in terms of its action on parts of the state. This can be expressed as follows:

$$FP = \bigcup\{(FX) * R \mid X * R = P\} \tag{1}$$

We have already seen a non-local transformer, $\mathsf{nothing} = \lambda Y.\, Y \cap emp$, and it will be helpful to see why this contradicts the frame rule. Note that $emp \subseteq \mathsf{nothing}\; emp$, but that (in the heap model) $\{1 \mapsto 2\} \not\subseteq \mathsf{nothing}\{1 \mapsto 2\}$ where $1 \mapsto 2$ is a singleton heap. Thus, as $\{1 \mapsto 2\} * emp = \{1 \mapsto 2\}$, we have $(emp * \{1 \mapsto 2\}) \not\subseteq \mathsf{nothing}(emp * \{1 \mapsto 2\})$. So the usual semantics of pre/post specs for predicate transformers does not validate the frame rule for $\mathsf{nothing}$.

The shape of the right-hand side of equation (1) suggests a connection between the transformers F and $F * \mathsf{skip}$. First, choosing $X = P$ and $R = emp$ we get, for arbitrary F,

$$FP \subseteq \bigcup\{(FX) * R \mid X * R = P\} \subseteq \bigcup\{(FX) * R \mid X * R \subseteq P\} = (F * \mathsf{skip})P$$

i.e., $F * \mathsf{skip} \sqsubseteq F$. We will see that this is a consequence of the exchange law.

So the nontrivial part of (1) is $\bigcup\{(FX) * R \mid X * R = P\} \subseteq FP$. Setting $P = X * R$ for arbitrary X, R, this implies $FX * R \subseteq F(X * R)$ and hence

$$(F * \mathsf{skip})P = \bigcup\{(FX) * R \mid X * R \subseteq P\} \subseteq \bigcup\{F(X * R) \mid X * R \subseteq P\} = FP$$

since F is monotone. This means $F \sqsubseteq F * \mathsf{skip}$. Conversely, if $F \sqsubseteq F * \mathsf{skip}$ then

$$FP \supseteq \bigcup\{(FX) * R \mid X * R \subseteq P\} \supseteq \bigcup\{(FX) * R \mid X * R = P\}$$

These results can be summarised by

Lemma 4.1. *A predicate transformer F satisfies (1) iff $F = F * \mathsf{skip}$.*

We now consider locality on the algebraic level. First, Lemma 3.7 implies that in an ordered bimonoid with exchange, all elements satisfy $p * \mathsf{skip} \sqsubseteq p$, so in this setting the definition below simply states the other direction.

Definition 4.2. An element p in an ordered bisemigroup is *local* if $p * \mathsf{skip} = p$.

Example 4.3 (Local skip). In the Resource Model, skip is local. ■

The idea of locality is to admit local reasoning, and indeed that can be verified on the level of algebra:

Lemma 4.4. *In an ordered bimonoid with exchange, the Frame rules for Plotkin and Hoare triples hold when the 'command' c is local. That is, for $c = c * \mathsf{skip}$*

$$\forall p, q, r.\ \{p\}\, c\, \{q\} \Rightarrow \{p * r\}\, c\, \{q * r\}$$

$$\forall p, q, r.\ p \to_c q \Rightarrow p * r \to_c q * r$$

The local elements form a subalgebra:

Lemma 4.5. *In an ordered bimonoid, the local elements are closed under $*$.*

Lemma 4.6. *In an ordered bimonoid with exchange, the local elements are closed under ;.*

The units nothing and skip may not be local, though [1], so one cannot immediately form a sub-bimonoid. We now consider the special case of an ordered bimonoid where skip is local. This enables us to localize elements and thus recover a sub-bimonoid.

Definition 4.7 (Locality Bimonoid). A *locality bimonoid* is an ordered bimonoid with exchange where skip is local, i.e., $\mathsf{skip} * \mathsf{skip} = \mathsf{skip}$.

Both our running examples, the Trace Model and the Resource Model, are examples of locality bimonoids. Proposition 3.13 and Example 4.3 states this.

Lemma 4.8. *Let G be a locality bimonoid and $p \in G$. Then $p * \mathsf{skip}$ is local.*

Another way to say this is that the *localizer* $(-) * \mathsf{skip}$ is idempotent. This allows us to define a sub-bimonoid:

Definition 4.9 (Local Core). Let $G = (M, \sqsubseteq, *, \mathsf{nothing}, ;, \mathsf{skip})$ be a locality bimonoid. Then we define the *local core* G_{loc} of G by

$$G_{loc} = (M_{loc}, \sqsubseteq, *, \mathsf{skip}, ;, \mathsf{skip})$$

where $M_{loc} = \{p \in M \mid p = p * \mathsf{skip}\}$, and $*$ and ; are restricted appropriately.

Since the localizer is idempotent, selecting the local elements is the same as selecting the image of the localizer.

Lemma 4.10. *If G is a locality bimonoid, then so is G_{loc}.*

Thus, the notion of Local Core is well defined, but we can say more. It is an instance of the special situation of when all elements are local, which gives us a structure called a *concurrent monoid* in [8] (see Definition 4.12 below).

Lemma 4.11. *In a bimonoid, the following are equivalent*

(i) $\mathsf{nothing}$ *is local*
(ii) *All elements are local*
(iii) skip *is a unit of* $*$
(iv) $\mathsf{nothing} = \mathsf{skip}$

Definition 4.12 (Concurrent Monoid, [8]). A *concurrent monoid* is an ordered bimonoid with exchange where $\mathsf{nothing} = \mathsf{skip}$.

Since all elements of a concurrent monoid are local, skip is as well, and we immediately have that a concurrent monoid is a locality bimonoid.

Proposition 3.13 and Example 4.3 states that the Trace Model is a concurrent monoid, while the Resource Model is a locality bimonoid which is not a concurrent monoid.

From Lemma 3.8 we can infer that the full and small exchange laws are valid in a concurrent monoid.

Corollary 4.13 (to Lemma 4.11). *Let G be a locality bimonoid. Then G_{loc} is a concurrent monoid.*

There is a simple connection between G and G_{loc}, given by the localizing map $(-) * \mathsf{skip}$. We can even phrase it without needing skip to be local:

Lemma 4.14. *In an ordered bimonoid with exchange, $p * \mathsf{skip}$ is the greatest local element smaller than p.*

When we have a locality bimonoid, the same reasoning gives us a Galois connection between the bimonoid and its core.

Lemma 4.15. *Let G be a locality bimonoid. There is a Galois connection between G and G_{loc}, where localization $(-) * \mathsf{skip} : G \to G_{loc}$ is right adjoint to the inclusion from G_{loc} into G.*

The final result explains the name locality bimonoid. It states that locality coincides with local reasoning in the form of the Frame rule.

Theorem 4.16. *In a locality bimonoid, the Frame rules for Plotkin and Hoare triples hold exactly when the 'command' c is local. That is, $c = c * \mathsf{skip}$ iff*

$$\begin{array}{rl} & \forall p, q, r.\ \{p\}\, c\, \{q\} \Rightarrow \{p * r\}\, c\, \{q * r\} \\ \textit{iff} & \forall p, q, r.\ p \to_c q \Rightarrow p * r \to_c q * r \end{array}$$

In terms of algebraic models of concurrency, this suggests to us that local elements should be considered 'programs', while in general a bimonoid might contain non-program-like elements, such as those that play the role of assertions. This is a crucial difference between locality bimonoids and Concurrent Kleene Algebras. There seems to be no compelling reason for requiring locality of (the denotations of) assertions, and a locality bimonoid does not require them to be. But, we will see in Section 5 that neither does requiring locality of all elements lose us anything in characterizing pre/post specs in our state-based model.

4.2 Lubs and Fixed-points

The notion of locality bimonoid is not, on its own, rich enough to interpret programs. There are many further constructs one could consider beyond sequencing and parallelism, but perhaps the most important is recursion or iteration; without it, we don't even have Turing-completeness. This subsection does some basic sanity checking in this direction, and can be skipped or skimmed on a first reading, without loss of continuity: it shows that our motivating Resource Model admits a standard treatment of recursion.

Theorem 4.17. *If the order of a locality bimonoid is a complete lattice, then*

1. *The local core is a complete lattice as well.*
2. *Any expression e built from local elements using $*$, ; and $\bigsqcup$ denotes a local element.*
3. *Any expression $e(X)$ built from local elements using $*$, ; and $\bigsqcup$ and one unknown X denotes a monotone function on the local core; it thus has a least (local) fixed-point $\mu X.\, e(X)$ which is again in the local core.*

Part *1* of this theorem is immediate from the Tarski-Knaster fixed-point theorem because the local elements are the fixed-points of the monotonic function $(-) * \mathsf{skip}$. Parts *2* and *3* are consequences of *1*.

This gives us enough information to interpret a programming language with sequencing, parallelism, nondeterminism and recursion. The form of recursion is sufficient to encode iterators for both sequential and parallel composition, as $\mu X.\, F; X$ and $\mu X.\, F * X$. We will connect this to program logic in Section 5.

Finally, although we have noted that the lub of local transformers is local, curiously, we do *not* have that $(-) * \mathsf{skip}$ distributes through $\bigsqcup$.

Example 4.18 (Localization does not preserve binary $\bigsqcup$). We first define a number of heaps:

$$h = \{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}$$
$$h_{12} = \{1 \mapsto 1, 2 \mapsto 2\} \qquad h_{23} = \{2 \mapsto 2, 3 \mapsto 3\}$$
$$h_1 = \{1 \mapsto 1\} \qquad\qquad h_2 = \{2 \mapsto 2\} \qquad\qquad h_3 = \{3 \mapsto 3\}$$

This allows us to define the following (constant) transformers:

$$F = \lambda P.\ \{h_1\} \qquad G = \lambda P.\ \{h_3\}$$

And now we obtain a counterexample to distribution through lubs as

$$((F \sqcup G) * \mathsf{skip})\{h_{12}, h_{23}\} \neq ((F * \mathsf{skip}) \sqcup (G * \mathsf{skip}))\{h_{12}, h_{23}\}$$

For the left hand side, we observe that $F \sqcup G = \lambda P.\ \{h_1\} \cap \{h_3\} = \lambda P.\ \emptyset$ and so $(F \sqcup G) * \mathsf{skip} = \lambda P.\ \emptyset$.

For the right hand side, we observe that $\{h\} = \{h_1\} * \{h_{23}\} = (F\ emp) * \{h_{23}\}$ and that $emp * \{h_{23}\} \subseteq \{h_{12}, h_{23}\}$, so $\{h\} \subseteq (F * \mathsf{skip})\{h_{12}, h_{23}\}$. Similarly, $\{h\} = \{h_3\} * \{h_{12}\} = (G\ emp) * \{h_{12}\}$ and $emp * \{h_{12}\} \subseteq \{h_{12}, h_{23}\}$ so $\{h\} \subseteq (G * \mathsf{skip})\{h_{12}, h_{23}\}$. Thus the right hand side is not empty. ■

Being a right adjoint, however, $(-) * \mathsf{skip}$ does distribute through $\bigsqcap$.

As a consequence of this result, we do *not* have that our locality bimonoid of predicate transformers is a quantale: a quantale requires that $p * (\cdot)$ preserves all lubs, for arbitrary p.

There are two ways to interpret this fact. The first reaction is that we expect distribution to hold; that the Resource Model is somehow defective. It is worth remarking that the Trace Model does satisfy distribution of $p * (\cdot)$ through lubs. The second reaction is that there is nothing in program logic (say, CSL) which *forces* this law of distribution of $p * (\cdot)$ through lubs, and thus we needn't insist upon it. We add that neither does sequencing (;) in the Resource Model preserve all lubs (in its second argument). Sequential composition of predicate transformers is so basic that it is hard to argue that one should *demand* full distribution laws (useful though they are when present), as that would rule out a host of natural models.

5 On Assertions, Triples and Programs

At the beginning of the paper we asked whether program logic rules 'meant the same thing' in the algebra models as in familiar state-based models of pre/post specs. This section answers that question in the affirmative, and also provides further perspective on whether we should insist that assertions satisfy locality.

5.1 Connections in the Resource Model

We first show how to connect the Dijkstra triple to the Hoare and Plotkin triples in the Resource Model. We remind the reader of the definitions:

$$\begin{array}{lrcl} \textbf{Hoare triple} & \{p\}\, c\, \{q\} & \Leftrightarrow & p; c \sqsubseteq q \\ \textbf{Plotkin triple} & p \rightharpoonup_c q & \Leftrightarrow & p \sqsupseteq c; q \\ \textbf{Dijkstra triple} & \langle\!\langle X \rangle\!\rangle\, F\, \langle\!\langle Y \rangle\!\rangle & \Leftrightarrow & X \subseteq FY \end{array}$$

Comparing these requires that we can turn the predicates X and Y in the relationship $X \subseteq FY$ defining the Dijkstra triple into predicate transformers, as the pre and post of Hoare and Plotkin triples are the same kinds of entities as the programs. We achieve this via $\mathit{bpt}[X, Y]$, the 'best predicate transformer' corresponding to precondition X and postcondition Y:

$$\mathit{bpt}[X, Y] \;=\; \lambda P.\, \text{if } Y \subseteq P \text{ then } X \text{ else } \emptyset.$$

It is easily seen that $\mathit{bpt}[X, Y]$ is the largest monotone transformer F such that $\langle\!\langle X \rangle\!\rangle\, F\, \langle\!\langle Y \rangle\!\rangle$. Using this definition, for a given X we can define $\mathit{before}[X]$ as

$$\mathit{before}[X] \;=\; \mathit{bpt}[\mathit{true}, X]$$

where true is the predicate Σ. We think of $\mathit{before}[X]$ as covering a 'past' in which X eventually becomes true, and this lets us characterize Dijkstra triples in terms of Hoare triples: It is not difficult to see that, in the Resource Model,

$$\{\mathit{before}[X]\}\, F\, \{\mathit{before}[Y]\} \quad \text{iff} \quad \langle\!\langle X \rangle\!\rangle\, F\, \langle\!\langle Y \rangle\!\rangle$$

In this sense, the algebraic Hoare triple subsumes the traditional state-based one. In fact, $\mathit{before}[]$ is not the only choice; the next result summarizes a few:

Proposition 5.1. *In the Resource Model*

$$\begin{aligned}\langle\!\langle X\rangle\!\rangle\, F\, \langle\!\langle Y\rangle\!\rangle &\Leftrightarrow \{\mathit{bpt}[\mathit{emp}, X]\}\, F\, \{\mathit{bpt}[\mathit{emp}, Y]\}\\ &\Leftrightarrow \{\mathit{bpt}[\mathit{true}, X]\}\, F\, \{\mathit{bpt}[\mathit{true}, Y]\}\\ &\Leftrightarrow \mathit{bpt}[X, \mathit{emp}] \rightarrow_F \mathit{bpt}[Y, \mathit{emp}]\\ &\Leftrightarrow \mathit{bpt}[X, \mathit{true}] \rightarrow_F \mathit{bpt}[Y, \mathit{true}]\end{aligned}$$

Here, $\mathit{bpt}[Y, \mathit{emp}]$ and $\mathit{bpt}[Y, \mathit{true}]$, characterizing the Plotkin triple, talk about the future rather than the past. For instance, we can think of $\mathit{bpt}[Y, \mathit{emp}]$, the largest transformer satisfying $\langle\!\langle Y\rangle\!\rangle - \langle\!\langle \mathit{emp}\rangle\!\rangle$, as an operation that cleans up memory, returning it to the operating system, after checking that Y holds.

It would be ideal if these characterizations were obtained using embeddings of the monoid $(P(\Sigma), *, \mathit{emp})$ of predicates into the locality bimonoid, and this is possible to achieve by choosing embeddings carefully. We observe that

$$\mathit{bpt}[X, Y] * \mathit{bpt}[X', Y'] = \mathit{bpt}[X * X', Y * Y']$$

So fixing one variable to an idempotent (such as *emp* or *true*) yields that the embedding of predicates into predicate transformers preserves $*$. If the idempotent is chosen to be *emp*, then the characterization of *emp* is $\mathit{bpt}[\mathit{emp}, \mathit{emp}] = \mathsf{nothing}$, and so in this case, the characterization preserves the unit as well.

Conceptually, there is no reason to demand that assertions are local in the same way that programs arguably should be. But as it turns out, in the Resource Model, the question of whether one faithfully recovers program logic in the algebra is independent of the question of whether all the elements are required to be local, i.e. one only considers local transformers.

The best predicate transformers are not necessarily local. An example is that $\mathit{bpt}[\mathit{emp}, \mathit{emp}] = \mathsf{nothing}$. However, we can use localization to define the best local transformer $\mathit{blt}[X, Y] = \mathit{bpt}[X, Y] * \mathsf{skip}$ which also happens to characterize triples in the sense that for F local

$$\begin{aligned}\langle\!\langle X\rangle\!\rangle\, F\, \langle\!\langle Y\rangle\!\rangle &\Leftrightarrow \{\mathit{blt}[\mathit{emp}, X]\}\, F\, \{\mathit{blt}[\mathit{emp}, Y]\}\\ &\Leftrightarrow \mathit{blt}[X, \mathit{emp}] \rightarrow_F \mathit{blt}[Y, \mathit{emp}]\end{aligned}$$

5.2 Recursion Rule

We now give a sufficient condition for modelling the Recursion Rule in locality bimonoids. The best transformer $\mathit{blt}[X, Y]$ is an example of a *greatest satisfier* for a predicate, in this case the predicate $\langle\!\langle X\rangle\!\rangle - \langle\!\langle Y\rangle\!\rangle$. This idea is used in our sufficient condition:

Proposition 5.2 (Recursion Rule). *If the order of a locality bimonoid G is a complete lattice, and if greatest local satisfiers for triples exist, then we have the usual Hoare logic proof rule for recursive (parameterless) procedures.*

Recursion Rule: *For all* $F, H \in [G_{loc} \rightarrow G_{loc}]$

$$\begin{aligned} & \{a\}\, x\, \{b\} \Rightarrow \{a\}\, F\ x\, \{b\}\\ \wedge\quad & \{a\}\, x\, \{b\} \Rightarrow \{p\}\, H\ x\, \{q\}\\ \Longrightarrow\quad & \{p\}\, H(\mu F)\, \{q\}\end{aligned}$$

where $[A \to B]$ *denotes the monotone function space and* μF *is the least fixed point of* F*.*

The Hoare triples can be replaced with any downwards closed predicate for which greatest local satisfiers exist (say, Plotkin or Dijkstra triples).

Proof. Since F is monotone it has a least (local) fixed point μF, given by

$$\mu F = \bigsqcap \{x \mid Fx \sqsubseteq x\}$$

Here x ranges over local elements and so μF is the greatest local lower bound of local prefixed points, besides being a fixed point.

We will argue that $\{a\}\, \mu F\, \{b\}$. Let s be the greatest local satisfier of $\{a\} - \{b\}$. Since s is a local satisfier, we know by assumption that $F(s)$ is a local satisfier. Since s is the greatest local satisfier, we know that $F(s) \sqsubseteq s$. Thus, s is a local prefixed point. This means that $\mu F \sqsubseteq s$, as μF is a lower bound of local prefixed points. We now use that satisfaction is downwards closed to conclude that $\{a\}\, \mu F\, \{b\}$. (This is easily seen as $a\,; \mu F \sqsubseteq a\,; s \sqsubseteq b$.)

Thus, by assumption, $H(\mu F)$ is a local satisfier of $\{p\} - \{q\}$. □

The statement in this lemma is another way of describing the usual Hoare proof rule for recursive parameterless procedures

$$\frac{\{A\}\, X\, \{B\} \vdash \{A\}\, F\, \{B\} \qquad \{A\}\, X\, \{B\} \vdash \{P\}\, H\, \{Q\}}{\vdash \{P\}\, \mathit{letrec}\ X = F\ \mathit{in}\ G\, \{Q\}}$$

where, by the previous results, this rule applies when commands are built using sequencing, composition, nondeterministic choice, and procedure declarations.

In the Resource Model we have greatest local satisfiers for Dijkstra triples, given by $\mathit{blt}[X, Y]$, and satisfaction for Dijkstra triples is downwards closed, so this provides a model for CSL with recursion.

Hoare triples, however, are a bit more general. To obtain greatest local satisfiers for the Hoare triple $\{p\} - \{q\}$, we look to

$$\bigsqcup \{c \mid p\,; c \sqsubseteq q\}$$

But this is not guaranteed to be a satisfier, because ; does not distribute through $\bigsqcup$ in its second argument.

As it turns out, ; *does* distribute through $\bigsqcup$ in its first argument, and so the Resource Model models recursion for Plotkin triples. Seeing as the Resource Model is built on *backwards* predicate transformers, it is perhaps natural to expect the Plotkin triples to be more well-behaved than the Hoare triples. We shall just note here that with of definition of ; suitable for forward predicate transformers, it would distribute through $\bigsqcup$ in its second argument.

The Trace Model, being a quantale, models the Recursion Rule for both Hoare and Plotkin triples.

6 Conclusion

In this paper we have described links between a standard model of Concurrent Separation Logic, the Resource model, and algebraic models inspired by the recent Concurrent Kleene Algebra. By looking at such a concrete, and previously-existing, model we hope to have shown that the notion of locality in the algebra generalizes the understanding obtained from the concrete resource-based semantics of Separation Logic.

The algebraic structure used in this paper admits both state-based and history-based models (the Resource and Traces models), and this exemplifies the unifying nature of the algebra, which allows principles to be stated independently of the syntax in specific models. Precursor works even used 'true concurrency' in addition to interleaving models, giving further evidence of the generality [6,8]. In fact, we ended up with a weaker structure than CKA, and perhaps this paper will also have some input into further developments of algebraic models for concurrency. The most pressing issues going forward include axiomatization of further primitives (e.g., distinguishing internal and external choice), exploring a wider range of concrete models, and determining the practical significance of the generalized program logic in any of the concrete models.

Acknowledgements. We are grateful to the referees for perceptive comments. Hussain, O'Hearn, Petersen and Struth were supported by the EPSRC. O'Hearn acknowledges the support of a Royal Soceity Wolfson Research Merit award and a gift from Microsoft Research.

References

1. http://staffwww.dcs.shef.ac.uk/people/G.Struth/isa/
2. Bloom, S.L., Ésik, Z.: Free shuffle algebras in language varieties. TCS 163(182), 55–98 (1996)
3. Brookes, S.D.: A semantics of concurrent separation logic. TCS 375(1-3), 227–270 (2007); Prelim. version appeared in CONCUR 2004
4. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, pp. 366–378. IEEE Computer Society, Los Alamitos (2007)
5. Dijkstra, E.W.: A discipline of programming. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs (1976)
6. Gischer, J.L.: The equational theory of pomsets. TCS 61, 199–224 (1988)
7. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall series in computer science. Prentice-Hall, Englewood Cliffs (1998)
8. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. Journal of Logic and Algebraic Programming (2011); Prelim. version appeared in CONCUR 2009
9. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. 110(2), 366–390 (1994)
10. O'Hearn, P.W.: Resources, concurrency and local reasoning. TCS 375(1-3), 271–307 (2004); Prelim. version appeared in CONCUR 2004
11. Yang, H., O'Hearn, P.W.: A semantic basis for local reasoning. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 402–416. Springer, Heidelberg (2002)

Typed ψ-calculi

Hans Hüttel[⋆]

Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, Denmark

Abstract. A large variety of process calculi extend the π-calculus with more general notions of messages. Bengtson et al. have shown that many of these π-like calculi can be expressed as so-called ψ-calculi. In this paper, we describe a simple type system for ψ-calculi. The type system satisfies a subject reduction property and a general notion of channel safety. A number of existing systems are shown to be instances of our system, and other, new type systems can also be obtained. We first present a new type system for the calculus of explicit fusions by Wischik and Gardner, then one for the distributed π-calculus of Hennessy and Riely and finally show how existing type systems for secrecy and authenticity in the spi calculus can be represented and shown to be safe.

1 Introduction

Process calculi are formalisms that allow us to describe and reason about parallel and distributed computations. A prominent example is the π-calculus due to Milner et al. [17,23], and one of the techniques for reasoning about properties of processes is that of *type systems*. The first type system, due to Milner [17], dealt with the notion of correct usage of channels, ensuring that only names of the correct type could be communicated.

Since then, a plethora of type systems have been introduced. Pierce and Sangiorgi [19] described a type system that uses subtyping and capability tags to control the use of names as input or output channels. There are also several type systems for variants of the π-calculus. In the spi-calculus [3], type systems have been used to capture properties of cryptographic protocols such as secrecy [1,2] and authenticity [11,9]. The distributed π-calculus, Dπ, by Hennessy and Riely [20] can describe located computations in which subprocesses can migrate between locations, and in this setting type systems have been proposed [20,21] for controlling migration.

The type systems mentioned above are seemingly unrelated and arise in different settings but share certain characteristics. They only classify processes as either well-typed or not, so type judgments for processes P are of the form $E \vdash P$, where E is a type environment that records the types of the free names in P. On the other hand, terms M are given a type T, so that type judgments are of the form $E \vdash M : T$.

Bengtson et al. introduced ψ-calculi [4] as a generalization of the many variants of the π-calculus in which structured message terms appear. A main innovation is the use of a small set of process constructs that generalize those of the π-calculus, and the use of general notions of terms, assertions and conditions. Each of these syntactic categories is then assumed to form a *nominal set* in the sense of Gabbay and Mathijssen [10]. It

[⋆] hans@cs.aau.dk

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 265–279, 2011.

has been shown [4] that both existing calculi such as the π-calculus and the spi-calculus and new variants can be captured as ψ-calculi.

The goal of the present paper is to describe a general framework for type systems for variants of the π-calculus within the above tradition by means of a type system for ψ-calculi. Our type system generalizes the so-called simply typed π-calculus introduced by Sangiorgi and Walker [23]. Within it, we can capture several existing, seemingly quite different type systems, including those mentioned above, and formulate new ones. An important advantage is that we can formulate general soundness results that allow us to conclude the soundness of several such type systems.

There has been other work aimed at giving a general account of type systems for process calculi. Most work has focused on general type systems for the π-calculus, and in much of it processes have types that come equipped with a notion of behaviour. Early work includes that of Honda [12] who introduces so-called typed algebras which can be provided with a notion of reduction semantics. Later, [13], Igarashi and Kobayashi described a general type system for a subset of the polyadic π-calculus. This type system can be instantiated to describe e.g. deadlock freedom and race freedom. The type of a π-process is a term in a process calculus reminiscent of CCS.

Bigraphs [18] provide another general setting for process calculi. Here, Elsborg et al. have proposed a type system[8]; however, there are as yet few actual results that show how this type system can be instantiated.

In [16], Makholm and Wells describe a general process calculus Meta* and a general type system Poly*. Meta* can be instantiated by a concrete set of reduction rules, and the resulting type system will satisfy a subject reduction property. Here, there are also significant differences from our approach. Firstly, the main focus of [16] is that of understanding variants of the calculus of Mobile Ambients [6], not variants of the π-calculus. Secondly, in the type system Poly* the typable entities are processes, not names. Finally, the type system of [16] is not instantiated to any existing type system, and it is not clear how this is to be done, given the importance of typed names in many existing type systems for variants of the π-calculus. In the present paper we develop a framework for π-calculus variants in which this can be achieved.

The rest of our paper is structured as follows. Section 2 gives a short introduction to ψ-calculi. In Section 3, we describe our type system for ψ-calculi. In Section 4 we establish important properties of the type system. In particular, we prove a subject reduction property and introduce a general notion of safety. Finally, in Section 5 we show how a number of existing type systems can be seen as instances of our type system.

2 ψ-calculi

The intention of ψ-calculi [4] is to generalize the common characteristics of variants of the π-calculus that allow for transmission of message terms that may be structured, i.e. that need not be names.

2.1 Syntax

A ψ-calculus has a set of *processes*, ranged over by $P, Q, \ldots$. Processes contain occurrences of *terms*, ranged over by $M, N \ldots$ and both processes and terms can contain *names*. We assume a countably infinite set of names $\mathcal{N}$. The set of names that appear in patterns (defined below) is called the set of *variable names* $\mathcal{N}_V$ and is ranged over by $x, y \ldots$. The set of other names, $\mathcal{N} \setminus \mathcal{N}_V$ is ranged over by $a, b, \ldots, m, n \ldots$. We let $u, v \ldots$ range over $\mathcal{N}$.

The formation rules for processes are the following.

$$\begin{array}{rll}
P ::= & \underline{M}(\lambda \boldsymbol{x})N.P & \text{input} \\
& \overline{M}N.P & \text{output} \\
& P_1 \mid P_2 & \text{parallel composition} \\
& (\nu n : T)P & \text{restriction of name } n \\
& !P & \text{replication} \\
& \textbf{case } \varphi_1 : P_1, \ldots, \varphi_k : P_k & \text{conditional} \\
& (\!|\Psi|\!) & \text{assertion process}
\end{array}$$

The process constructs are similar to those of the π-calculus; however, the object M of an input or output prefix can be an arbitrary term and in the input construct $\underline{M}(\lambda \boldsymbol{x})N.P$ the subject $(\lambda \boldsymbol{x})N$ is a *pattern* whose variable names $\boldsymbol{x}$ can occur free in N and P. Any term received on channel M must match this pattern; a term N_1 matches the pattern $(\lambda \boldsymbol{x})N$ if N_1 can be found by instantiating the variable names $\boldsymbol{x}$ in N with terms. Finally note that assertions (see below) can also be used as processes.

As we consider typed ψ-calculi, we assume that a name n in a restriction $(\nu n : T)P$ is annotated with a type T; types are introduced in Section 3.1.

To understand the properties of names, an important notion is that of a *nominal set*. Informally speaking, this is a set whose members can be affected by names being bound or swapped. If x is an element of a nominal set and $a \in \mathcal{N}$, we write $a\#x$, to denote that a is fresh for x; the notion extends to sets of names in the expected way.

Another difference from [4] is that the sets of types, assertions and terms are each assumed to be generated by a signature of constructors. A *nominal data type* [4] is a nominal set with internal structure. Let Σ be a signature. Then a nominal data type over Σ is a Σ-algebra, whose carrier set is a nominal set (for the precise definition, see [10]). In the nominal data types of ψ-calculi we use simultaneous term substitution $X[\boldsymbol{z} := \boldsymbol{Y}]$ – terms in $\boldsymbol{Y}$ replace the names in $\boldsymbol{z}$ in X.

Further, nominal data types are assumed to be *distributive*: for every function symbol f and term substitution σ acting on variable names, we have $f(M_1, \ldots, M_k)\sigma = f(M_1\sigma, \ldots, M_k\sigma)$. In other words, term substitution distributes over function symbols. This requirement of distributivity, which is also not required in [4], will ensure that a standard substitution lemma for type judgments will hold if the substitution is well-typed, i.e. if $\sigma(x)$ and x have the same type for any variable name $x \in \text{dom}(\sigma)$.

The set of types $\mathcal{T}$ is a nominal datatype, since names can appear in types. We let T range over $\mathcal{T}$ and let $\text{fn}(T)$ denote the set of free names in type T.

Terms, *assertions* and *conditions* belong to the following nominal data types.

T	terms	M, N
C	conditions	φ
A	assertions	Ψ

Given valid choices of $\mathbf{T}$, $\mathbf{C}$ and $\mathbf{A}$, the following operations on data types are always assumed:

$$\begin{array}{rr} \otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A} & \text{composition of assertions} \\ \dot{\leftrightarrow} : \mathbf{T} \times \mathbf{T} \to \mathbf{C} & \text{channel equivalence} \\ \mathbf{1} \in \mathbf{A} & \text{unit assertion} \\ \models \subseteq \mathbf{A} \times \mathbf{C} & \text{entailment} \end{array}$$

The notion of channel equivalence tells us which terms represent the same communication channel; $\dot{\leftrightarrow}$ thus appears in the rules describing communication and prefixes. The entailment relation $\models$ describes when conditions are true, given an assertion set, and is needed to describe the behaviour of conditionals and to decide channel equivalence.

For any process P, we let $\mathrm{n}(P)$ denote the support of P, i.e. the names of P, and let $\mathrm{fn}(P)$ denote the set of free names in P. This notion is also defined for terms.

2.2 Labelled Semantics

In ψ-calculi, the assertion information of a process P can be extracted as its *frame* $\mathcal{F}(P) = \langle E_P, \Psi_P \rangle$, where Ψ_P is the composition of assertions in P and E_P records the types of the names local to Ψ_P. We call these *qualified frames*, since (unlike [4]) names are now annotated with types. Composition of frames is defined by $\langle E_1, \Psi_1 \rangle \otimes \langle E_2, \Psi_2 \rangle = \langle E_1 E_2, \Psi_1 \otimes \Psi_2 \rangle$ whenever we have dom$(E_1)\#$dom(E_2), dom$(E_1)\#\Psi_2$, and dom$(E_2)\#\Psi_1$. Moreover, we write $(\nu b : T)F$ to mean $\langle b : T, E_F, \Psi_F \rangle$.

Definition 1 (Frame of a process).

$$\begin{array}{rr} \mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q) & \mathcal{F}((\nu b : T)P) = (\nu b : T)\mathcal{F}(P) \\ \mathcal{F}((\!|\Psi|\!)) = \langle \varepsilon, \Psi \rangle & \mathcal{F}(P) = \mathbf{1} \quad \textit{otherwise} \end{array}$$

Labelled transitions are of the form $\Psi \rhd P \xrightarrow{\alpha} P'$ where the label α is defined by the formation rules

$$\alpha ::= \tau \mid \underline{M}N \mid \overline{M}N \mid (\nu \boldsymbol{b} : \boldsymbol{T})\overline{M}N$$

We let $\mathrm{bn}((\nu \boldsymbol{b} : \boldsymbol{T})\overline{M}N) = \boldsymbol{b}$ and $\mathrm{bn}(\alpha) = \emptyset$ otherwise.

The transitions are given by the rules in Table 1. We omit the symmetric versions of (Com) and (Par). In (Com) we assume that $\mathcal{F}(P) = \langle E_P, \Psi_P \rangle$ and $\mathcal{F}(Q) = \langle E_Q, \Psi_Q \rangle$, where dom$(E_P)$ is fresh for all of Ψ, Ψ_Q, Q, M, and P – and that the symmetric freshness condition holds for E_Q. In (Par) we assume that $\mathcal{F}(Q) = \langle E_Q, \Psi_Q \rangle$, where dom$(E_Q)$ is fresh for Ψ, P and α. In (Case), if more than one condition holds, the choice between them is nondeterministic.

Our semantics differs from that of [4], since types may contain names and appear in restrictions, so the order of restrictions now matters. In the rule (Open) we write $\nu \boldsymbol{a} \cup \{b : T\}$ to denote the typed *sequence* $\boldsymbol{a}$ extended with $b : T$ and extend the side condition of [4] to deal with the case where an extruded name appears in the type. To see why this is necessary, assume a type $\mathbf{Ch}(b)$ of channels that can carry the name b and consider the process $(\nu b : T_1)(\nu c : \mathbf{Ch}(b))\overline{a}c.$. Here, both b and c must be extruded to make use of c as a channel, even though b does not appear as a name in $\overline{a}c$.

Table 1. Labelled transition rules (for name freshness conditions, see Section 2.2)

$$\textsc{(In)}\quad \frac{\Psi \models M \dot{\leftrightarrow} K}{\Psi \rhd \underline{M}(\lambda \boldsymbol{x})N.P \xrightarrow{\underline{K}N[\boldsymbol{x}:=\boldsymbol{L}]} P[\boldsymbol{x} := \boldsymbol{L}]} \qquad \textsc{(Out)}\quad \frac{\Psi \models M \dot{\leftrightarrow} K}{\Psi \rhd \overline{M}N.P \xrightarrow{\overline{K}N} P}$$

$$\textsc{(Com)}\quad \frac{\begin{array}{c}\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}(\nu \boldsymbol{a}:\boldsymbol{T})N} P' \\ \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}N} Q' \\ \Psi \otimes \Psi_P \otimes \Psi_Q \vdash M \dot{\leftrightarrow} K\end{array}}{\Psi \rhd P \mid Q \xrightarrow{\tau} (\nu \boldsymbol{a} : \boldsymbol{T})(P' \mid Q')} \quad \boldsymbol{a}\#Q \qquad \textsc{(Case)}\quad \frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \quad \Psi \models \varphi_i}{\Psi \rhd \textbf{case}\ \tilde{\varphi} : \tilde{P} \xrightarrow{\alpha} P'}$$

$$\textsc{(Par)}\quad \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathrm{bn}(\alpha)\#Q \qquad \textsc{(Scope)}\quad \frac{\Psi \rhd P \xrightarrow{\alpha} P' \quad b\#\alpha, \Psi}{\Psi \rhd (\nu b : T)P \xrightarrow{\alpha} (\nu b : T)P'}$$

$$\textsc{(Open)}\quad \frac{\Psi \rhd P \xrightarrow{\overline{M}(\nu \boldsymbol{a}:\boldsymbol{T})N} P'}{\Psi \rhd (\nu b : T_1)P \xrightarrow{\overline{M}(\nu \boldsymbol{a}:\boldsymbol{T}\cup\{b:T_1\})N} P'} \qquad \textsc{(Rep)}\quad \frac{\Psi \rhd P \mid !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'}$$

$$b\#\boldsymbol{a}, \Psi, M \text{ and } b \in \mathrm{n}(N) \cup \mathrm{n}(\boldsymbol{T}) \cup \mathrm{n}(T_1)$$

3 A Simple Type System for ψ

Our type system extends that of the simply typed π-calculus of [23] in two ways. Firstly, we assign types to messages, and secondly, typability may depend on assertions.

3.1 Types and Type Environments

In type systems for process calculi with names, a type environment records the types of free names. Since typability in our setting can also depend on assertions, a type environment E can also contain assertions. We define the set of type environments by

$$E ::= E, x : T \mid E, \Psi \mid \emptyset$$

A type environment E is *well-formed*, written $E \vdash \diamond$, if it is a finite partial function from names to types such that if $E = E_1, \Psi, E_2$ and x is a name in Ψ, then $x \in \mathrm{dom}(E_1)$. We refer to the type annotations of E as $\mathsf{E}(E)$ and to the composed assertion $\bigotimes_{\Psi \in E} \Psi$ as $\Psi(E)$.

Definition 2. *We write $E_1 <_T E_2$ if $E_1 \vdash \diamond$, $E_2 \vdash \diamond$, $E_1 = E_{10}, E_{11}, E_{12}, \ldots, E_{1(k+1)}$ and $E_2 = E_{10}, u_1 : T_1, E_{11}, \ldots, u_k : T_k, E_{1k}, E_{1(k+1)}$ for some $u_1 : T_1, \ldots, u_k : T_k$.*

Thus, $E <_T E'$ if E' extends E with additional type annotations.

Definition 3. *Let E and E' be type environments. We write $E <^0_A E'$ if $E' = E, \Psi$ for some assertion Ψ. We let $<_A$ denote the transitive closure of $<^0_A$ and let $<$ denote the least preorder containing $<_T \cup <_A$.*

3.2 Type Judgements and Type Rules

A type system for a ψ-calculus must have type judgements for each of the three nominal datatypes: $E \vdash M : T$, $E \vdash \varphi$ and $E \vdash \Psi$. Let $\mathcal{J}$ range over judgements, i.e.

$$\mathcal{J} ::= M : T \mid \varphi \mid \Psi$$

We consider only *qualified* judgements $\mathcal{J}_E$ of the form $E \vdash \mathcal{J}$ where $\mathrm{fn}(\mathcal{J}) \subseteq dom(E)$.

Definition 4. *Let $\mathcal{J}_E = E \vdash \mathcal{J}$ and $\mathcal{J}'_E = E' \vdash \mathcal{J}'$ be qualified judgements. We write $\mathcal{J}_E < \mathcal{J}'_E$ if $\mathcal{J} = \mathcal{J}'$ and $E < E'$.*

General Assumptions. We make a series of general assumptions concerning our type rules that are sufficient for establishing usual properties – in particular, the weakening and strengthening properties – of any concrete instance of our type system.

A natural assumption is that bindings in the type environment can be used:

$$(\text{VAR}) \quad E \vdash x : T \quad \text{if } E(x) = T$$

Type rules have zero or more *premises* and a *conclusion* that are qualified judgements and may also include a *side condition*; a side condition is a predicate that is not a qualified type judgement but can depend on judgements in the rule. A side condition is dependent on the qualified judgements $\boldsymbol{\mathcal{J}_E}$ of the rule it occurs in and is therefore denoted $\chi(\boldsymbol{\mathcal{J}_E})$. We require the following to hold for all rules for terms and assertions.

- Every side condition χ must be *monotone* wrt. environment extensions. Let $\boldsymbol{\mathcal{J}_E}$ be an arbitrary instance of a rule. Suppose that whenever $\chi(\boldsymbol{\mathcal{J}_E})$ holds and $\boldsymbol{\mathcal{J}_E} < \boldsymbol{\mathcal{J}'_E}$ for another rule instance $\boldsymbol{\mathcal{J}'_E}$ then also $\chi(\boldsymbol{\mathcal{J}'_E})$. Then χ is monotone.
- Every side condition χ must be *topical*; removing unnecessary type annotations will not affect the validity of a side condition. Let $\boldsymbol{\mathcal{J}_E}$ be an arbitrary instance of a rule, and suppose that whenever $\boldsymbol{\mathcal{J}_E} <_T \boldsymbol{\mathcal{J}'_E}$ for some other instance $\boldsymbol{\mathcal{J}'_E}$ and $\chi(\boldsymbol{\mathcal{J}'_E})$ holds, then also $\chi(\boldsymbol{\mathcal{J}_E})$. Then χ is topical.
- Assertion typability must *respect composition* of environment assertions. If $E \vdash \Psi$, then $\mathsf{E}(E), \Psi(E) \vdash_E \Psi$.

Finally, we require *compositionality.*

- For composite assertions, we require that there is a compositional rule

$$\frac{E \vdash \Psi_1 \quad E \vdash \Psi_2}{E \vdash \Psi_1 \otimes \Psi_2}$$

- If a type rule types a composite condition $g(\varphi_1, \ldots, \varphi_k)$, it must be of the form

$$\frac{E \vdash \varphi_i \quad 1 \leq i \leq k \quad \chi}{E \vdash g(\varphi_1, \ldots, \varphi_k)}$$

- If a type rule types a composite term $f(M_1, \ldots, M_k)$, it must be of the form

$$\frac{E \vdash M_i : T_i \quad 1 \leq i \leq k \quad \chi}{E \vdash f(M_1, \ldots, M_k) : T}$$

We do not impose any constraints on how the result type arises from the types of immediate constituents, but we require channels to have the same type in environment E, if they are equivalent for an assertion that is well-typed wrt. E.

$$\text{If } E \vdash M : T \text{ , } E \vdash \Psi \text{ and } \Psi \models M \dot{\leftrightarrow} N \text{ then } E \vdash N : T \qquad (1)$$

To deal with pattern matching in inputs, we introduce the following type rule for message patterns:

$$(\textsc{Pattern}) \quad \frac{E, \boldsymbol{x} : \boldsymbol{T} \vdash M : U}{E \vdash (\lambda \boldsymbol{x})M : \boldsymbol{T} \to U}$$

If an input abstraction has type $\boldsymbol{T} \to U$, it will receive any term of type U if this term contains pattern variables of types corresponding to $\boldsymbol{T}$.

Type Rules. The type rules are found in Table 2. In the rule (Par), for each component P we collect the relevant assertions Ψ_P and type bindings E_P associated with bound names that may occur in the types and include these when typing the other component. In the rules (In) and (Out), the type of the subject M and the type of the object (the term transmitted on channel M) must be compatible wrt. a *compatibility predicate* $\looparrowleft$.

$$(\textsc{In}) \quad \frac{\begin{array}{c} E, \boldsymbol{x} : \boldsymbol{T} \vdash P \\ E \vdash (\lambda \boldsymbol{x})N : \boldsymbol{T} \to U_o \\ E \vdash M : U_s \end{array}}{E \vdash \underline{M}(\lambda \boldsymbol{x})N.P} \quad U_s \looparrowleft U_o$$

$$(\textsc{Out}) \quad \frac{E \vdash M : T_s \quad E \vdash N : T_o \quad E \vdash P}{E \vdash \overline{M}N.P} \quad T_s \looparrowleft T_o$$

$$(\textsc{Par}) \quad \frac{E, E_{P_2}, \Psi_{P_2} \vdash P_1 \quad E, E_{P_1}, \Psi_{P_1} \vdash P_2}{E \vdash P_1 \mid P_2}$$

$$(\textsc{Res}) \quad \frac{E, x : T \vdash P}{E \vdash (\nu x : T)P} \qquad (\textsc{Rep}) \quad \frac{E \vdash P}{E \vdash !P}$$

$$(\textsc{Ass}) \quad \frac{E \vdash \Psi}{E \vdash (\!|\Psi|\!)} \qquad (\textsc{Case}) \quad \frac{E \vdash \varphi_i \quad E \vdash P_i \quad 1 \leq i \leq k}{E \vdash \textbf{case } \varphi_1 : P_1, \ldots, \varphi_k : P_k}$$

Table 2. Type rules for processes

Example 1. To capture a type system with channel types such as the original sorting system by Milner [17] we can let $\looparrowleft$ be defined by $T_1 \looparrowleft T_2$ if $T_1 = \mathbf{Ch}(T_2)$.

4 Properties of the Simple Type System

Type systems normally guarantee two properties of well-typed programs. A *subject reduction* property ensures that if a program is well-typed, it stays well-typed under subsequent reduction steps. A *safety property* ensures that if a program is well-typed, a certain safety predicate will hold. We now consider these two properties in our setting.

4.1 Standard Lemmas

The following standard properties follow from the assumptions of compositionality, monotonicity and topicality.

Lemma 1. *The following properties hold for all instances that satisfy the general assumptions of Section 3.2.*

Substitution. *Let σ be a term substitution. If $E \vdash \mathcal{J}$, $dom(E) = dom(\sigma)$ and $E \vdash \sigma(x) : E(x)$ for all $x \in dom(\sigma)$ then $E \vdash \mathcal{J}\sigma$*
Interchange. *If $E, E_1, x : T, E_2 \vdash \mathcal{J}$ and $x \notin dom(E_1)$ and $x \notin \text{fn}(E_1)$ then $E, x : T, E_1, E_2 \vdash \mathcal{J}$.*
Weakening. *If $E \vdash \mathcal{J}$ and $E, E' \vdash \diamond$ then $E, E' \vdash \mathcal{J}$*
Strengthening. *If $E, x : T_1 \vdash \mathcal{J}$ and $x \notin \text{fn}(\mathcal{J})$ then $E \vdash \mathcal{J}$*

4.2 The Subject Reduction Property

Our type system guarantees a subject reduction property, namely that typability is preserved under τ-actions.

Theorem 1. *Suppose $E \vdash \Psi$ and $E \vdash P$ and that $\Psi \rhd P \xrightarrow{\tau} P'$. Then also $E \vdash P'$.*

Proof. (Sketch) Induction in the labelled transition rules. We first establish a lemma for transitions not labelled with τ: that if $E \vdash \Psi$, $E \vdash P$ and $\Psi \rhd P \xrightarrow{\alpha} P'$ where $\alpha \neq \tau$ and E' is any type environment such that $E, E' \vdash \alpha$ (i.e. the terms in α can be typed), then $E, E' \vdash P'$. The proof of the theorem then consists in an induction in the rules defining τ-transitions.

4.3 Safety in the Simple Type System

The notion of safety depends on the particular instantiation of the type system. However, all instantiations guarantee a general notion of channel safety.

Given a predicate on process configurations, $\mathsf{now}(E, \Psi, P)$, which defines a notion of *now-safety* of process P relative to a type environment E and an assertion Ψ. We can then define the general notion of safety as invariant now-safety.

Definition 5. *Given assertion Ψ, process P, type environment E and predicate* now*, $\Psi \rhd P$ is* safe *for E if for any P' where $\Psi \rhd P \xrightarrow{\tau}^{*} P'$, we have that $\mathsf{now}(E, \Psi, P')$.*

Following this approach, we can show that an instance of the simple type system has a safety property by defining the property in terms of a suitable notion of now-safety and then showing that typability implies now-safety.

The side conditions of the type rules (IN) and (OUT) guarantee a general form of *channel safety*. This property guarantees that channels are always used to transmit messages whose type is compatible with that of the channel. Note that in Definition 6 we may need to extend the type environment; this is the case, since an input action may involve the reception of a term containing extruded names.

Definition 6. *Define* $\mathsf{now}_{\mathrm{Ch}}(E, \Psi, P)$ *to hold if when* $\Psi \rhd P \xrightarrow{\alpha} P'$ *and* $E \vdash \Psi$*, then*

1. *If* $\alpha = \overline{M}(\nu \boldsymbol{x} : \boldsymbol{T})N$ *and* $E \vdash M : U_s$ *, then we have* $E, \boldsymbol{x} : \boldsymbol{T}, \Psi_1 \otimes \cdots \otimes \Psi_k \vdash N : U_o$ *for some* $\Psi_1, \ldots, \Psi_k$ *and* U_o *where* $U_s \looparrowleft U_o$
2. *If* $\alpha = \underline{M}N$*, if* $\mathrm{fn}(N) = \boldsymbol{x}$ *and it is the case that* $E \vdash M : U_s$*, then for some* U_o*,* $\boldsymbol{T}$ *and* $\Psi_1, \ldots, \Psi_k$ *we have* $E, \boldsymbol{x} : \boldsymbol{T}, \Psi_1 \otimes \cdots \otimes \Psi_k \vdash N : U_o$*, such that* $U_s \looparrowleft U_o$.

We say that $\Psi \rhd P$ *is* now-channel safe for E.

Theorem 2. *If* $E \vdash P$ *and* $E \vdash \Psi$*, then* $\mathsf{now}_{\mathrm{Ch}}(E, \Psi, P)$.

Safety now follows from the subject reduction property of Theorem 1.

Theorem 3. *If* $E \vdash \Psi$ *and* $E \vdash P$ *then* $\Psi \rhd P$ *is channel-safe for* E.

5 Instances of the Simple Type System

We now describe how a series of new and existing type systems can be expressed as instances of our simple type system and how their safety properties can be established using the general results that we now have.

5.1 The Calculus of Explicit Fusions

In [4], a ψ-calculus translation is given of the calculus of explicit fusions of Wischik and Gardner [24]. No type systems were formulated for this calculus; here is one.

Fusions express that names a and b are considered equivalent and can therefore be represented as assertions. For any binary relation R, we let $\mathrm{EQ}(R)$ denote its reflexive, transitive and symmetric closure. The ψ-calculus representation of the calculus explicit fusions is now as follows.

The set of terms $\mathbf{T}$ is taken to be $\mathcal{N}$. The set of conditions $\mathbf{C}$ is the set of name identities of the form $a = b$ with $a, b \in \mathbf{T}$. The assertion set $\mathbf{A}$ is the family of finite sets of identities of the form $\{a_1 = b_1, \ldots, a_n = b_n\}$ for $n \geq 0$. Composition $\otimes$ is set union, the identity assertion $\mathbf{1}$ is $\emptyset$ and the entailment relation is defined by

$$\Psi \models a = b \text{ if } a = b \in \mathrm{EQ}(\Psi)$$

In the calculus of explicit fusions, a relevant notion of safety is that only names of the same type will ever be fused.

Definition 7. P *is now-safe wrt.* E *if for any assertion* $a = b$ *in* P*,* $E(a) = E(b)$.

To capture this, we use a notion of channel type. Names have types of the form

$$T ::= \mathbf{Ch}(T) \mid X$$

where X ranges over the set of *type variables* **TVar**. Actual types are defined by means of a *sorting*, which is a finite function $\Delta : \mathbf{TVar} \rightarrow \mathcal{T}$. Again, the compatibility relation is defined by $\mathbf{Ch}(T) \looparrowleft T$. The type rule for assertions is then as follows.

$$E \vdash \{a_1 = b_1, \ldots, a_n = b_n\} \quad \text{if } E(a_i) = E(b_i) \text{ for } 0 \leq i \leq n$$

For conditions, the type rules are similar:

$$E \vdash a = b \quad \text{if } E(a) = E(b) \qquad\qquad E \vdash a \dot{\leftrightarrow} b \quad \text{if } E(a) = E(b)$$

The safety result is now the following.

Theorem 4. *If $E \vdash P$, then P is safe wrt. E.*

5.2 The Distributed π-calculus

The distributed π-calculus was first proposed by Hennessy and Riely [20]. In Dπ, processes are located at named locations ($l[P]$) and can migrate to a named location (**go** $k.P$). In our version P ranges over the set of processes and N over networks.

$$\begin{aligned} P &::= \mathbf{0} \mid \overline{a}\langle x\rangle.P_1 \mid a(x).P_1 \mid P_1 \mid P_2 \mid (\nu n : T)P_1 \mid !P_1 \mid \mathbf{go}\ k.P_1 \\ N &::= \mathbf{0} \mid N_1 \mid N_2 \mid (\nu n : T)N_1 \mid l[P] \end{aligned}$$

The labelled transition semantics, which involves a notion of structural congruence $\equiv$, has transitions of the form $P \xrightarrow{l@\alpha} P'$ for processes and $N \xrightarrow{\alpha} N'$ for networks, where either $\alpha = \overline{a}n$, $\alpha = an$ or $\alpha = \tau$.

Dπ can be represented as a ψ-calculus by a translation, due to Carbone and Maffeis [5]. The central insight is to view a channel a at location l as a composite term $l \cdot a$, so the set of terms is $\mathbf{T} = \{l \cdot a \mid l, a \in \mathcal{N}\}$. This set is not closed wrt. arbitrary substitutions, but the set of well-typed terms is closed under well-typed substitutions, which suffices. The set of conditions $\mathbf{C}$ is $\mathbf{C} = \{l \cdot a \dot{\leftrightarrow} l \cdot a \mid l, a \in \mathcal{N}\}$.

For networks, the translation is

$$\begin{aligned} [\![l[P]]\!] &= [\![P]\!]_l & [\![\mathbf{0}]\!] &= \mathbf{1} \\ [\![N_1 \mid N_2]\!] &= [\![N_1]\!] \mid [\![N_2]\!] & [\![(\nu n)N_1]\!] &= (\nu n)[\![N_1]\!] \end{aligned}$$

For processes, we have

$$\begin{aligned} [\![\mathbf{go}\ k.P]\!]_l &= [\![P]\!]_k & [\![a(x).P]\!]_l &= (l \cdot a)(x).[\![P]\!]_l \\ [\![\overline{a}\langle x\rangle.P]\!]_l &= \overline{(l \cdot a)}\langle x\rangle.[\![P]\!]_l & [\![P_1 \mid P_2]\!]_l &= [\![P_1]\!]_l \mid [\![P_2]\!]_l \\ [\![!P_1]\!]_l &= ![\![P_1]\!]_l & [\![(\nu n)P]\!] &= (\nu n)[\![P]\!] \end{aligned}$$

The only assertion is the trivial assertion $\mathbf{1}$; we always have $E \vdash \mathbf{1}$.

Most type systems for Dπ are concerned about notions of channel safety. Here, we describe a type system that assigns types to both location and channel names. We therefore consider types of the form

$$T ::= \mathbf{Ch}(T) \mid \mathbf{Loc}\{a_i : \mathbf{Ch}(T_i)\}_{i \in I} \mid B$$

where I is a finite index set and B ranges over a set of base types. A location type $\mathbf{Loc}\{a_i : \mathbf{Ch}(T_i)\}_I$ describes the available interface of a location: only the specified channel names can be used for communication at locations of this particular type. The type rule for composite names is then

$$\frac{E \vdash l : \mathbf{Loc}\{a_i : \mathbf{Ch}(T_i)\}_{i \in I} \quad E \vdash a_i : \mathbf{Ch}(T_i) \quad \text{for some } i \in I}{E \vdash l \cdot a_i : \mathbf{Ch}(T_i)}$$

where I is a finite set. The compatibility relation is given by $\mathbf{Ch}(T) \hookleftarrow T$

Definition 8. *A* Dπ *network N is now-safe wrt. E if whenever $N \equiv (\nu \boldsymbol{k})l[P] \mid N'$ and $P \xrightarrow{l@\alpha}$ where either $\alpha = \overline{a}n$ or $\alpha = an$, then $E(l) = \boldsymbol{Loc}\{a_i : \boldsymbol{Ch}(T_i)\}_I$ where $a = a_i$ and $E(n) = T_i$ for some $i \in I$.*

The following result follows from Theorem 3.

Theorem 5. *Let $P = [\![N]\!]$ for a* Dπ *network N. If $E \vdash P$, then N is safe wrt. E.*

5.3 A Type System for Secrecy

Next, we show how to represent a type system for secrecy in the spi calculus due to Abadi and Blanchet [2]. The version of the spi calculus used in [2] has primitives for asymmetric cryptography and its formation rules are

$$\begin{aligned} M &::= x \mid a \mid \{\boldsymbol{M}\}_M \\ P &::= \overline{M}\langle \boldsymbol{M} \rangle \mid a(x_1, \ldots, x_n).P \mid \mathbf{0} \mid P|Q \mid !P \mid (\nu a)P \\ &\quad \mid \textbf{case } M \textbf{ of } \{x_1, \ldots, x_n\}_a : P \textbf{ else } Q \mid \textbf{if } M = N \textbf{ then } P \textbf{ else } Q \end{aligned}$$

The term $\{\boldsymbol{M}\}_M$ denotes that the tuple $\boldsymbol{M} = (M_1, \ldots, M_k)$ is encrypted with key M. The process constructs resemble those of ψ-calculi, but output is asynchronous, and the subject of an input must be a *name* a (as opposed to a variable x). Similarly, in a decryption **case** M **of** $\{x_1, \ldots, x_n\}_k : P$ **else** Q, the key k must be a name. Thus, input and decryption capabilities cannot be transmitted – the latter lets us distinguish between private keys (only used for decryption) and public keys (only used for encryption).

The representation of this as a ψ-calculus is straightforward and based on [4]. We introduce conditions $M = \mathsf{enc}(*, k)$ and $M \neq \mathsf{enc}(*, k)$ whose intended meaning is that M is, resp. is not, encrypted using key k. Similarly, we introduce match conditions $M = N$ and mismatch $M \neq N$.

The decryption construct now becomes

$$(\nu \boldsymbol{x})(\textbf{case } M = \mathsf{enc}(*, k) : ((\!|\boldsymbol{x} = \mathsf{dec}(M, k)|\!) \mid P) \ ; \ M \neq \mathsf{enc}(*, k) : Q)$$

and the biconditional construct is **case** $M = N : P \; ; M \neq N : Q$. Channel equality $\dot{\leftrightarrow}$ is defined to be the identity relation on terms.

Abadi and Blanchet consider a notion of *secrecy under all opponents* [2]. If RW is a finite set of names and W a finite set of closed terms, then an (RW, W)-opponent is a spi-calculus process Q such that $Q = Q'[\boldsymbol{x} := \boldsymbol{N}]$ where $\mathrm{fn}(Q') \subseteq RW$, $W = \boldsymbol{N}$ and such that the set of free variables of Q' contains at most $\boldsymbol{x}$.

Definition 9. *Let RW be a finite set of names and W a finite set of closed terms.. Process P preserves the secrecy of M wrt. RW if whenever $P \xrightarrow{\tau}^{*} P'$ then it is not the case that $P' \xrightarrow{\overline{c}M} P''$ for any P'' or $c \in RW$. P preserves the secrecy of M from (RW, W) if $P|Q$ preserves the secrecy of M wrt. RW for any (RW, W)-opponent Q.*

The types given in [2] have the syntax

$$T ::= \mathsf{Pub} \mid \mathsf{Sec} \mid C^{\mathsf{Pub}}[\boldsymbol{T}] \mid C^{\mathsf{Sec}}[\boldsymbol{T}] \mid K^{\mathsf{Sec}}[\boldsymbol{T}] \mid K^{\mathsf{Pub}}[\boldsymbol{T}]$$

where $\boldsymbol{T}$ is any tuple of types. $C^{\mathsf{Pub}}[\boldsymbol{T}]$ and $C^{\mathsf{Sec}}[\boldsymbol{T}]$ are channel types for sending public, resp. secret, data and $K^{\mathsf{Sec}}[\boldsymbol{T}]$ and $K^{\mathsf{Pub}}[\boldsymbol{T}]$ are key types for encrypting these.

Abadi and Blanchet introduce a subtype relation which states that public types are subtypes of Pub (so e.g. $C^{\mathsf{Pub}}[\boldsymbol{T}] \leq \mathsf{Pub}$) and secret types are subtypes of Sec. Moreover, a special judgment $E \vdash M : \mathcal{S}$ describes the set of types $\mathcal{S}$ that M can have.

In our representation of the type system we must distinguish between names and variables. Our syntax of types is therefore

$$T ::= \mathsf{Pub} \mid \mathsf{Sec} \mid C^{A}_{B}[\boldsymbol{T}] \mid K^{A}_{B}[\boldsymbol{T}] \mid \boldsymbol{T}$$

where $A \in \{\mathsf{Pub}, \mathsf{Sec}\}$ and $B \in \{\mathsf{Name}, \mathsf{Var}\}$. We capture the subtype relation and the type set judgment by rules including the following.

$$(\textsc{PubVar}) \quad \frac{E(x) = \mathsf{Pub}}{E \vdash x : C^{\mathsf{Pub}}_{\mathsf{Var}}[\boldsymbol{T}]} \qquad (\textsc{KeyP}) \quad \frac{E \vdash M : C^{\mathsf{Pub}}_{\mathsf{Var}}[\boldsymbol{T}]}{E \vdash M : \mathsf{Pub}}$$

The original type rules for the decryption construct are now captured via type rules for assertions, two of which are shown below.

$$(\textsc{Dec-PK1}) \quad \frac{\begin{array}{c} E \vdash M : \mathsf{Pub} \quad E(k) = K^{\mathsf{Pub}}_{\mathsf{Name}}[\boldsymbol{T}] \\ E(x_i) = T_i \\ \text{for } 1 \leq i \leq |\boldsymbol{x}| \quad |\boldsymbol{T}| = |\boldsymbol{x}| \end{array}}{E \vdash \boldsymbol{x} = \mathsf{dec}(M, k)} \qquad (\textsc{Equal}) \quad \frac{E \vdash M : T \quad E \vdash N : T}{E \vdash M = N}$$

For the assertion $M = N$, the type system of [2] allows **if** $M = N$ **then** P **else** Q to be well-typed if $E \vdash M : S_1$ and $E \vdash N : S_2$ but $S_1 \cap S_2 = \emptyset$. This is not allowed in our type system; our rules mirror the stronger requirement (originally made in [1]) that $S_1 \cap S_2 \neq \emptyset$. The remaining assertions are defined to be always well-typed.

For the input and output rules the original type system allows channels of type $C^{\mathsf{Pub}}_{B}[\boldsymbol{T}]$ to transmit either messages of type Pub or a tuple of type $\boldsymbol{T}$. We can capture this by defining the compatibility predicate as follows (where **Pub** stands for a tuple type of arbitrary length, all of whose components are Pub).

$$C^{\mathsf{Pub}}_{B}[\boldsymbol{T}] \hookrightarrow \textbf{Pub} \qquad C^{\mathsf{Pub}}_{B}[\boldsymbol{T}] \hookrightarrow \boldsymbol{T} \qquad C^{\mathsf{Sec}}_{B}[\boldsymbol{T}] \hookrightarrow \boldsymbol{T}$$

The secrecy result for the type system is the following.

Theorem 6 ([2]). *Let P be a closed process. Suppose that $E \vdash P$ and $E \vdash s : \mathsf{Sec}$. Let*

$$RW = \{a \mid E(a) = \mathsf{Pub}\} \quad W = \{a' \mid E(a') = C^{\mathsf{Pub}}[\cdots] \text{ or } E(a') = K^{\mathsf{Pub}}[\cdots]\}$$

Then P preserves the secrecy of s from (RW, W).

We obtain this safety result from Theorem 3 by first noticing that channel safety implies that if a process R satisfies that $E \vdash R$, then a secret name s cannot be leaked on any channel in RW as defined. To conclude the proof, now apply the result (established in [2]) that $E \vdash P \mid Q$ for any (RW, W)-opponent Q.

5.4 Correspondence Types for Authenticity

Next, we represent a type and effect system for a spi-calculus capturing non-injective authenticity using correspondence assertions[9]. This case shows how the typability of processes and terms can depend on assertions.

We now consider symmetric encryption, and the decryption operation is written as **case** M **of** $\{x_1, \ldots, x_n\}_M : P$ **else** Q, where the key M can be an arbitrary term.

We introduce *correspondence assertions* **begin** $\ell(M)$ and **end** $\ell(M)$; these are labelled message terms where ℓ ranges over a set of labels disjoint from the set of message terms. In a process, **begin** assertions are placed where authentication is to be initiated, and **end** assertions are placed where authentication should be completed.

The set of message terms is defined by the formation rules

$$M ::= x \mid (M_1, M_2) \mid \{M_1\}_{M_2} \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \mid \mathbf{ok}$$

where $\mathbf{fst}\ M$ and $\mathbf{snd}\ M$ extract the first, respectively second coordinate of a pair M. The $\mathbf{ok}$ term is an *explicit effect term* used to transfer the capability to match **begin** assertions. $\mathbf{ok}$ terms and correspondence assertions do not influence process behaviour; their only role is in the type system.

Definition 10. *A process P is* safe *for type environment E if whenever $P \xrightarrow{\tau}^{*} (\nu \boldsymbol{n})(\boldsymbol{end}\ \ell(M) \mid P')$, then either we have $P' \equiv (\!|\boldsymbol{begin}\ \ell(M)|\!) \mid P^{(2)}$ or $\boldsymbol{begin}\ \ell(M) \in E$. An* opponent *is a spi calculus process Q that has no* **end** *assertions and where every term in P can be typed with opponent type* $\mathbf{Un}$. *Process P is* robustly safe *if $P \mid Q$ is safe for any opponent Q.*

Here, $\equiv$ is the structural congruence relation of the spi calculus [3]. Note that the safety property employs the following notion of now-safety: P is now-safe, if every end-assertion can be matched by a begin-assertion with the same label.

The important property is similar to that for the secrecy type system: if P can be well-typed when all its free variables have opponent type $\mathbf{Un}$, then P is robustly safe.

Theorem 7. *[9] If $x_1 : \mathbf{Un}, \ldots, x_k : \mathbf{Un} \vdash P$ where $\{x_1, \ldots, x_k\} = \mathrm{fv}(P)$, then P is robustly safe.*

The representation of the spi calculus is as in Section 5.3, and we can keep the original types of [9].

$$T ::= \mathbf{Ch}(T) \mid \mathbf{Pair}(x : T_1, T_2) \mid \mathbf{Ok}(S) \mid \mathbf{Un}$$

Here $\textbf{Pair}(x : T_1, T_2)$ is a dependent pair type, $\textbf{Ok}(S)$ is an ok-type, where S is a finite set of assertions called an *effect*, and $\mathbf{Un}$ is an *opponent type*. We incorporate opponent types by defining the compatibility relation by $\mathbf{Un} \looparrowleft \mathbf{Un}$.

Correspondence assertions **begin** $\ell(M)$ and **end** $\ell(M)$ are added to the assertions $\mathbf{A}$ of Section 5.3. Their type rules are defined below. All other assertions are assumed to be always well-typed. Note that there are two rules for end-assertions, since effects can either occur directly in the type environment or be hidden within an ok-type. Also note that these rules show how typability can depend on assertions in the type environment.

$$(\textsc{Begin})\ E \vdash \textbf{begin}\ \ell(M) \qquad (\textsc{End-1})\ \frac{E = E_1, \ell(M), E_2 \quad \text{fn}(M) \subseteq \text{dom}(E_1)}{E \vdash \textbf{end}\ \ell(M)}$$

$$(\textsc{End-2})\ \frac{\begin{array}{c} E = E_1, x : \textbf{Ok}(S), E_2 \\ \ell(M) \in S \end{array}}{E \vdash \textbf{end}\ \ell(M)} \qquad \text{fn}(M) \subseteq \text{dom}(E_1)$$

Some of the type rules for terms are shown below.

$$(\textsc{Enc})\ \frac{E \vdash M : T \quad E \vdash N : \textbf{Key}(T)}{E \vdash \{M\}_N : \mathbf{Un}} \qquad (\textsc{Ok})\ \frac{E \vdash \diamond \quad E \vdash \psi \quad \forall \psi \in S}{E \vdash \textbf{ok} : \textbf{Ok}(S)}$$

$$(\textsc{Enc Un})\ \frac{E \vdash M : \mathbf{Un} \quad E \vdash N : \mathbf{Un}}{E \vdash \{M\}_N : \mathbf{Un}} \qquad (\textsc{Ok Un})\ \frac{E \vdash \diamond}{E \vdash \textbf{ok} : \mathbf{Un}}$$

The authenticity result of Theorem 7 can now be established by Theorem 1 combined with a lemma that every well-typed process is now-safe wrt. correspondences. This is easily proved by induction in the type derivation of a well-typed P.

6 Conclusions and Further Work

In this paper we have presented a simple type system for ψ-calculi where term types belong to a nominal data type and judgements for processes are of the form $E \vdash P$ and given by a fixed set of rules. Terms, assertions and conditions are assumed to form nominal datatypes, and only a few requirements on type rules are imposed, such as compositionality and substitutivity. The type system lets us represent existing type systems for secrecy and authenticity in the spi calculus and location safety in the distributed π-calculus and also gives rise to a first type system for the calculus of explicit fusions.

The type system represents forms of channel subtyping by a compatibility relation and deals with opponent typability, but a more general account of subtyping is another important line of future research.

Other type systems in the literature deal with *resource-aware* properties such as linearity or receptiveness of names [14,22], or notions of termination [7,15]. Common to these systems is that the rules for parallel composition and prefixes are modified and the use of replication is limited; most often by only allowing replicated inputs. An extension of our work to such type systems is a topic of ongoing work.

References

1. Abadi, M.: Secrecy by typing in security protocols. J. ACM 46(5), 749–786 (1999)
2. Abadi, M., Blanchet, B.: Secrecy types for asymmetric communication. Theor. Comput. Sci. 298(3), 387–415 (2003)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: the spi calculus. In: Proc. CCS 1997, pp. 36–47. ACM, New York (1997)
4. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: Mobile Processes, Nominal Data, and Logic. In: Proc. of LICS 2009, pp. 39–48. IEEE, Los Alamitos (2009)
5. Carbone, M., Maffeis, S.: On the expressive power of polyadic synchronisation in π-calculus. Nordic Journal of Computing 10(2), 70–98 (2003)
6. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, p. 140. Springer, Heidelberg (1998)
7. Deng, Y., Sangiorgi, D.: Ensuring termination by typability. Inf. Comput. 204(7), 1045–1082 (2006)
8. Elsborg, E., Hildebrandt, T., Sangiorgi, D.: Type systems for bigraphs. In: Kaklamanis, C., Nielson, F. (eds.) TGC 2008. LNCS, vol. 5474, pp. 126–140. Springer, Heidelberg (2009)
9. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. ACM Trans. Program. Lang. Syst. 29(5) (2007)
10. Gabbay, M.J., Mathijssen, A.: Nominal (universal) algebra: Equational logic with names and binding. J. Log. Comput. 19(6), 1455–1508 (2009)
11. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. J. Comput. Secur. 11(4), 451–519 (2003)
12. Honda, K.: Composing processes. In: Proc. of POPL 1996, pp. 344–357. ACM, New York (1996)
13. Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. Theor. Comput. Sci. (11), 121–163 (2004)
14. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. 21(5), 914–947 (1999)
15. Kobayashi, N., Sangiorgi, D.: A hybrid type system for lock-freedom of mobile processes. ACM Trans. Program. Lang. Syst. 32(5) (2010)
16. Makholm, H., Wells, J.B.: Instant polymorphic type systems for mobile process calculi: Just add reduction rules and close. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 389–407. Springer, Heidelberg (2005)
17. Milner, R.: The polyadic pi-calculus: a tutorial, pp. 203–246. Springer, Heidelberg (1993)
18. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press, Cambridge (2009)
19. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
20. Riely, J., Hennessy, M.: A typed language for distributed mobile processes. In: Proceedings of POPL 1998, pp. 378–390. ACM, New York (1998)
21. Riely, J., Hennessy, M.: Trust and partial typing in open systems of mobile agents. J. Autom. Reasoning 31(3-4), 335–370 (2003)
22. Sangiorgi, D.: The name discipline of uniform receptiveness. Theor. Comput. Sci. 221(1-2), 457–493 (1999)
23. Sangiorgi, D., Walker, D.: The π-Calculus - A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
24. Wischik, L., Gardner, P.: Explicit fusions. Theor. Comput. Sci. 340(3), 606–630 (2005)

Full Abstraction in a Subtyped pi-Calculus with Linear Types

Romain Demangeon and Kohei Honda

Queen Mary, University of London

Abstract. We introduce a concise pi-calculus with directed choices and develop a theory of subtyping. Built on a simple behavioural intuition, the calculus offers exact semantic analysis of the extant notions of subtyping in functional programming languages and session-based programming languages. After illustrating the idea of subtyping through examples, we show type-directed embeddings of two known subtyped calculi, one for functions and another for session-based communications. In both cases, the behavioural content of the original subtyping is precisely captured in the fine-grained subtyping theory in the pi-calculus. We then establish full abstraction of these embeddings with respect to their standard semantics, Morris's contextual congruence in the case of the functional calculus and testing equivalence for the concurrent calculus. For the full abstraction of the embedding of the session-based calculus, we introduce a new proof method centring on non-deterministic computational adequacy and definability. Partially suggested by a technique used by Quaglia and Walker for their full abstraction result, the new proof method extends the framework used in game-based semantics to the May/Must equivalences, giving a uniform proof method for both deterministic and non-deterministic languages.

1 Introduction

A subtyping is a form of polymorphism where we can assign to a program a type which is more inclusive than the original type of the program, called subsumption. This notion of inclusion forms a partial order on types, where "more inclusive" may most simply be interpreted as having more inhabitants satisfying the type specification. In the standard subtyping theories, this inclusiveness is structurally calculable from the construction of types, such as through the well-known variance rule for arrow types, records and variants, cf. [2]. The notion of subtyping plays a key role in the practice of programming languages [20].

In this paper we study a simple theory of subtyping for interacting processes and show that it subsumes extant notions of subtyping in programming languages through encoding. We first introduce a concise pi-calculus with directed choices and linear types, and develop a theory of subtyping purely based on these choices. The resulting calculus is called $\pi^{\{1,\bar{1},\omega\}}$ for brevity. After illustrating a simple behavioural intuition behind the subtyping theory through examples, we show that the calculus offers exact semantic analysis of the existing notions of subtyping in functional programming languages and session-based programming

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 280–296, 2011.

languages. First we introduce type-directed embeddings of two known subtyped calculi, one for functions [22] (which uses Milner's encoding [16] but with novelty in the treatment of sums) and another for session-based communications [25,13]. In both cases, the behavioural content of the original subtyping is precisely captured in the fine-grained subtyping theory in the pi-calculus.

We then establish full abstraction of each embedding with respect to a standard semantics of the target calculus, Morris's contextual congruence in the case of the functional calculus and the testing/failure equivalence for the concurrent calculus. The full abstraction, together with the concision of the encoding, may offer an exact interactional elucidation of these existing subtyping notions. For the full abstraction of the embedding of the session-based calculus, we introduce a new proof method centring on non-deterministic computational adequacy and definability. Partially suggested by a technique used by Quaglia and Walker for their full abstraction of the polyadic synchronous π-calculus in the monadic asynchronous π-calculus [23], as well as by those from game-based semantics [15], the new proof method is uniform (the method for non-deterministic languages specialises the one for the traditional, deterministic languages), has generality (the type structure of a meta calculus, here the linear subtyped π-calculus, can be disjoint from that of an object language, here the λ-calculus and the session calculus), and is generic (as far as some key properties hold for adequacy and definability, it automatically gives full abstraction).

We summarise some of the main technical contributions of the work.

1. A concise subtyped π-calculus with linear typing ($\pi^{\{1,\overline{1},\omega\}}$), giving rise to a simple and general theory of subtyping, whose key properties we establish.
2. Type-directed embeddings of a call-by-value λ-calculus with record and variant subtyping and a concurrent calculus with session subtyping in $\pi^{\{1,\overline{1},\omega\}}$, obtaining full abstractions. For the latter we use a new proof method centring on non-deterministic computational adequacy and definability.

To our knowledge, this is the first full abstraction results for these subtyped calculi in the π-calculus: further, the corresponding results have not been known in game-based semantics (which is in close corresponding with the π-calculus, cf. [12,14,9]). The semantically sound encoding of the session calculus itself looks new. In another vein, this may be the first full abstraction result for the interactional representation of a non-trivial, fully non-deterministic concurrent calculus.

In the rest of the paper, Section 2 introduces $\pi^{\{1,\overline{1},\omega\}}$ and develops the theory of subtyping, illustrating its intuition through examples and establishing its key properties. Section 3 fully abstractly embeds $\lambda^{\Pi,\Sigma,\sqsubseteq}$ in $\pi^{\{1,\overline{1},\omega\}}$. Section 4 fully abstractly embeds the session-calculus from [13] with subtyping in $\pi^{\{1,\overline{1},\omega\}}$. Section 5 discusses related works. The full proofs can be found in [7].

2 A Concise, Subtyped π-Calculus

In the following, we use the shortcut $\widetilde{e}$ for a vector $(e_1, \ldots, e_k)$ for some integer k.

Processes and Reduction. We use $a, b, c, \ldots, u, v, \ldots, x, y$ to denote names, or channels, $X, Y, \ldots$ for agent variables, and $l, \ldots$ for labels. Syntax for processes of $\pi^{\{1,\overline{1},\omega\}}$, our linear-affine π-calculus (cf. [3,28]), is given by the following grammar.

$$P ::= (P \mid P) \mid \mathbf{0} \mid X\langle\widetilde{v}\rangle \mid (\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle \mid \overline{u} \oplus^m l\langle\widetilde{v}\rangle \mid u\&^m_{i\in I}\{l_i(\widetilde{x_i}).P_i\} \mid (\nu u)\ P$$

where $m ::= 1 \mid \overline{1} \mid \omega$ is called a *mode*. The mode can be either linear 1, affine $\overline{1}$ or replicated ω. We use a standard recursion $(\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle$. The two prefixes of our calculus are the asynchronous output (or *selection*) $\overline{u} \oplus^m l\langle\widetilde{v}\rangle$ which is the output of the values $\widetilde{v}$ on the channel u as well as selecting the label l, and the input (or *choice*) $u\&^m_{i\in I}\{l_i(\widetilde{x_i}).P_i\}$ which offers on channel u several branches to choose from, labelled by the l_is, each with continuation P_i. We use a standard structural congruence $\equiv$ on $\pi^{\{1,\overline{1},\omega\}}$, described in Figure 1.

$$P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \mid \mathbf{0} \equiv P$$

$$(\nu a)(\nu b)\ P \equiv (\nu b)(\nu a)\ P \qquad (\nu a)\ (P_1 \mid P_2) \equiv ((\nu a)\ P_1) \mid P_2 \text{ if } a \text{ not free in } P_2$$

$$(\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle \equiv P\{\widetilde{v}/\widetilde{x}\}\{\mu X(\widetilde{x}).P/X\}$$

Fig. 1. Structural congruence rules

$$(\textbf{cong})\frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

$$(\textbf{comm})\frac{}{\mathbf{E}[\overline{u} \oplus^{1,\overline{1}} l_j\langle\widetilde{v}\rangle \mid u\&^{1,\overline{1}}_{i\in I}\{l_i(\widetilde{x_i}).P_i\}] \to \mathbf{E}[P_j\{\widetilde{v}/\widetilde{x_i}\}]}$$

$$(\textbf{trig})\frac{}{\mathbf{E}[\overline{u} \oplus^{\omega} l_j\langle\widetilde{v}\rangle \mid u\&^{\omega}_{i\in I}\{l_i(\widetilde{x_i}).P_i\}] \to \mathbf{E}[P_j\{\widetilde{v}/\widetilde{x_i}\} \mid u\&^{\omega}_{i\in I}\{l\}_i(\widetilde{x_i}).P_i]}$$

Fig. 2. Reduction rules

The rules to generate the reduction relation $\to$ are given in Figure 2, using *evaluation contexts* (*à la Wright-Felleisen*) given by: $\mathbf{E} ::= [\,] \mid \mathbf{E} \mid P \mid (\nu u)\ \mathbf{E}$. Henceforth $\to^+$ (resp. $\to^*$) denotes the transitive (resp. reflexive-transitive) closure of $\to$. We also use $\not\to$ to notify that a process cannot reduce further.

Types. Types consist of base types (integers, booleans and unit), the choice and selection types (together called *interaction types*), and recursive types. The choice types can be seen as a generalisation of input types (as used in [3]); the selection types as a generalisation of output types. Their branching structure plays a key role in our subtyping theory. The syntax for types is given by the following grammar:

$$T \ ::= \ \&^m_{i\in I}\{l_i(\widetilde{T_i})\} \ \mid \ \oplus^m_{i\in I}\{l_i(\widetilde{T_i})\} \ \mid \ \mu t.T \ \mid \ t \ \mid \ \mathtt{uc} \ \mid \ \mathtt{N} \ \mid \ \mathtt{B} \ \mid \ \bigstar$$

where $\mathtt{N}$, $\mathtt{B}$ and $\bigstar$ are the types for integers, booleans and the unit, respectively. We assume recursive types are contractive (type variables occur guarded) [20]. *Closed types* are types without free type variables. $\mathtt{uc}$, never occurring in another type, means a pair of dual linear-affine channels are present and is now"uncomposable". Types are considered up to the standard tree isomorphisms.

If T has form $\&^m_{i\in I}\{l_i(\widetilde{T_i})\}$ (resp. $\oplus^m_{i\in I}\{l_i(\widetilde{T_i})\}$), the *mode* of T, $\mathtt{mod}(T)$, is m.

For brevity we shall often use shortcuts for prefixes and types when their branching is reduced to a single branch. Below we set $x_1 = x$, $P_1 = P$ and $l_1 = l^{\mathsf{one}}$ (l^{one} is a distinguished label we fix).

$$u(x).P = u\&^1_{i\in\{1\}}\{l_i(x_i).P_i\} \qquad\qquad !u(x).P = u\&^\omega_{i\in\{1\}}\{l_i(x_i).P\}$$

$$\overline{u}\langle v\rangle = u \oplus^m l^{\mathsf{one}}\langle v\rangle \text{ when } u \text{ has type } \oplus^m_{i\in\{1\}}\ \{l_i(T_i)\}$$

$$\uparrow^m (T) = \oplus^m_{i\in\{1\}}\{l_i(T)\} \qquad\qquad \downarrow^m (T) = \&^m_{i\in\{1\}}\{l_i(T)\}$$

Example 1 (Intuitive meaning of types)

1. $\uparrow^1 (\downarrow^1 (\mathtt{N}))$ indicates a behaviour at an output channel, through which a process surely sends a channel exactly once; and through that channel, surely receiving a natural number (considered to be a constant channel) exactly once.
2. $\&^1_{i\in\{1,2\}}\{l_1(\mathtt{N}), l_2(\uparrow^1 (\downarrow^1 (\mathtt{N})))\}$ indicates a behaviour of exactly once receiving one of the two options, l_1 and l_2, in the former with an integer, in the latter with a channel which the process will use as specified in 1 above.
3. $\&^{\overline{1}}_{i\in\{1,2\}}\{l_1(\mathtt{N}), l_2(\uparrow^1 (\downarrow^1 (\mathtt{N})))\}$ is the same behaviour as 2 above, except it receives an initial option at most once, by the modality $\overline{1}$.

The interaction types form a self-contained universe in that base types can be considered as syntactic sugar, through encodings. There are several faithful (and semantically isomorphic) encodings, of which we present a simple and convenient one. We write T°, if T is a base type, for the encoding of T as an interaction type.

$$\mathtt{N}^\circ \stackrel{\text{def}}{=} \downarrow^\omega (\oplus^1_{i\in\mathtt{N}}\{\mathbf{i}()\}) \qquad \mathtt{B}^\circ \stackrel{\text{def}}{=} \downarrow^\omega (\oplus^1_{i\in\mathtt{B}}\{\mathbf{i}()\}) \qquad \bigstar^\circ \stackrel{\text{def}}{=} \downarrow^\omega (\oplus^1_{i\in\bigstar}\{\mathbf{i}()\})$$

where we use the labels which represent natural numbers, booleans and the single element of the unit. Each describes a behaviour which can be enquired about its content and responds with one. We then extend $(\ )^\circ$ so that when a base type is used for output, we use the above encoding, e.g. $(\uparrow (\mathtt{B}))^\circ = \uparrow (\mathtt{B}^\circ)$, and for input, its dual, e.g. $(\downarrow (\mathtt{B}))^\circ = \downarrow (\overline{\mathtt{B}^\circ})$, where $\overline{T}$ is defined below.

A key idea in interaction types is duality, defined co-inductively [21] to capture recursion. Note by taking recursive types modulo their tree isomorphism, we can safely regard each closed type as either a base, choice or selection type.

Definition 2. *A relation over closed types $\mathcal{R}$ is* duality *if $T_1\ \mathcal{R}\ T_2$ implies:*

1. *either $T_1 = T_2$ where $T_1 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$,*
2. *or (a) $T_1 = \&^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, (b) $T_2 = \oplus^m_{i\in I}\{l_i(\widetilde{T^2_i})\}$ and (c) $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$*
3. *or (a) $T_1 = \oplus^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$ (b) $T_2 = \&^m_{i\in I}\{l_i(\widetilde{T^2_i})\}$ and (c) $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$*

There is the largest duality relation denoted $\bowtie$*.*

When we encode away base types through $(\ )^\circ$, we dispense with the first clause.

Fact and Definition 3 (Dualisation). *The duality* $\bowtie$ *defines a total involution (i.e. a symmetric total function), which we write* $\overline{T}$*, called the* dual *of* T*.*

A generalisation of duality is *coherence*, which we introduce below. Intuitively, coherence specifies when two types can match: this is not only when two types are dual, but also when one offers more choices than the options the other wishes to select. This intuition is the basis of the whole subtyping theory.

Definition 4. *A relation* $\mathcal{R}$ *over closed types is a coherence if* $T_1\ \mathcal{R}\ T_2$ *implies:*

1. *either* $T_1 = T_2$ *where* $T_1 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$*.*
2. *or (a)* $T_1 = \&^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$*, (b)* $T_2 = \oplus^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$*, and (c)* $\forall j \in J, T^1_j\ \mathcal{R}\ T^2_j$ *where* $J \subseteq I$*.*
3. *or (a)* $T_1 = \oplus^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$*, (b)* $T_2 = \&^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$*, and (c)* $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$ *where* $I \subseteq J$

There is the largest coherence relation noted $\asymp$*.*

Note $\bowtie\ \subsetneq\ \asymp$. Non-trivial inclusion among base types can be incorporated into coherence: however how we can do so is already in $\asymp$, through the encoding discussed above. We shall come back to this point later.

Typing. The typing rules for $\pi^{\{1,\overline{1},\omega\}}$ use typing contexts and compatibility over them, which we introduce below.

A *typing context* Γ is a partial map from names to types and from process variable to vectors of types. When $\Gamma(a)$ is undefined, we write $\Gamma, a{:}T$ to denote the map Γ' defined by $\Gamma'(a) = T$, $\Gamma'(u) = \Gamma(u)$ for $u \neq a$ and $\Gamma'(Y) = \Gamma(Y)$ for all Y. $\Gamma, X{:}\widetilde{T}$ is defined the same way. We write $\emptyset$ for the empty typing context.

Definition 5. *We define* compatibility $\odot$ *as a partial symmetric function over pair of typing contexts generated from:*

$$
\begin{array}{ll}
\emptyset \odot \emptyset = \emptyset & \\
(\Gamma_1, X : \widetilde{T}) \odot \Gamma_2 = \Gamma_1 \odot \Gamma_2, X : \widetilde{T} & (\Gamma_1, u : \mathsf{uc}) \odot \Gamma_2 = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} & \textit{if } T_1 \asymp T_2 \textit{ and } \forall i, \mathtt{mod}(T_i) = 1 \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} & \textit{if } T_1 \asymp T_2 \textit{ and } \forall i, \mathtt{mod}(T_i) = \overline{1} \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : T_1 & \textit{if } T_1 \asymp T_2, \mathtt{mod}(T_1) = \mathtt{mod}(T_2) = \omega \\
 & \quad \textit{and} \ \ T_1 = \&^\omega_{i\in I}\{l_i(\widetilde{T_i})\}
\end{array}
$$

Compatibility stipulates, through its partiality, when a parallel composition of two typed processes is allowed. Typing rules are presented in Figure 3.

In (**Cho**), we assume no 1 types occur in Γ when $m = 1$; and no 1 and $\overline{1}$ types occur when $m = \omega$. We observe:

$$(\mathbf{Nil})\frac{}{\emptyset \vdash_\pi \mathbf{0}} \qquad (\mathbf{Rec})\frac{\mathit{\Gamma}, X : \widetilde{T}, \widetilde{x} : \widetilde{T} \vdash_\pi P \qquad \mathit{\Gamma}(\widetilde{v}) = \widetilde{T}}{\mathit{\Gamma} \vdash_\pi (\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle}$$

$$(\mathbf{Res})\frac{\mathit{\Gamma}, u : T \vdash_\pi P \qquad T = \mathsf{uc} \text{ or } \&^{\omega}_{i\in I}\{l_i(T_i)\}}{\mathit{\Gamma} \vdash_\pi (\nu u)\ P} \qquad (\mathbf{Par})\frac{\mathit{\Gamma}_1 \vdash_\pi P_1 \qquad \mathit{\Gamma}_2 \vdash_\pi P_2}{\mathit{\Gamma}_1 \odot \mathit{\Gamma}_2 \vdash_\pi P_1 \mid P_2}$$

$$(\mathbf{Var})\frac{}{X : \widetilde{T}, \widetilde{v} : \widetilde{T} \vdash_\pi X\langle\widetilde{v}\rangle} \qquad (\mathbf{Sel})\frac{j \in I}{u : \oplus^m_{i\in I}\{l_i(\overline{\widetilde{T_i}})\}, \widetilde{v_j} : \widetilde{T_j}, \mathit{\Gamma} \vdash_\pi \overline{u} \oplus^m l_j\langle\widetilde{v_j}\rangle}$$

$$(\mathbf{Cho})\frac{(\mathit{\Gamma}, \widetilde{x_i} : \widetilde{T_i} \vdash_\pi P_i)_{i\in I}}{\mathit{\Gamma}, u : \&^m_{i\in I}\{l_i(\widetilde{T_i})\} \vdash_\pi u\&^m_{i\in I}\{l_i(\widetilde{x_i}) : P_i\}}$$

Fig. 3. Typing rules for $\pi^{\{1,\overline{1},\omega\}}$

Proposition 6 (Subject Reduction). *If $\mathit{\Gamma} \vdash_\pi P$ and $P \to P'$, then $\mathit{\Gamma} \vdash_\pi P'$.*

Example 7. *As an example of the expressive power, we present an easy way to encode references (or* states*):*

$$\mathbf{Mem} = !ref(cell, val).cell\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r, new). & (\overline{r}\langle\,()\rangle \mid \overline{ref}\langle cell, new\rangle) \\ \mathtt{get}(s). & (\overline{s}\langle val\rangle \mid \overline{ref}\langle cell, val\rangle)\end{matrix}\right\}$$

Here **Mem** *is a server that create memory cells. Clients can interact with a cell either to fetch the value store inside or to update it with a new value. Consider:*

$$\mathbf{E}_1 = \mathbf{Mem} \mid \overline{ref}\langle cell_1, 0\rangle \mid \overline{ref}\langle cell_2, 3\rangle$$

which reduces in two steps to:

$$\mathbf{Mem} \mid cell_1\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r_1, new_1). & (\overline{r_1}\langle\,()\rangle \mid \overline{ref}\langle cell_1, new_1\rangle) \\ \mathtt{get}(s_1). & (\overline{s_1}\langle 0\rangle \mid \overline{ref}\langle cell_1, 0\rangle)\end{matrix}\right\}$$

$$\mid cell_2\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r_2, new_2). & (\overline{r_2}\langle\,()\rangle \mid \overline{ref}\langle cell_2, new_2\rangle) \\ \mathtt{get}(s_2). & (\overline{s_2}\langle 3\rangle \mid \overline{ref}\langle cell_2, 3\rangle)\end{matrix}\right\}$$

creating two memory cells, one called $cell_1$ containing 0 *and one called $cell_2$ containing* 3*. Each cell offers an input with two labels,* `set` *and* `get`*. The former waits for a return channel r, returns the current value through r and construct the cell again, the latter waits for a return channel r and a new value new, reconstructs the updated cell with the new value and return an acknowledgement through r. The process* **Mem** *can be typed by giving to ref the type:*

$$T_{ref} = \downarrow^{\omega} (\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(\uparrow^{\overline{1}} (\bigstar), \mathtt{N}) \\ \mathtt{get}(\uparrow^{\overline{1}} (\mathtt{N}))\end{matrix}\right\}, \mathtt{N})$$

As another example showing concurrency, consider:

$$\mathbf{Ex} = (\nu ans_1, ans_2)\ (\mathbf{E}_1 \mid \overline{cell_1}\mathtt{set}\langle 1, ans_1\rangle \mid ans_1() \mid \overline{cell_1}\mathtt{get}\langle ans_2\rangle \mid ans_2(x)$$

When reducing **Ex***, x can be instantiated either by* 1 *or* 0 *depending on which output on* $cell_1$ *occurs first.*

Subtyping theory. As hinted in Definition 4, the framework of $\pi^{\{1,\overline{1},\omega\}}$ lets us define a notion of subtyping: informally, a type will be a subtype of another if it "offers more choices" (as input) or if it "selects among less options" (as an output). Intuitively, a type which is ready to receive no less labels, and which may potentially sends no more labels, (in other words, a type representing a more gentle behaviour), is a subtype of another.

Definition 8. *A relation $\mathcal{R}$ is a* subtyping relation *if when $T_1 \mathcal{R} T_2$ then:*

1. *either* $T_1 = T_2 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$,
2. *or (a)* $T_1 = \&^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, *(b)* $T_2 = \&^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$ *and (c)* $\forall j \in J, T^1_j \mathcal{R} T^2_j$, *where* $J \subseteq$.
3. *or (a)* $T_1 = \oplus^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, *(b)* $T_2 = \oplus^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$, *and (c)* $\forall i \in I, T^1_i \mathcal{R} T^2_i$, *where* $I \subseteq J$.

$\sqsubseteq$ *is the largest (for* $\subseteq$*) subtyping relation.*

The subsumption is admissible in our typing system. Below we write $\Gamma \sqsubseteq \Gamma'$ to denote that the two typing environments have same domain and that for each a (resp. X) in the domain of Γ, $\Gamma(a) \sqsubseteq \Gamma'(a)$ (resp. $\Gamma(X) \sqsubseteq \Gamma'(X)$).

Proposition 9 (Subsumption). *If* $\Gamma \vdash_\pi P$ *and* $\Gamma \sqsubseteq \Gamma'$ *then* $\Gamma' \vdash_\pi P$.

That is, if P satisfies Γ and if Γ' is more inclusive as a specification then P also satisfies Γ' (noting Γ in $\Gamma \vdash_\pi P$ specifies the behaviour of P, cf. Example 1).

Now it is known in the literature that subtyping is closely related to composability of types, cf. [11], which we may call *compatibility.* In brief, T_2 *has more compatibility than* T_1 if it is coherent with every type with which T_1 is coherent: it is more composable. In the following we show compatibility and subtyping coincide in our theory, showing its consistency as well as giving useful theoretical tools.

Definition 10. *For* T_1 *and* T_2 *closed,* $T_1 \leq^{\mathtt{comp}} T_2$ *when* $\forall T, T_1 \asymp T \Rightarrow T_2 \asymp T$.

Below Propositions 11 relates together duality, subtyping and coherence, using which we show coincidence, Proposition 12.

Proposition 11. *(1)* $T \asymp \overline{T}$. *(2)* $T_1 \asymp T_2$ *iff* $T_2 \asymp T_1$. *(3)* $T_1 \sqsubseteq T_2$ *iff* $\overline{T_2} \sqsubseteq \overline{T_1}$. *(4) If* $T'_1 \sqsubseteq T_1$, $T'_1 \asymp T'_2$, $T'_2 \sqsubseteq T_2$ *then* $T_1 \asymp T_2$. *(5)* $T_1 \sqsubseteq T_2$ *iff* $T_1 \asymp \overline{T_2}$

Proposition 12 (Coincidence). $T_1 \sqsubseteq T_2$ *iff* $T_2 \leq^{\mathtt{comp}} T_1$.

Proof (Sketch). First we show $\leq^{\mathtt{comp}}$ is a subtyping relation coinductively by inspecting the shape of T_2 and Def. 4 gives information on T_1. Then we prove $T \asymp T_1$ implies $T \asymp T_2$ by the form of T_2, using Definition 8. □

Example 13. *To illustrate our subtyping, recall* T_{ref} *from section 2 inhabited by* **Mem***. An obvious subtype of* T_{ref} *is the following* T'_{ref}*:*

$$T'_{ref} = \downarrow^{\omega} (\&^{\overline{1}}_{l \in \{\mathtt{set},\mathtt{get},\mathtt{del}\}} \left\{ \begin{array}{l} \mathtt{set}(\uparrow^{\overline{1}}(\bigstar), \mathtt{N}) \\ \mathtt{get}(\uparrow^{\overline{1}}(\mathtt{N})) \\ \mathtt{del}(\uparrow^{\overline{1}}(\bigstar)) \end{array} \right\}, \mathtt{N})$$

Since this is a subtype of T_{ref}, there are some processes which live (are typed by) $ref : T_{ref}$ as well as by $ref : T'_{ref}$. The following is such a process:

$$\mathbf{Mem'} =$$

$$!ref'(cell, val).cell\&^{\overline{1}}_{l \in \{\mathtt{set},\mathtt{get},\mathtt{del}\}} \left\{ \begin{array}{ll} \mathtt{set}(r, new). & (\overline{r}\langle\, ()\rangle \mid \overline{ref'}\langle cell, new\rangle) \\ \mathtt{get}(s). & (\overline{s}\langle val\rangle \mid \overline{ref'}\langle cell, val\rangle) \\ \mathtt{del}(t). & (\overline{t}\langle\, ()\rangle) \end{array} \right\}$$

The process **Mem′** *performs the same role as* **Mem** *but offer one additional choice, the label* `del` *allows one to delete a memory cell, preventing further interactions with this cell to be performed. One can notice that in every process containing* **Mem** *under $ref : T_{ref}$, it can be replaced with* **Mem′***: as far as the supertype T_{ref} goes, they have the same behaviour.*

3 Embedding Functional Subtyping

A subtyped call-by-value functional calculus. For brevity we consider a typed, PCF-like, call-by-value λ-calculus with $\bigstar$ as its base type, which we call $\lambda^{\Pi,\Sigma,\sqsubseteq}$. The syntax of terms contains both products (or *records*) $\{l_i.M_i\}_{i\in I}$ and projections, and sums (or *variants*) and case-branches, and is given, along with the syntax of types, by the following grammar:

$$M ::= M\ M \mid x \mid () \mid \lambda x.M \mid \{l_i.M_i\}_{i\in I} \mid M.l$$

$$\mid \quad \mathtt{inj}_l(M) \mid \mathtt{case}\ M\ \mathtt{of}\ [l_i(x_i).M_i]_{i\in I} \mid \mathtt{Y}\ V$$

$$T ::= \bigstar \mid \mathtt{N} \mid T \to T \mid \Pi\{l_i : T_i\}_{i\in I} \mid \Sigma[l_i : T_i]_{i\in I}$$

The subtyping relation $T_1 \sqsubseteq T_2$ on $\lambda^{\Pi,\Sigma,\sqsubseteq}$ types is defined as the largest reflexive and transitive relation satisfying:

1. If $T_1 = T_a \to T_b$, then $T_2 = T'_a \to T'_b$ and $T_a \sqsubseteq T'_a$, $T'_b \sqsubseteq T_b$.
2. If $T_1 = \Pi\{l_i : T_i\}_{i\in I}$, then $T_2 = \Pi\{l_i : T'_i\}_{i\in J}$, $J \subseteq I$ and $\forall i \in J, T_i \sqsubseteq T'_i$.
3. If $T_1 = \Sigma[l_i : T_i]_{i\in I}$, then $T_2 = \Sigma[l_j : T'_j]_{j\in J}$, $I \subseteq J$ and $\forall i \in I, T_i \sqsubseteq T'_i$.

The typing system and the call-by-value reduction rules are completely standard: for reference, below we only list the key rule for subtyping, the subsumption, and leave the rest in [7].

$$(\mathbf{Sub})\frac{\Gamma \vdash M : T_1 \qquad T_1 \sqsubseteq T_2}{\Gamma \vdash M : T_2}$$

Encoding of types. We present the encoding of $\lambda^{\Pi,\Sigma,\sqsubseteq}$ into our calculus $\pi^{\{1,\overline{1},\omega\}}$. Definition 14 presents how we encode $\lambda^{\Pi,\Sigma,\sqsubseteq}$ types into $\pi^{\{1,\overline{1},\omega\}}$ types. A notable point of this encoding is the encoding of the *arrow* type, which is decomposed in such a way that function application is seen as a choice between the possible arguments. As a result, the type $\mathtt{B} \to T$, for instance, is encoded into $\uparrow^{\overline{1}} (\&^{\omega}_{l\in\{\mathtt{true},\mathtt{false}\}}\{\mathtt{true}(T);\mathtt{false}(T)\})$. This means that a function having booleans as domain can be seen as being composed of two terms, one associated to the argument `true`, the other for `false`.

Definition 14 (Encoding of types)

$$\langle\bigstar\rangle = \uparrow^{\overline{1}} (\bigstar) \qquad\qquad \langle \Pi\{l_i : T_i\}_{i\in I}\rangle = \uparrow^{\overline{1}} (\&^{\omega}_{i\in I}\{l_i(\langle T_i\rangle)\})$$

$$\langle \Sigma[l_i : T_i]_{i\in I}\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(\downarrow^{\omega} (\langle T_i\rangle))\}$$

$$\langle T_1 \to T_2\rangle = \uparrow^{\overline{1}} (\&^{\omega}_{i\in I}\{l_i(\overline{T_i}, \langle T_2\rangle)\}) \quad \textit{if } \langle T_1\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(T_i)\}$$

The last line is well-defined since $\langle T_1\rangle$ is always an output type. In the first line, we can encode $\bigstar$ (of $\pi^{\{1,\overline{1},\omega\}}$) by $(\ )^{\circ}$ in Section 2, similarly any base types.

Encoding of terms. Figure 4 gives the encoding of terms following that of types and using a return channel u and an environment ζ (required to remember encoding of variables). The former is standard [24]. The latter may be notable, coming from our arrow type encoding which forces us to remember the association between a branch label and a variable. An environment ζ maps $\lambda^{\Pi,\Sigma,\sqsubseteq}$ each variable to a *choice* (l_i, x_i), a pair composed of one label and one $\pi^{\{1,\overline{1},\omega\}}$ variable.

A brief illustration of three key cases: when encoding an application $M\ N$ on u, one compute the encoding on M and N with two new return channels (respectively m and n), then the address of the function y is caught on m and the possible arguments are decomposed into an address x_i and a label l_i, fetched on n. Then both of them are sent to the function together with the return channel u. Symmetrically, to encode the abstraction $\lambda x.M$, we create a new channel c, send it on the return channel of the function, then we wait for a label l_i which will determine which branch of the function is chosen, an argument x_i and a return channel m, and we proceed to the execution of the encoding of M, with a new environment where the variable x is associated with the choice (l_i, x_i). Finally, to encode a variable x on the return channel u, we fetch in the environment the choice (l_i, x_i) associated with x and send it on the channel u.

As our encoding makes use of environment, we formally define the encoding for typing contexts accordingly.

$$\langle \Gamma, x : T\rangle^{\zeta, x\mapsto(l_i,x_i)} = \langle\Gamma\rangle^{\zeta}, x_i : T_i \text{ if } \langle T\rangle = \oplus^{m}_{i\in I}\{l_i(T_i)\}$$

A variable environment ζ is *reasonable w.r.t. a term* M when ζ maps every free variable of M to a choice.

Proposition 15. *If* $\Gamma \vdash M : T$, *then* $\langle\Gamma\rangle^{\zeta}, u : \langle T\rangle \vdash_{\pi} \langle M\rangle^{\zeta}_{u}$ *for all reasonable* ζ.

$$\langle ()\rangle_u^\zeta = \overline{u}\langle ()\rangle$$

$$\begin{array}{rl}\langle M\ N\rangle_u^\zeta = & (\nu m, n)\ (\langle M\rangle_m^\zeta \mid \langle N\rangle_n^\zeta \mid m(y).n\&_{i\in I}^{\overline{1}}\{l_i(x_i).y \oplus^\omega l_i\langle x_i, u\rangle\}) \\ & \text{if } M \text{ has type } T \rightarrow T' \text{ with } \langle T\rangle = \oplus_{i\in I}^{\overline{1}}\{l_i(T_i)\} \\ \langle \lambda x.M\rangle_u^\zeta = & (\nu c)\ (\overline{u}\langle c\rangle.c\&_{i\in I}^{\omega}\{l_i(x_i, m).\langle M\rangle_m^{\zeta, x\mapsto(l_i,x_i)}\}) \\ & \text{if } \lambda x.M \text{ has type } T \rightarrow T' \text{ with } \langle T\rangle = \oplus_{i\in I}^{\overline{1}}\{l_i(T_i)\}\end{array}$$

$$\langle x\rangle_u^{\zeta.x\mapsto(l_i,x_i)} = \overline{u} \oplus^{\overline{1}} l_i\langle x_i\rangle \qquad \langle \{l_i : M_i\}_{i\in I}\rangle_u^\zeta = (\nu p)\ \overline{u}\langle p\rangle.p\&_{i\in I}^{\omega}\{l_i().\langle M_i\rangle_u^\zeta\}$$

$$\langle M.l\rangle_u^\zeta = (\nu m)\ (\langle M\rangle_m^\zeta \mid m(y).\overline{y} \oplus^\omega l\langle\rangle)$$

$$\langle \texttt{inj}_l(M)\rangle_u^\zeta = (\nu m)\ (\langle M\rangle_m^\zeta \mid m\&_{j\in J}^{\overline{1}}\{l'_j(y_j).(\nu c)\ (\overline{u} \oplus^{\overline{1}} l\langle c\rangle \mid !c(n).\overline{n} \oplus^\omega l'_j\langle y_j\rangle)\})$$

$$\langle \texttt{case}\ M\ \texttt{of}\ [l_i(x_i).M_i]_{i\in I}\rangle_u^\zeta =$$

$$(\nu m)\ (\langle M\rangle_m^\zeta \mid m\&_{\imath\in I}^{\overline{1}}\{l_i(x_i).(\nu p)\ (\overline{x_i}\langle p\rangle \mid p\&_{j\in J}^{\omega}\{l'_j(y_j).\langle M_i\rangle_u^{\zeta, x_i\mapsto(l'_j,y_j)}\})\})$$

$$\langle \texttt{Y}\ V\rangle_u^\zeta = \mu X(x).((\nu p, m)\ \langle V\rangle_p^\zeta \mid (X\langle m\rangle) \mid m(f).p(a).\overline{f}\langle a, x\rangle)\langle u\rangle$$

Fig. 4. Encoding for λ-terms

Example 16. *Consider the following $\lambda^{\Pi,\Sigma,\sqsubseteq}$ term:*

$$F^{opt} = \lambda x.\texttt{case}\ x\ \texttt{of}\ \left[\begin{matrix}\texttt{s}(x_1) & (F\ x_1) \\ \texttt{n}(x_2) & 0\end{matrix}\right]_{l\in\{\texttt{s},\texttt{n}\}}$$

$F^{opt} : (\texttt{s} : \texttt{N}) + (\texttt{n} : ())\rightarrow \texttt{N}$ is a partial version of the function $F : \texttt{N} \rightarrow \texttt{N}$. If its argument is an actual value $\texttt{inj}_{\texttt{s}}(3)$ it will apply F to it. And if its argument is undefined $\texttt{inj}_{\texttt{n}}(())$ then it will return 0. With $\langle \texttt{N}\rangle =\uparrow^{\overline{1}} (\texttt{N})$ and $\langle \bigstar\rangle =\uparrow^{\overline{1}} (\bigstar)$, its encoding $\langle F^{opt}\rangle_u^\emptyset$ is given by:

$$(\nu v)\ \overline{u}\langle v\rangle.v\&_{l\in\{\texttt{s},\texttt{n}\}}^{\omega}\left\{\begin{matrix}\texttt{s}(x, b).\ (\nu c)\ (\overline{c} \oplus^{\overline{1}} \texttt{s}\langle x\rangle \mid \textbf{Caseof}) \\ \texttt{n}(x, b).\ (\nu c)\ (\overline{c} \oplus^{\overline{1}} \texttt{n}\langle x\rangle \mid \textbf{Caseof})\end{matrix}\right\}$$

$$\textbf{Caseof}\ \textit{being}\ c\&_{l\in\{\texttt{s},\texttt{n}\}}^{\overline{1}}\left\{\begin{matrix}\texttt{s}(x_1).\ (\nu p_1)\ (\overline{x_1}\langle p_1\rangle \mid !p_1(n).\textbf{App}) \\ \texttt{n}(x_2)\ \ (\nu p_2)\ (\overline{x_1}\langle p_2\rangle \mid !p_2.\overline{b}\langle 0\rangle)\end{matrix}\right\}$$

$$\textbf{App}\ \textit{being}\ (\nu f, a)\ (\langle F\rangle_f^\emptyset \mid \overline{a}\langle n\rangle \mid f(y).a(z).\overline{y}\langle z, b\rangle)$$

If we look through the several indirections induced by the encoding, we can notice that the choice induced by the option type will be translated as two choices, one in the abstraction encoding and one in the case-construct encoding.

Functional subtyping through encoding. The following proposition relates the subtyping for $\lambda^{\Pi,\Sigma,\sqsubseteq}$ with the subtyping for $\pi^{\{1,\overline{1},\omega\}}$. It is proved by showing that the relation $\mathcal{R}$, defined by $T_a\mathcal{R}T_b$ when there exist T_1, T_2 s.t. $\langle T_1\rangle = T_a$, $\langle T_2\rangle = T_b$, and $T_1\ \mathcal{R}\ T_2$, is a subtyping according to Definition 8.

Proposition 17 *If $T_1 \sqsubseteq T_2$, then $\langle T_1 \rangle \sqsubseteq \langle T_2 \rangle$.*

The base type encoding in $\pi^{\{1,\overline{1},\omega\}}$ through the operator $(\ )^\circ$ in section 2 allows us to extend the above result to $\lambda^{\Pi,\Sigma,\sqsubseteq}$ with non-trivial subtyping on base types, e.g. with the type $\mathtt{R}$ of reals. Indeed, for $\mathtt{N} \sqsubseteq \mathtt{R}$, we get $\langle \mathtt{N} \rangle^\circ \ \sqsubseteq\ \langle \mathtt{R} \rangle^\circ$, by $\uparrow^{\overline{1}} (\downarrow^{\omega} (\oplus^{1}_{i\in\mathbb{N}}\{\mathbf{i}()\})) \ \sqsubseteq\ \uparrow^{\overline{1}} (\downarrow^{\omega} (\oplus^{1}_{i\in\mathbb{R}}\{\mathbf{i}()\}))$.

Full abstraction. In order to obtain a full abstraction result we restrict our π-calculus, imposing *sequentiality* to typed processes by controlling the number of activities compositionally as in [3]. We derive a definability result [7], that is, every sequential process typable with the encoding a λ-typing context is equivalent to the encoding of a term typable with that context. Using as the equivalence on the functional side $\simeq_\lambda$ Morris congruence [17] and as the equivalence of the concurrent side $\simeq_\pi$ the standard reduction-closed barbed congruence [24], we obtain:

Theorem 18. *Let M_1, M_2 be two $\lambda^{\Pi,\Sigma,\sqsubseteq}$ terms, then $M_1 \simeq_\lambda M_2$ if and only if $\langle M_1 \rangle^{\emptyset}_{u} \simeq_\pi \langle M_1 \rangle^{\emptyset}_{u}$ under sequential typing.*

4 Embedding Communication Subtyping

This section is dedicated to the study of the encoding of a session-based concurrent calculus $\pi^{\mathtt{session}}$ with synchronous interactions based on session types. Session types abstract protocols of communicating processes as types, and ensure their sound communication behaviour through the associated type discipline, see [8] for a survey.

Our presentation of $\pi^{\mathtt{session}}$ follows [13]. In this language, names are divided into channels $u, v, \ldots$ and sessions $s, k, \ldots$; we use $a, b, c, \ldots$ to denote names of any kind. Syntax for terms is given by the following grammar:

$$P \ ::= \ u(x).P \ \mid \ \overline{u}(x).P \ \mid \ k!l\langle v\rangle.P \ \mid \ k?\{l_i(x_i).P_i\}_{i\in I} \ \mid \ \text{if } e \text{ then } P \text{ else } P$$

$$\mid \ \ (\nu a)P \ \mid \ P|P \ \mid \ (\mu X(\widetilde{x}).P)\langle \widetilde{v}\rangle \ \mid \ X\langle \widetilde{v}\rangle \ \mid \ \mathbf{0}$$

The definition of evaluation contexts is straightforward. For brevity, we do not include delegation, though its encoding follows that of shared name passing. The semantics is generated from the following two base rules:

$$\frac{}{\mathbf{E}[u(x_1).P_1 \mid \overline{u}(x_2).P_2] \rightarrow \mathbf{E}[(\nu s)\ (P_1\{s/x_1\} \mid P_2\{s/x_2\})]}$$

$$\frac{}{\mathbf{E}[s?\{(x_i).P_i\}_{i\in I} \mid s!l_j\langle v\rangle.P] \rightarrow \mathbf{E}[P_j\{v/x_j\} \mid P]}$$

We use binary session types given as follows.

$$T, S ::= \&^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\} \mid \oplus^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\} \mid \mathtt{end} \mid \ \downarrow^{\mathtt{ses}} (S) \mid \ \uparrow^{\mathtt{ses}} (S) \mid \mu S.S \mid \mathtt{N} \mid \bigstar$$

As in $\pi^{\{1,\overline{1},\omega\}}$, we use shortcuts: when I is a singleton $\{1\}$, we use $?(T_1).S_1$ (resp. $!(T_1).S_1$) to denote $\&^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\}$ (resp. $\oplus^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\}$). Compatibility and duality for $\pi^{\mathtt{session}}$ are the straightforward adaptation of those of $\pi^{\{1,\overline{1},\omega\}}$.

$$(\mathbf{SOut})\,\frac{\Gamma(u) = \uparrow^{\mathsf{ses}} (\overline{T}) \quad \Gamma, x : \overline{T} \vdash_{\mathsf{ses}} P}{\Gamma \vdash_{\mathsf{ses}} \overline{u}(x).P} \qquad (\mathbf{SIn})\,\frac{\Gamma(u) = \downarrow^{\mathsf{ses}} (T) \quad \Gamma, x : T \vdash_{\mathsf{ses}} P}{\Gamma \vdash_{\mathsf{ses}} u(x).P}$$

$$(\mathbf{SCho})\,\frac{(\Gamma, x_i : T_i', s : S_i \vdash_{\mathsf{ses}} P_i)_{i \in I} \qquad J \subseteq I \qquad \forall j \in J. T_j' \sqsubseteq T_j}{\Gamma, s : \&^{\mathsf{ses}}_{i \in J}\{l_i(T_i).S_i\} \vdash_{\mathsf{ses}} s?\{l_i(x_i).P_i\}_{i \in I}}$$

$$(\mathbf{SSel})\,\frac{\Gamma(v) = T_j' \qquad \Gamma, s : S_j \vdash_{\mathsf{ses}} P \qquad T_j' \sqsubseteq \overline{T_j}}{\Gamma, s : \oplus^{\mathsf{ses}}_{i \in I}\{l_i(\overline{T_i}).S_i\} \vdash_{\mathsf{ses}} s!l_j\langle v\rangle.P} \qquad (\mathbf{SPar})\,\frac{\Gamma_1 \vdash_{\mathsf{ses}} P_1 \mid \Gamma_2 \vdash_{\mathsf{ses}} P_2}{\Gamma_1 \odot \Gamma_2 \vdash_{\mathsf{ses}} P_1 \mid P_2}$$

Fig. 5. Some of the main typing rules for $\pi^{\mathtt{session}}$

Some of the key typing rules are given in Figure 5. In session types, the type of a session name s in the prefix $s!l\langle v\rangle.P$ gives information not only on the type of v but also on how the session s will be used in the continuation P.

The encodings. The encoding of $\pi^{\mathtt{session}}$ terms into $\pi^{\{1,\overline{1},\omega\}}$ given in Figure 6. The main points are that, in $\pi^{\{1,\overline{1},\omega\}}$, we lack both synchronous outputs and the way to ensure that sessions behave correctly, that is, how the names in subject positions in later prefixes of the same session, are used following the stipulated protocol, i.e. its session type.

First, to encode the synchronous outputs, we use an administrative synchronisation on a linear name. This is standard [23]: we encode $\overline{a}(v).P$ into $(\nu v, c)\ \overline{a}\langle v, c\rangle \mid c.P$. The new name c is output with the value v and the synchronising party will emit it after inputting the message, thus activating the guarded continuation P. Then, to make sure that sessions are encoded following their protocols, we proceed as follows: if a session name s has type $?(S_1).S_2$, we create a new name k that is given type $[\![S_2]\!]$; and replace in the continuations the name s by k. This gives an equivalent process, and the type of k is now the encoding of the new type S_2 of the session s, after one communication step. The encoding is given by Figure 7.

Soundness of the encoding is stated in the following proposition and proved by induction on the typing derivation.

Proposition 19. *If* $\Gamma \vdash_{\mathsf{ses}} P$ *then* $[\![\Gamma]\!] \vdash_{\pi} [\![P]\!]$.

$$[\![(\nu a)\ P]\!] = (\nu a)\ [\![P]\!] \qquad\qquad [\![u(x).P]\!] = u(x, c).(\overline{c} \mid [\![P]\!])$$

$$[\![\overline{u}(x).P]\!] = (\nu x, c)\ (\overline{u}\langle x, c\rangle \mid c.[\![P]\!])$$

$$[\![k!l\langle e\rangle.P]\!] = (\nu c)\ (k \oplus^{\overline{1}} l\langle e, c\rangle \mid c(s).[\![P]\!]\{s/k\})$$

$$[\![k?\{l_i(x_i).P_i\}_{i \in I}]\!] = (\boldsymbol{\nu} s)\ k\&^{\overline{1}}_{i \in I}\{l_i(x_i, c).([\![P_i]\!]\{s/k\} \mid \overline{c}\langle s\rangle)\}$$

Fig. 6. Encoding of $\pi^{\mathtt{session}}$ terms

$$[\![\oplus^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\}]\!] = \oplus^{\omega}_{i\in I}\{l_i([\![T_i]\!], \downarrow^{1} ([\![S_i]\!]))\}$$

$$[\![\&^{\mathtt{ses}}_{i\in I}\{l_i(T_i).S_i\}]\!] = \oplus^{\omega}_{i\in I}\{l_i([\![T_i]\!], \uparrow^{1} (\overline{[\![S_i]\!]}))\} \qquad [\![\downarrow^{\mathtt{ses}} (S)]\!] = \downarrow^{\overline{1}} ([\![S]\!], \uparrow^{1} (\bigstar))$$

$$[\![\uparrow^{\mathtt{ses}} (S)]\!] = \uparrow^{\overline{1}} ([\![S]\!], \downarrow^{1} (\bigstar)) \qquad [\![\bigstar]\!] = \bigstar \qquad [\![\mathtt{N}]\!] = \mathtt{N}$$

Fig. 7. Encoding of $\pi^{\mathtt{session}}$ types

Example 20. *As an example, consider this toy* $\pi^{\mathtt{session}}$ *process:*

$$\mathbf{S} \;=\; \overline{a}(x).x?(z_2).x!\langle z_2\rangle \mid a(y).(y!\langle 0\rangle.y?(z_1)$$

$\mathbf{S}$ *is composed of two subprocesses, one initiates a new session through channel* a*, then receives and emits, the other behaves dually. Its encoding is given by:*

$$\begin{aligned}[\![\mathbf{S}]\!] = (\nu x, c)(\overline{a}\langle x, c\rangle \mid c.(\nu k_2)\ (x(z_2, c_3).(\overline{c_3}\langle k_2\rangle \mid \overline{k_2}\langle z_2, c_4\rangle \mid c_4)))\\ \mid a(y, c_0).(\overline{c_0} \mid (\nu c_1)\ (\overline{y}\langle 0, c_1\rangle \mid c_1(k_1).(\nu c_2)\ k_1(z_1, c_2).\overline{c_2}))\end{aligned}$$

First, a session initialisation takes place on x *(after* y *has been instantiated to* x *with a "channel" synchronisation), but a new name* k_2 *is later created and transmitted in order to continue the session.*

For subtyping on session types, we can closely follow $\pi^{\{1,\overline{1},\omega\}}$: a relation over types $\mathcal{R}$ in $\pi^{\mathtt{session}}$ is a *subtyping relation* if, whenever $S_1\ \mathcal{R}\ S_2$, we have:

1. either $S_1 = S_2 = \mathtt{N}$ or $S_1 = S_2 = \bigstar$,
2. or (a) $S_1 = \downarrow^{\mathtt{ses}} (S_1)$ (resp. $\uparrow^{\mathtt{ses}} (S_1)$), (b) $S_2 = \downarrow^{\mathtt{ses}} (S_2)$ (resp. $\uparrow^{\mathtt{ses}} (S_2)$), (c) $S_1\ \mathcal{R}\ S_2$
3. or (a) $S_1 = \&^{\mathtt{ses}}_{i\in I}\{l_i(T^1_i).S^1_i\}$, (b) $S_2 = \&^{\mathtt{ses}}_{j\in J}\{l_j(T^2_j).S^2_j\}$, (c) $I \subseteq J$, (c) $\forall j \in J, T^1_j\ \mathcal{R}\ T^2_j$, (d) $\forall j \in J, S^1_j\ \mathcal{R}\ S^2_j$
4. or (a) $S_1 = \oplus^{\mathtt{ses}}_{i\in I}\{l_i(T^1_i).S^1_i\}$, (b) $S_2 = \oplus^{\mathtt{ses}}_{j\in J}\{l_j(T^2_j).S^2_j\}$, (c) $J \subseteq I$, (d) $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$, (e) $\forall i \in I, S^1_i\ \mathcal{R}\ S^2_i$

Then $\sqsubseteq$ is the largest subtyping relation. We can then show the subsumption is admissible in the typing rules for $\pi^{\mathtt{session}}$[1].

Using the fact that branching session prefixes are encoded into branching $\pi^{\{1,\overline{1},\omega\}}$ prefixes, we prove the following proposition, relating subtyping in $\pi^{\mathtt{session}}$ with subtyping in $\pi^{\{1,\overline{1},\omega\}}$.

Proposition 21 *If* $S_1 \sqsubseteq S_2$*, then* $[\![S_1]\!] \sqsubseteq [\![S_2]\!]$

[1] In the subtyping, a carried type for a shared channel is covariant in both output and input: this is because we choose each carried name to be typed as the dual of how the other party should use it, following $\pi^{\{1,\overline{1},\omega\}}$. We can use the standard format through dualisation.

For full abstraction, we use a testing (may-must) equivalence based on [6], both for $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$ processes. A maximal reduction sequence starting form P is a sequence $(P_i)_{i\leq n}$ with $n \in \mathbb{N} \cup \{\omega\}$ such that $P_0 = P$, $\forall i$, $P_i \rightarrow P_{i+1}$ and $P_n \not\rightarrow$.

Definition 22 *We define the barbs for* $\pi^{\{1,\overline{1},\omega\}}$ *and* $\pi^{\mathtt{session}}$ *as follows:*

- *If* $P \in \pi^{\{1,\overline{1},\omega\}}$, $P \Downarrow \overline{a}$ *when* $P \equiv (\nu\widetilde{c})\ (\overline{a} \oplus^m l\langle v\rangle \mid P_1)$ *and* $a \notin \widetilde{c}$.
- *If* $P \in \pi^{\mathtt{session}}$,
  - $P \Downarrow \overline{u}$ *when* $P \equiv (\nu\widetilde{c})\ (\overline{u}(v).P_2 \mid P_1)$ *and* $u \notin \widetilde{c}$
  - *and* $P \Downarrow \overline{s}$ *when* $P \equiv (\nu\widetilde{c})\ (s!l\langle v\rangle.P_2 \mid P_1)$ *and* $s \notin \widetilde{c}$.

We define the may observation for $\pi^{\{1,\overline{1},\omega\}}$ *and* $\pi^{\mathtt{session}}$ *as:* $P \Downarrow^{\mathtt{may}}_{\overline{a}}$ *when there exists* R, $P \rightarrow^* R$, $R \not\rightarrow$ *and* $R \Downarrow \overline{a}$. *We also define the must observation for* $\pi^{\{1,\overline{1},\omega\}}$ *and* $\pi^{\mathtt{session}}$ *as* $P \Downarrow^{\mathtt{must}}_{\overline{a}}$ *when for all maximal reduction sequences* $(P_i)_{i\leq n}$, $n \in \mathbb{N} \cup \{\omega\}$ *starting from* P, $P_j \Downarrow \overline{a}$.

Using Definition 22, we define may, must and testing barbed equivalences (considering only observables from processes), denoted $\sim_{\mathtt{may}}$, $\sim_{\mathtt{must}}$ and $\sim_{\mathtt{test}}$; and the corresponding congruences (considering testers), denoted $\simeq_{\mathtt{may}}$, $\simeq_{\mathtt{must}}$ and $\simeq_{\mathtt{test}}$. In both cases, testing is the conjunction of may and must.

Definition 23. $P \sim_{\mathtt{may}} Q$ *if for all* a, $P \Downarrow^{\mathtt{may}}_{\overline{a}}$ *implies* $Q \Downarrow^{\mathtt{may}}_{\overline{a}}$ *and* $P \rightarrow P'$ *implies there exists* Q' *s.t.* $Q \rightarrow^* Q'*$ *and* $P' \sim_{\mathtt{may}} Q'$.

$P \sim_{\mathtt{must}} Q$ *if for all* a, $P \Downarrow^{\mathtt{must}}_{\overline{a}}$ *implies* $Q \Downarrow^{\mathtt{must}}_{\overline{a}}$ *and* $P \rightarrow P'$ *implies there exists* Q' *s.t.* $Q \rightarrow^* Q'*$ *and* $P' \sim_{\mathtt{must}} Q'$. *Then,* $P \sim_{\mathtt{test}} Q$ *when* $P \sim_{\mathtt{may}} Q$ *and* $P \sim_{\mathtt{must}} Q$.

For $\pi^{\{1,\overline{1},\omega\}}$ *and* $\pi^{\mathtt{session}}$, *we define* $P \simeq_{\mathtt{may}} Q$ *(resp.* $P \simeq_{\mathtt{must}} Q$*) if for all* $R \in \pi^{\{1,\overline{1},\omega\}}$, $(R \mid P) \sim_{\mathtt{may}} (R \mid Q)$ *(resp.* $(R \mid P) \sim_{\mathtt{must}} (R \mid Q)$*). Then,* $P \simeq_{\mathtt{test}} Q$ *when* $P \simeq_{\mathtt{may}} Q$ *and* $P \simeq_{\mathtt{must}} Q$.

Full abstraction. Lemma 24 is crucial, it shows how the original process and its encoding are able to simulate each other. The main difficulty is that the encoding introduces communications on new linear names.

Lemma 24. *Below let* P *be a well-typed* $\pi^{\mathtt{session}}$*-term.*

1. *If* $P \rightarrow P'$, *then there exists* R, $[\![P]\!] \rightarrow R$ *and* $R \rightarrow^+ [\![P']\!]$
2. *If* $[\![P]\!] \rightarrow R$, *then there exists* P', $P \rightarrow P'$ *and* $R \rightarrow [\![P']\!]$.
3. *If* $(Q_i)_{i\leq n}$ *is a maximal reduction sequence starting from* $[\![P]\!]$, *there exists a maximal reduction sequence* $(P_i)_{i\leq n}$ *starting from* P *and a strictly increasing function* $\phi : \mathbb{N} \rightarrow \mathbb{N}$ *s.t.* $[\![P_i]\!] \equiv Q_{\phi(i)}$

The proof of 3 above concerns infinite reduction sequences. In this case, one has first to prove that such a reduction sequence contains an infinite number of non-linear reduction steps.

Lemma 25 (Definability). *For all* $P \in \pi^{\{1,\overline{1},\omega\}}$, $\Gamma \in \pi^{\mathtt{session}}$ *s.t.* $[\![\Gamma]\!] \vdash_{\pi} P$, *exists* $R \in \pi^{\mathtt{session}}$ *s.t.* $\Gamma \vdash_{\mathtt{ses}} R$ *and* $P \simeq_{\mathtt{test}} [\![R]\!]$.

A key idea of the proof is decoding P into a corresponding $\pi^{\mathtt{session}}$-process. First, for every e.g. output at a hidden name say a which is not hereditarily typed in $[\![\Gamma]\!]$, we replace it by an encoding of a session type, similarly for a hidden input. Then all names in P are now typed with the encoding of session types, without changing behaviour. We now use induction on typing rules for $\pi^{\{1,\overline{1},\omega\}}$ to extract the shape of encoded processes from P by induction on the size of P.

Now the computational adequacy lemma concludes the full-abstraction proof.

Lemma 26 (Adequacy). *For all $P \in \pi^{\mathtt{session}}$ s.t. $\Gamma \vdash_{\mathtt{ses}} P$, $P \sim_{\mathtt{test}} [\![P]\!]$.*

Theorem 27 (Full abstraction) *Let P, Q be two $\pi^{\mathtt{session}}$ processes s.t. $\Gamma \vdash_{\mathtt{ses}} P$ and $\Gamma \vdash_{\mathtt{ses}} Q$, then $P \simeq_{\mathtt{test}} Q$ if and only if $[\![P]\!] \simeq_{\mathtt{test}} [\![Q]\!]$.*

Proof (Sketch). For both implications, we present only the 'may' case, the 'must' one is very similar.

- We take a tester K, from Lemma 26, $(P \mid K) \sim_{\mathtt{may}} [\![P \mid K]\!]$. By hypothesis, $[\![P \mid K]\!] \sim_{\mathtt{may}} [\![Q \mid K]\!]$ and by Lemma 26, $[\![Q \mid K]\!] \sim_{\mathtt{may}} Q \mid K$, hence done.
- We take a tester R, from Lemma 25, we get K s.t. $R \simeq_{\mathtt{may}} [\![K]\!]$ and thus $(R \mid [\![P]\!]) \sim_{\mathtt{may}} [\![K \mid P]\!]$. By Lemma 26, $[\![K \mid P]\!] \sim_{\mathtt{may}} (K \mid P)$. By hypothesis, $(K \mid P) \sim_{\mathtt{test}} (K \mid Q)$. By Lemma 26, $(K \mid Q) \sim_{\mathtt{test}} [\![K \mid Q]\!]$ and then, $R \mid [\![P]\!] \sim_{\mathtt{test}} R \mid [\![Q]\!]$.

5 Related Work and Further Topics

In [21], the authors present both the idea of distinguishing output and input types in the π-calculus, and the use of subtyping for controlling the right to perform actions on a given channel. They show that this notion of subtyping allows them to state an operational correspondence between λ-terms and the second encoding into the π-calculus proposed in [16]. The work in [5] addresses the question of the subtyping in a semantic way: a type T_1 is a subtype of a type T_2 if the interpretation of T_2 (the set of all processes that can be typed with T_2) is included into the interpretation of T_1. They prove that their definition of subtyping is decidable and present a π-calculus with dynamic type-checks. The work in [10] first introduces to session types a notion of subtyping based on branching. Through dualisation, the session subtyping we treated in the present paper is essentially identical. The present work has embedded the session subtyping in a more fine-grained linear typing, and has shown that it leads to not only the embedding of the subtyping but also semantic full abstraction, which may shed light on this ordering relation and its theory. The author of [18] studies the semantics of session-types, which contains a notion of subtyping for sessions, called *subsession*, a session S_1 is a subsession of S_2 when for every session S, if $(S_1 \mid S)$ must reach a successful state (according to a *must* semantics similar to the one we use) implies that $(S_2 \mid S)$ must reach a successful state. This notion of subtyping is more related to what we called *compatibility ordering* in Section 2.

Our full abstraction proof for the $\lambda^{\Pi,\Sigma,\sqsubseteq}$ embedding follows game semantics except the subtyping and, in the context of the π-calculus, the one found in [3], which uses both a restriction to a sequential π-calculus and the use of a finite definability lemma.

Our full abstraction proof for the $\pi^{\texttt{session}}$ embedding is inspired by the one in [23] where the authors prove the full abstraction of a polyadic, output-synchronous π-calculus into a monadic, output-asynchronous one. They use a computational adequacy lemma stating that a process and its encoding are barbed bisimilar. Together with a definability result, it leads to the full abstraction result, with a barbed congruence as equivalence. As a comparison, in [23], the meta-calculus (monadic asynchronous π) is a sub-calculus of the target calculus (polyadic synchronous π), which may have facilitated the proof based on barbed bisimilarity. On the other hand, barbed congruence is in terms of the branching structure finer than the testing equivalence. It is an interesting future topic how we can obtain a similar result for such finer equivalences in the present setting.

The calculus we propose in this paper can be easily extended to accommodate more features. For instance, we believe we can adapt a standard definition of polymorphic types [26] in order to include it into our subtyping framework. This opens the possibility of studying the encoding of subtyped System F [4], together with the polymorphic subtypes, into our calculus. We are also interested in showing that our subtyping theory can accommodate objects. Though an encoding of object-oriented calculi into the π-calculus have already been proposed (see [27] for instance) and subtyping for objects is well-studied [1], we believe the analysis as we have carried out in this work will shed new light on their nature. Finally, we are interested in studying how to accommodate other definitions for subtyping for sessions, as the one presented in [19].

Acknowledgements. We thank the CONCUR referees for their useful comments. This work is supported by EPSRC grants EP/F002114/1 and EP/G015481/1.

References

1. Abadi, M., Cardelli, L.: A semantics of object types. In: LICS, pp. 332–341. IEEE Computer Society, Los Alamitos (1994)
2. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. 15(4), 575–631 (1993)
3. Berger, M., Honda, K., Yoshida, N.: Genericity and the pi-calculus. Acta. Inf. 42(2-3), 83–141 (2005)
4. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. 17(4), 471–522 (1985)
5. Castagna, G., Nicola, R.D., Varacca, D.: Semantic subtyping for the pi-calculus. Theor. Comput. Sci. 398, 217–242 (2008)
6. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: Arvind, V., Ramanujam, R., (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 90–102. Springer, Heidelberg (1998)

7. Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types, long version (2011) (in preparation), http://perso.ens-lyon.fr/romain.demangeon/subtyping_long.pdf
8. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and Session Types: An Overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
9. Fiore, M.P., Honda, K.: Recursive types in games: Axiomatics and process representation. In: LICS, pp. 345–356 (1998)
10. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta. Inf. 42(2-3), 191–225 (2005)
11. Honda, K.: Composing processes. In: POPL, pp. 344–357 (1996)
12. Honda, K.: Processes and games. Electr. Notes Theor. Comput. Sci. 71 (2002)
13. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
14. Honda, K., Yoshida, N.: Game-theoretic analysis of call-by-value computation. Theor. Comput. Sci. 221(1-2), 393–456 (1999)
15. Laird, J.: A game semantics of the asynchronous *pi*-calculus. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 51–65. Springer, Heidelberg (2005)
16. Milner, R.: Functions as processes. Mathematical Structures in Computer Science 2(2), 119–141 (1992)
17. Morris, J.H.: Lambda-Calculus Models of Programming Languages. PhD Thesis, M.I.T (1968)
18. Padovani, L.: Session types at the mirror. In: ICE, pp. 71–86 (2009)
19. Padovani, L.: Fair subtyping for multi-party session types. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 127–141. Springer, Heidelberg (2011)
20. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
21. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
22. Pierce, B.C., Steffen, M.: Higher-order subtyping. Theor. Comput. Sci. 176(1-2), 235–282 (1997)
23. Quaglia, P., Walker, D.: Types and full abstraction for polyadic *pi*-calculus. Inf. Comput. 200(2), 215–246 (2005)
24. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
25. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
26. Turner, N.: The polymorphic pi-calculus: Theory and Implementation. PhD thesis, Department of Computer Science, University of Edinburgh (1996)
27. Walker, D.: Objects in the pi-calculus. Inf. Comput. 116(2), 253–271 (1995)
28. Yoshida, N., Berger, M., Honda, K.: Strong Normalisation in the Pi-Calculus. Information and Computation 191(2), 145–202 (2004)

Controlling Reversibility in Higher-Order Pi*

Ivan Lanese[1], Claudio Antares Mezzina[2],
Alan Schmitt[2], and Jean-Bernard Stefani[2]

[1] University of Bologna & INRIA, Italy
[2] INRIA Grenoble-Rhône-Alpes, France

Abstract. We present in this paper a fine-grained rollback primitive for the higher-order π-calculus (HOπ), that builds on the reversibility apparatus of reversible HOπ [9]. The definition of a proper semantics for such a primitive is a surprisingly delicate matter because of the potential interferences between concurrent rollbacks. We define in this paper a high-level operational semantics which we prove sound and complete with respect to reversible HOπ backward reduction. We also define a lower-level distributed semantics, which is closer to an actual implementation of the rollback primitive, and we prove it to be fully abstract with respect to the high-level semantics.

1 Introduction

Motivation and contributions. Reversible computing, or related notions, can be found in many areas, including hardware design, program debugging, discrete-event simulation, biological modeling, and quantum computing (see [2] and the introduction of [10] for early surveys on reversible computing). Of particular interest is the application of reversibility to the study of programming abstractions for fault-tolerant systems. In particular, most fault tolerance schemes based on *system recovery* techniques [1], including rollback/recovery schemes and transaction abstractions, imply some form of undo. The ability to undo any single action in a reversible computation model provides an ideal setting to study, revisit, or imagine alternatives to these different schemes. This is in part the motivation behind the recent development of the reversible process calculi RCCS [4] and $\rho\pi$ [9], with [5] showing how a general notion of interactive transaction emerges from the introduction of irreversible (commit) actions in RCCS. However, these calculi provide very little in the way of controlling reversibility. The notion of irreversible action in RCCS only prevents a computation from rolling back past a certain point. Exploiting the low-level reversibility machinery available in these models of computation for fault-recovery purposes would require more extensive control on the reversal of actions, including *when* they can take place and *how far back* (along a past computation) they apply.

We present in this paper the study of a fine-grained rollback control primitive, where potentially every single step in a concurrent execution can be undone.

* Partly funded by the EU project FP7-231620 HATS, the ANR-2010-SEGI-013 project AEOLUS, and the ANR-2010-BLAN-0305-01 project PiCoq.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 297–311, 2011.

Specifically, we introduce a rollback construct for an asynchronous higher-order π-calculus (HOπ [11]), building on the machinery of $\rho\pi$, the reversible higher-order π-calculus presented in [9]. We chose HOπ as our substrate because we find it a convenient starting point for studying distributed programming models with inherently higher-order features such as dynamic code update, which we aim to combine with abstractions for system recovery and fault tolerance. Surprisingly, finding a suitable definition for a fine-grained rollback construct in HOπ is more difficult than one may think, even with the help of the reversible machinery from [9]. There are two main difficulties. The first one is in actually pinning down the intended effect of a rollback operation, especially in presence of concurrent rollbacks. The second one is in finding a suitably distributed semantics for rollback, dealing only with local information and not relying on complex atomic transitions involving a potentially unbounded number of distinct processes.

We show in this paper how to deal with these difficulties by making the following contributions: (i) we define a high-level operational semantics for a rollback construct in an asynchronous higher-order π-calculus, which we prove *maximally permissive*, in the sense that it makes reachable all past states in a given computation; (ii) we present a low-level semantics for the proposed rollback construct which can be understood as a fully distributed variant of our high-level semantics, and we prove it to be *fully abstract* with respect to the high-level one.

Paper Outline. In Section 2, we informally present our rollback calculus, which we call roll-π, and illustrate the difficulties that may arise in defining a fine-grained rollback primitive. In Section 3, we formalize roll-π and its high-level operational semantics. In Section 4, we present a distributed operational semantics for roll-π, and we prove that it is fully abstract with respect to the high-level one. Section 5 discusses related work and concludes the paper. The interested reader can find proofs of the main results in [8].

2 Informal Presentation

To define roll-π and its rollback construct, we rely on the same support for reversibility as in $\rho\pi$ [9]. Let us review briefly its basic mechanisms.

Reversibility in $\rho\pi$. We attach to each process P a unique tag κ (either simple, written as k, or composite, denoted as $\langle h_i, \tilde{h}\rangle \cdot k$). The uniqueness of tags for processes is achieved thanks to the following structural congruence rule that defines how tags and parallel composition commute.

$$k : \prod_{i=1}^{n} \tau_i \equiv \nu\tilde{h}.\ \prod_{i=1}^{n}(\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \quad \text{with} \;\; \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2 \qquad (1)$$

In equation (1), $\prod_{i=1}^{n}$ is n-ary parallel composition and ν is the restriction operator, both standard from the π-calculus. Each *thread* τ_i is either a message, of the form

$a\langle P\rangle$ (where a is a channel name), or a receiver process (also called a *trigger*), of the form $a(X) \triangleright P$. A *forward* computation step (or forward reduction step, noted with arrow $\twoheadrightarrow$) consists of the reception of a message by a receiver process, and takes the following form (note that $\rho\pi$ is an asynchronous calculus).

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k.\, k : Q\{^P/_X\} \mid [M; k] \qquad (2)$$

In this forward step, κ_1 identifies a thread consisting of message $a\langle P\rangle$ on channel a, and κ_2 identifies a thread consisting of a trigger process $a(X) \triangleright Q$ that expects a message on channel a. The result of the message input yields, as usual, an instance $Q\{^P/_X\}$ of the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., the process P ($\rho\pi$ is higher-order). Message input also has two side effects: (i) the tagging of the newly created process $Q\{^P/_X\}$ by a fresh tag k, and (ii) the creation of a memory $[M; k]$, which records the original two threads, $M = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q)$, together with tag k.

In $\rho\pi$, a forward reduction step such as (2) above is systematically associated with a backward reduction step (noted with arrow $\rightsquigarrow$) of the form:

$$(k : Q) \mid [M; k] \rightsquigarrow M \qquad (3)$$

which undoes the communication between threads κ_1 and κ_2. When necessary to avoid confusion, we will add a $\rho\pi$ subscript to arrows representing $\rho\pi$ reductions.

Given a configuration M, the set of memories present in M provides us with an ordering $:>$ between tags in M that reflects their causal dependency: if memory $[\kappa_1 : P_1 \mid \kappa_2 : P_2; k]$ occurs in M, then $\kappa_i > k$. Also, $k > \langle h_i, \tilde{h}\rangle \cdot k$, and we define the relation $:>$ as the reflexive and transitive closure of the $>$ relation. We say that tag κ has κ' as a causal antecedent if $\kappa' :> \kappa$.

Reversibility in roll-π*.* The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. In roll-π, we exploit this intuition to introduce an explicit form of backward reduction. Specifically, backward reduction is not allowed by default as in $\rho\pi$, but has to be triggered by an instruction of the form roll k, whose intent is that the current computation be rolled back to a state just prior to the creation of the memory bearing the tag k. To be able to form an instruction of the form roll k, one needs a way to pass the knowledge of a memory tag to a process. This is achieved in roll-π by adding a bound variable to each trigger process, which now takes the form $a(X) \triangleright_\gamma P$, where γ is the tag variable bound by the trigger construct and whose scope is P. A forward reduction step in roll-π therefore is:

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [M; k] \qquad (4)$$

where the only difference with (2) lies in the fact that the newly created tag k is passed as an argument to the trigger body Q. We write $a(X) \triangleright P$ in place of $a(X) \triangleright_\gamma P$ if the tag variable γ does not appear free in P.

Now, given the above intent for the rollback primitive roll, how does one define its operational semantics? As hinted at in the introduction, this is actually a subtler affair than one may expect. A big difference with $\rho\pi$, where communication steps are undone one by one, is that the k in $\mathsf{roll}\ k$ may refer to a communication step far in the past. So the idea behind a $\mathsf{roll}\ k$ is to restore the content of a memory $[M;k]$ and to delete all its forward history. Consider the following attempt at a rule for roll:

$$(\textsc{Naive})\quad \frac{N \blacktriangleright k \qquad \texttt{complete}(N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k))}{N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k) \rightsquigarrow M \mid N\lightning_k}$$

The different predicates and the $\lightning$ operator used in the rule are defined formally in the next section, but an informal explanation should be enough to understand how the rule works. Briefly, the assertion $N \blacktriangleright k$ states that all the active threads and memories in N bear tags κ that have k as causal antecedent, i.e., $k :> \kappa$ (N does not contain unrelated processes). The assertion $\texttt{complete}(M_c)$ states that configuration M_c gathers all the threads (inside or outside memories) whose tags have as a causal antecedent the tag of a memory in M_c itself, i.e., if a memory in M_c is of the form $[M';k']$ (the communication M' created a process tagged with k'), then a process or a memory containing a process tagged with k' has to be in M_c (M_c contains every related process). The premises of rule Naive thus asserts that the configuration $M_c = N \mid [M;k] \mid \kappa : \mathsf{roll}\ k$, on the left hand side of the reduction in the conclusion of the rule, gathers all (and only) the threads and memories which have originated from the process tagged by k, itself created by the interaction of the message and trigger recorded in M. Being complete, M_c is thus ready to be rolled back and replaced by the configuration M which is at its origin. Rolling back M_c has another effect, noted as $N\lightning_k$ in the right hand side of the conclusion, which is to release from memories those messages or triggers which do not have k as a causal antecedent, but which participated in communications with causal descendants of k.

For instance, the configuration $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$, has the following forward reductions (where $M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$):

$$\begin{aligned} M_0 &\twoheadrightarrow \nu k.\, [M_1;k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\ &\twoheadrightarrow \nu k, l.\, [M_1;k] \mid [M_2;l] \mid (l : \mathsf{roll}\ k) = M_3 \end{aligned}$$

Applying rule Naive (and structural congruence, defined later) on M_3 we get:

$$M_3 \rightsquigarrow M_1 \mid [M_2;l]\lightning_k = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y) = M_0$$

where $(\kappa_2 : c(Y) \triangleright_\delta Y)$ is released from memory $[M_2;l]$ because it does not have k as a causal antecedent.

Rule Naive looks reasonable enough, but difficulties arise when concurrent rollbacks are taken into account. Consider the following configuration:

$$M = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0}\rangle)$$

where[1] $\tau_1 = a(X) \triangleright_\gamma d\langle \mathbf{0}\rangle \mid (c(Y) \triangleright \mathsf{roll}\ \gamma)$ and $\tau_3 = b(Z) \triangleright_\delta c\langle \mathbf{0}\rangle \mid (d(U) \triangleright \mathsf{roll}\ \delta)$.

[1] We assume parallel composition has precedence over trigger.

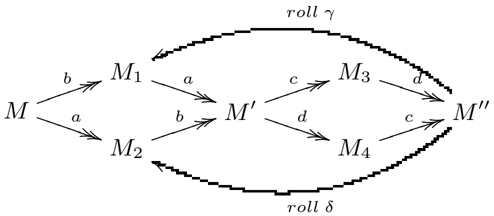


Fig. 1. Concurrent rollback anomaly

The most interesting reductions of M are depicted in Figure 1. Forward reductions are labelled by the name of the channel used for communication, while backward reductions are labelled by the executed roll instruction. The main processes and short-cuts are detailed below:

$$M_1 = \nu l_2, h_3, h_4.\, \sigma_1 \mid [\sigma_2; l_2] \mid (\kappa_3 : c\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4)$$
$$M_2 = \nu l_1, h_1, h_2.\, [\sigma_1; l_1] \mid (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_2 : \tau_2) \mid \sigma_2$$
$$M'' = \nu l_1 \dots l_3, h_1 \dots h_4.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2)$$

$$\sigma_1 = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0} \rangle) \qquad \sigma_2 = (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0} \rangle) \qquad \tau_2 = c(Y) \triangleright \mathsf{roll}\ l_1$$
$$\sigma_3 = (\kappa_2 : \tau_2) \mid (\kappa_3 : c\langle \mathbf{0} \rangle) \qquad \sigma_4 = (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4) \qquad \tau_4 = d(U) \triangleright \mathsf{roll}\ l_2$$

The anomaly here is that there is no way from M_1 or M_2 to get back to the original configuration M, despite the fact that M'' has two roll instructions which would seem sufficient to undo all the reductions which lead from M to M''. Note that M_1 and M_2 are configurations which could both have been reached from M. Thus rule Naive is not unsound, but incomplete or insufficiently permissive, at least with respect to what is possible in $\rho\pi$: if we were to undo actions in M'' step by step, using $\rho\pi$'s backward reductions, we could definitely reach all of M, M_1, and M_2. Note that the higher-order aspects do not matter here.

The main motivation to have a complete rule comes from the fact that, in an abstract semantics, one wants to be as liberal as possible, and not unduly restrict implementations. If we were to pick the Naive rule as our semantics for rollback, then a correct implementation would have to enforce the same restrictions with respect to states reachable from backward reductions, restrictions which, in the case of rule Naive, are both complex to characterize (in terms of conflicting rollbacks) and quite artificial since they do not correspond to any clear execution policy. In the next section, we present a *maximally permissive* semantics for rollback, using $\rho\pi$ as our benchmark for completeness.

3 The roll-π Calculus and Its High-Level Semantics

3.1 Syntax

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, the

$$\begin{aligned}
P, Q &::= \mathbf{0} \mid X \mid \nu a.\, P \mid (P \mid Q) \mid a\langle P\rangle \mid a(X) \triangleright_\gamma P \mid \mathsf{roll}\ k \mid \mathsf{roll}\ \gamma \\
M, N &::= \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid \kappa : P \mid [\mu; k] \mid [\mu; k]^\bullet \\
\kappa &::= k \mid \langle h, \tilde{h}\rangle \cdot k \\
\mu &::= ((\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)) \\
& a \in \mathcal{N} \quad X \in \mathcal{V}_\mathcal{P} \quad \gamma \in \mathcal{V}_\mathcal{K} \quad u \in \mathcal{I} \quad h, k \in \mathcal{K}
\end{aligned}$$

Fig. 2. Syntax of roll-π

set $\mathcal{V}_\mathcal{K}$ of *tag variables*, and the set $\mathcal{V}_\mathcal{P}$ of process variables. The set $\mathcal{I} = \mathcal{N} \cup \mathcal{K}$ is called the set of *identifiers*. We note $\mathbb{N}$ the set of natural integers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{I}$; δ, γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We note $\tilde{u}$ a finite set of identifiers $\{u_1, \ldots, u_n\}$.

Syntax. The syntax of the roll-π calculus is given in Figure 2 (we often add balanced parenthesis around roll-π terms to disambiguate them). *Processes*, given by the P, Q productions in Figure 2, are the standard processes of the asynchronous higher-order π-calculus, except for the presence of the roll primitive and the extra bound tag variable in triggers. A trigger in roll-π takes the form $a(X) \triangleright_\gamma P$, which allows the receipt of a message of the form $a\langle Q\rangle$ on channel a, and the capture of the tag of the receipt event with tag variable γ.

Processes in roll-π cannot directly execute, only *configurations* can. *Configurations* in roll-π are given by the M, N productions in Figure 2. A configuration is built up from *tagged processes* and *memories*.

In a tagged process $\kappa : P$ the tag κ is either a single key k or a pair of the form $\langle h, \tilde{h}\rangle \cdot k$, where $\tilde{h}$ is a set of keys with $h \in \tilde{h}$. A tag serves as an identifier for a process. As in $\rho\pi$ [9], tags and memories help capture the flow of causality in a computation.

A *memory* is a configuration of the form $[\mu; k]$, which keeps track of the fact that a configuration μ was reached during execution, that triggered the launch of a process tagged with the fresh tag k. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory, and k the *tag* of the memory. The configuration part $\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)$ of a memory records the message $a\langle P\rangle$ and the trigger $a(X) \triangleright_\gamma Q$ involved in the message receipt, together with their respective thread tags κ_1, κ_2. A *marked memory* is a configuration of the form $[\mu; k]^\bullet$, which just serves to indicate that a rollback operation targeting this memory has been initiated.

We note $\mathcal{P}$ the set of roll-π processes, and $\mathcal{C}$ the set of roll-π configurations. We call *agent* an element of the set $\mathcal{A} = \mathcal{P} \cup \mathcal{C}$. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$; L, M, N range over $\mathcal{C}$; and A, B, C range over $\mathcal{A}$. We call *thread*, a process that is either a message $a\langle P\rangle$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction roll k. We let τ and its decorated variants range over threads.

Free identifiers and free variables. Notions of free identifiers and free variables in $\mathsf{roll}\text{-}\pi$ are usual. Constructs with binders are of the following forms: $\nu a. P$ binds the name a with scope P; $\nu u. M$ binds the identifier u with scope M; and $a(X) \triangleright_\gamma P$ binds the process variable X and the tag variable γ with scope P. We note $\mathtt{fn}(P)$, $\mathtt{fn}(M)$, and $\mathtt{fn}(\kappa)$ the set of free names, free identifiers, and free keys, respectively, of process P, of configuration M, and of tag κ. Note in particular that $\mathtt{fn}(\kappa : P) = \mathtt{fn}(\kappa) \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$, $\mathtt{fn}(k) = \{k\}$ and $\mathtt{fn}(\langle h, \tilde{h} \rangle \cdot k) = \tilde{h} \cup \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or tag) variable. We note $\mathcal{P}^{cl}$, $\mathcal{C}^{cl}$ and $\mathcal{A}^{cl}$ the sets of closed processes, configurations, and agents, respectively.

Initial and consistent configurations. Not all configurations allowed by the syntax in Figure 2 are meaningful. For instance, in a memory $[\mu; k]$, tags occurring in the configuration part μ must be different from the key k; if a tagged process $\kappa_1 : \mathsf{roll}\ k$ occurs in a configuration M, we expect a memory $[\mu; k]$ to occur in M as well. In the rest of the paper, we only will be considering well-formed, or *consistent*, closed configurations. A configuration is consistent if it can be derived using the rules of the calculus from an *initial* configuration. A configuration is initial if it does not contain memories, all the tags are distinct and simple (i.e., of the form k), and the argument of each roll is bound by a trigger.

We do not give here a syntactic characterization of consistent configurations as it is not essential to understand the developments in this paper (the interested reader may find some more details in [9], where a syntactic characterization of $\rho\pi$ consistent configurations is provided).

Remark 1. We have no construct for replicated processes or guarded choice in $\mathsf{roll}\text{-}\pi$: as in HOπ, these can easily be encoded.

Remark 2. In the remainder of the paper, we adopt *Barendregt's Variable Convention*: if terms $t_1, \ldots, t_n$ occur in a certain context (e.g., definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones.

3.2 Operational Semantics

The operational semantics of the $\mathsf{roll}\text{-}\pi$ calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over closed configurations ($\rightarrow \subset \mathcal{C}^{cl} \times \mathcal{C}^{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations ($\equiv \subset \mathcal{P}^2 \cup \mathcal{C}^2$). We define *evaluation contexts* as "configurations with a hole $\cdot$", given by the following grammar:

$$\mathbb{E} ::= \cdot \mid (M \mid \mathbb{E}) \mid \nu u. \mathbb{E}$$

General contexts $\mathbb{C}$ are just processes or configurations with a hole $\cdot$. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that is closed for general contexts: $P \mathrel{\mathcal{R}} Q \implies \mathbb{C}[P] \mathrel{\mathcal{R}} \mathbb{C}[Q]$ or $M \mathrel{\mathcal{R}} N \implies \mathbb{C}[M] \mathrel{\mathcal{R}} \mathbb{C}[N]$.

The relation $\equiv$ is defined as the smallest congruence on processes and configurations that satisfies the rules in Figure 3. We note $t =_\alpha t'$ when terms

$$(\text{E.}\textsc{ParC})\ A \mid B \equiv B \mid A \qquad (\text{E.}\textsc{ParA})\ A \mid (B \mid C) \equiv (A \mid B) \mid C$$

$$(\text{E.}\textsc{ParN})\ A \mid \mathbf{0} \equiv A \qquad (\text{E.}\textsc{NewN})\ \nu u.\, \mathbf{0} \equiv \mathbf{0} \qquad (\text{E.}\textsc{NewC})\ \nu u.\, \nu v.\, A \equiv \nu v.\, \nu u.\, A$$

$$(\text{E.}\textsc{NewP})\ (\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B) \qquad (\text{E.}\alpha)\ A =_\alpha B \implies A \equiv B$$

$$(\text{E.}\textsc{TagN})\ \kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$$

$$(\text{E.}\textsc{TagP})\ k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\ \prod_{i=1}^{n} (\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 3. Structural congruence for roll-π

t, t' are equal modulo α-conversion. If $\tilde{u} = \{u_1, \ldots, u_n\}$, then $\nu\tilde{u}.\, A$ stands for $\nu u_1. \ldots \nu u_n.\, A$. We note $\prod_{i=1}^{n} A_i$ for $A_1 \mid \ldots \mid A_n$ (there is no need to indicate how the latter expression is parenthesized because the parallel operator is associative by rule E.ParA). In rule E.TagP, processes τ_i are threads. Recall the use of the variable convention in these rules: for instance, in the rule $(\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$ the variable convention makes implicit the condition $u \notin \mathtt{fn}(B)$. The structural congruence rules are the usual rules for the π-calculus (E.ParC to E.α) without the rule dealing with replication, and with the addition of two new rules dealing with tags: E.TagN and E.TagP. Rule E.TagN is a scope extrusion rule to push restrictions to the top level. Rule E.TagP allows to generate unique tags for each thread in a configuration. An easy induction on the structure of terms provides us with a kind of normal form for configurations (by convention $\prod_{i \in I} A_i = \mathbf{0}$ if $I = \emptyset$, and $[\mu; k]^\circ$ stands for $[\mu; k]$ or $[\mu; k]^\bullet$):

Lemma 1 (Thread normal form). *For any configuration M, we have*

$$M \equiv \nu\tilde{u}.\ \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [\mu_j; k_j]^\circ$$

with $\rho_i = \mathbf{0}$, $\rho_i = \mathsf{roll}\ k_i$, $\rho_i = a_i\langle P_i\rangle$, or $\rho_i = a_i(X_i) \triangleright_{\gamma_i} P_i$.

We say that a binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$(\text{R.}\textsc{Ctx})\ \frac{M \mathrel{\mathcal{R}} N}{\mathbb{E}[M] \mathrel{\mathcal{R}} \mathbb{E}[N]} \qquad (\text{R.}\textsc{Eqv})\ \frac{M \equiv M' \qquad M' \mathrel{\mathcal{R}} N' \qquad N' \equiv N}{M \mathrel{\mathcal{R}} N}$$

The reduction relation $\rightarrow$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the *backward* reduction relation $\rightsquigarrow$: $\rightarrow\ =\ \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined to be the smallest evaluation-closed binary relations on closed configurations satisfying the rules in Figure 4 (note again the use of the variable convention: in rule H.Com the key k is fresh).

The rule for forward reduction H.Com is the standard communication rule of the higher-order π-calculus with three side effects: (i) the creation of a new

$$(\text{H.C}\textsc{om}) \quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\text{H.S}\textsc{tart}) \quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^{\bullet}$$

$$(\text{H.R}\textsc{oll}) \quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [\mu; k])}{N \mid [\mu; k]^{\bullet} \rightsquigarrow \mu \mid N \lightning_k}$$

Fig. 4. Reduction rules for roll-π

memory to record the configuration that gave rise to it; (ii) the tagging of the continuation of the message receipt with the fresh key k; (iii) the passing of the newly created tag k as a parameter to the newly launched instance of the trigger's body Q.

Backward reduction is subject to the rules H.Roll and H.Start. Rule H.Roll is similar to rule Naive defined in the previous section, except that it relies on the presence of a marked memory instead of on the presence of the process $\kappa : \mathsf{roll}\ k$ to roll back a given configuration. Rule H.Start just marks a memory to enable rollback.

The definition of rule H.Roll exploits several predicates and relations which we define below.

Definition 1 (Causal dependence). *Let M be a configuration and let T_M be the set of tags occurring in M. The binary relation $>_M$ on T_M is defined as the smallest relation satisfying the following clauses:*

- *$k >_M \langle h_i, \tilde{h}\rangle \cdot k$;*
- *$\kappa' >_M k$ if κ' occurs in μ for some memory $[\mu; k]^\circ$ that occurs in M.*

The causal dependence relation $:>_M$ is the reflexive and transitive closure of $>_M$.

Relation $\kappa :>_M \kappa'$ reads "κ is a causal antecedent of κ' according to M". When configuration M is clear from the context, we write $\kappa :> \kappa'$ for $\kappa :>_M \kappa'$.

Definition 2 (κ dependence). *Let $M \equiv \nu\tilde{u}.\, \prod_{i \in I} \kappa_i : \rho_i \mid \prod_{j \in J} [\mu_j; \kappa_j]^\circ$. Configuration M is κ-dependent, written $M \blacktriangleright \kappa$, if $\forall i \in I \cup J$, $\kappa :>_M \kappa_i$.*

We now define the *projection* operation on configurations $M \lightning_\kappa$, that captures the parallel composition of all tagged processes that do not depend on κ occurring in memories in M.

Definition 3 (Projection). *Let $M \equiv \nu\tilde{u}.\, \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [\mu_j; \kappa_j]^\circ$, with $\mu_j = \kappa'_j : R_j \mid \kappa''_j : T_j$. Then:*

$$M \lightning_\kappa = \nu\tilde{u}.\, (\prod_{j' \in J'} \kappa'_{j'} : R_{j'}) \mid (\prod_{j'' \in J''} \kappa''_{j''} : T_{j''})$$

where $J' = \{j \in J \mid \kappa :\not\succ \kappa'_j\}$ *and* $J'' = \{j \in J \mid \kappa :\not\succ \kappa''_j\}$.

Finally we define the notion of *complete* configuration, used in the premise of rule H.Roll.

Definition 4 (Complete configuration). *A configuration M* contains *a tagged process $\kappa : P$, written $\kappa : P \in M$, if $M \equiv \nu\tilde{u}.\,(\kappa : P) \mid N$ or $M \equiv \nu\tilde{u}.\,[\kappa : P \mid \kappa_1 : Q; k]^\circ \mid N$.*

A configuration M is complete*, noted* $\texttt{complete}(M)$*, if for each memory $[\mu; k]^\circ$ that occurs in M, one of the following holds:*

1. *There exists a process P such that $k : P \in M$.*
2. *There is $\tilde{h}$ such that for each $h_i \in \tilde{h}$ there exists a process P_i such that $\langle h_i, \tilde{h}\rangle \cdot k : P_i \in M$.*

Barbed bisimulation. The operational semantics of the roll-π calculus is completed classically by the definition of a contextual equivalence between configurations, which takes the form of a barbed congruence. We first define observables in configurations. We say that name a is *observable in configuration M*, noted $M \downarrow_a$, if $M \equiv \nu\tilde{u}.\,(\kappa : a\langle P\rangle) \mid N$, with $a \notin \tilde{u}$. Keys are not observable: this is because they are just an internal device used to support reversibility. We note $\Rightarrow$, $\twoheadrightarrow^*$, $\rightsquigarrow^*$ the reflexive and transitive closures of $\rightarrow$, $\twoheadrightarrow$, and $\rightsquigarrow$, respectively.

One of the aims of this paper is to define a low-level semantics for roll-π, and show that it is equivalent to the high-level one. We want to use weak barbed congruence for this purpose. Thus we need a definition of barbed congruence able to relate roll-π configurations executed under different semantics. These semantics will also rely on different runtime syntaxes. Thus, we define a family of relations, each labeled by the semantics to be used on the left and right components of its elements. We also label sets of configurations with the corresponding semantics, thus highlighting that the corresponding runtime syntax has to be included. However, contexts do not include runtime syntax, since we never add contexts at runtime.

Definition 5 (Barbed bisimulation and congruence). *A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ on closed consistent configurations is a* strong *(resp.* weak*)* barbed simulation *if whenever $M \;{}_{s1}\mathcal{R}_{s2}\; N$*

- *$M \downarrow_a$ implies $N \downarrow_a$ (resp. $N \Rightarrow_{s2}\downarrow_a$)*
- *$M \rightarrow_{s1} M'$ implies $N \rightarrow_{s2} N'$, with $M' {}_{s1}\mathcal{R}_{s2} N'$ (resp. $N \Rightarrow_{s2} N'$ with $M' {}_{s1}\mathcal{R}_{s2} N'$)*

A relation ${}_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ is a strong *(resp.* weak*)* barbed bisimulation *if ${}_{s1}\mathcal{R}_{s2}$ and $({}_{s1}\mathcal{R}_{s2})^{-1}$ are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* barbed bisimilarity *and note ${}_{s1}{\sim}_{s2}$ (resp. ${}_{s1}{\approx}_{s2}$) the largest strong (resp. weak) barbed bisimulation with respect to semantics $s1$ and $s2$.*

We say that two configurations M and N are strong *(resp.* weak*)* barbed congruent*, written ${}_{s1}{\sim}^c_{s2}$ (resp. ${}_{s1}{\approx}^c_{s2}$), if for each roll-π context $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are consistent, then $\mathbb{C}[M] \;{}_{s1}{\sim}_{s2}\; \mathbb{C}[N]$ (resp. $\mathbb{C}[M] \;{}_{s1}{\approx}_{s2}\; \mathbb{C}[N]$).*

3.3 Soundness and Completeness of Backward Reduction in roll-π

We present in this section a Loop Theorem, that establishes the soundness of backward reduction in roll-π, and we prove the completeness (or maximal permissiveness) of backward reduction in roll-π.

Theorem 1 (Loop Theorem - Soundness of backward reduction). *For any (consistent) configurations M and M' with no marked memories, if $M \rightsquigarrow^* M'$, then $M' \twoheadrightarrow^* M$.*

To state the completeness result for backward reduction in roll-π, we define a family of functions $\phi_e : \mathcal{C}_{\mathsf{roll}\text{-}\pi} \rightarrow \mathcal{C}_{\rho\pi}$, where $e \in \mathcal{N}$, mapping a roll-π configuration to a $\rho\pi$ configuration. Function ϕ_e is defined by induction as follows:

$$\begin{array}{ccc}
\phi_e(\nu u.\, A) = \nu u.\, \phi_e(A) & \phi_e(A \mid B) = \phi_e(A) \mid \phi_e(B) & \phi_e(\kappa : P) = \kappa : \phi_e(P) \\
\phi_e([\mu; k]^\circ) = [\phi_e(\mu); k] & \phi_e(\mathbf{0}) = \mathbf{0} & \phi_e(X) = X \\
\phi_e(\mathsf{roll}\ k) = e\langle \mathbf{0} \rangle & \phi_e(\mathsf{roll}\ \gamma) = e\langle \mathbf{0} \rangle & \phi_e(a\langle P \rangle) = a\langle \phi_e(P) \rangle
\end{array}$$
$$\phi_e(a(X) \triangleright_\gamma P) = a(X) \triangleright \phi_e(P)$$

Note that roll instructions are transformed not into $\mathbf{0}$ but into a thread $e\langle \mathbf{0} \rangle$: this is to ensure a consistent roll-π configuration is transformed into a consistent $\rho\pi$ configuration (recall that $\mathbf{0}$ is not a thread, thus it may be collected by structural congruence and there would be no thread corresponding to the $\mathsf{roll}\ k$ process).

We now state that roll-π is maximally permissive: any subset of roll primitives in evaluation context may successfully be executed, unlike in the naive example of Section 2. Let $M = \nu \tilde{u}.\, [\mu; k] \mid (k : P) \mid N$ be a $\rho\pi$ configuration and $S = \{k_1, \ldots, k_n\}$ a set of keys. We note $M \rightsquigarrow_S M'$ if $M \rightsquigarrow_{\rho\pi} M'$, $M' = \nu \tilde{u}.\, \mu \mid N$, and $k_i :> k$ for some $k_i \in S$ (here k is the key of the memory $[\mu; k]$ consumed by the reduction). If $M' \not\rightsquigarrow_S$, we say that M' is *final with respect to S*. We note $\rightsquigarrow_S^*$ the reflexive and transitive closure of $\rightsquigarrow_S$. We assume here that reductions are name-preserving, i.e., existing keys are not α-converted (cf. [9] for a discussion on the topic).

Theorem 2 (Completeness of backward reduction). *Let M be a (consistent) roll-π configuration such that $M \equiv \nu \tilde{u}.\, \prod_{i=1}^{n} \kappa_i : \mathsf{roll}\ k_i \mid M_1$, let $S = \{k_1, \ldots, k_n\}$, and let $e \in \mathcal{N} \setminus \mathtt{fn}(M)$. Then for all $T \subseteq S$, if $\phi_e(M) \rightsquigarrow_T^* N$ and N is final with respect to T, there exists M' such that $N = \phi_e(M')$, and $M \rightsquigarrow_{\mathsf{roll}\text{-}\pi}^* M'$.*

4 A Distributed Semantics for roll-π

The semantics defined in the previous section captures the behavior of rollback, but its H.Roll rule specifies an atomic action involving a configuration with an unbounded number of processes and relies on global checks on this configuration, for verifying that it is complete and κ-dependent. This makes it arduous to implement, especially in a distributed setting.

$$(\textsc{L.Com})\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow_{LL} \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$(\textsc{L.Start})\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{LL} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet \mid \mathsf{rl}\ k$$

$$(\textsc{L.Span})\quad \mathsf{rl}\ \kappa_1 \mid [\kappa_1 : P \mid M; k]^\circ \rightsquigarrow_{LL} [\lfloor \kappa_1 : P \rfloor \mid M; k]^\circ \mid \mathsf{rl}\ k$$

$$(\textsc{L.Branch})\quad \frac{\langle h_i, \tilde{h}\rangle \cdot k \text{ occurs in } M}{\mathsf{rl}\ k \mid M \rightsquigarrow_{LL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \mid M}$$

$$(\textsc{L.Up})\quad \mathsf{rl}\ \kappa_1 \mid (\kappa_1 : P) \rightsquigarrow_{LL} \lfloor \kappa_1 : P \rfloor \qquad (\textsc{L.Stop})\quad [\mu; k]^\circ \mid \lfloor k : P \rfloor \rightsquigarrow_{LL} \mu$$

Fig. 5. Reduction rules for LL

$$(\textsc{E.Gb1})\quad \nu k.\, \mathsf{rl}\ k \equiv_{LL} 0 \qquad (\textsc{E.Gb2})\quad \nu k.\, \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \equiv_{LL} 0$$

$$(\textsc{E.TagPFr})\ \lfloor k : \prod_{i=1}^{n} \tau_i \rfloor \equiv_{LL} \nu \tilde{h}.\, \prod_{i=1}^{n} \lfloor (\langle h_i, \tilde{h}\rangle \cdot k : \tau_i) \rfloor \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Fig. 6. Additional structural laws for LL

We thus present in this section a low-level (written LL) semantics, where the conditions above are verified incrementally by relying on the exchange of rl notifications. We show that the LL semantics captures the same intuition as the one introduced in Section 3 by proving that given a (consistent) configuration, its behaviors under the two semantics are weak barbed congruent according to Definition 5.

To avoid confusion between the two semantics, we use a subscript LL to identify all the elements (reductions, structural congruence, ...) referred to the low-level semantics presented here, and HL (for high-level) for the semantics described in Section 3.

The LL semantics $\rightarrow_{LL}$ of roll-π is defined as for the HL one (cf. Section 3.2), as $\rightarrow_{LL} = \twoheadrightarrow_{LL} \cup \rightsquigarrow_{LL}$, where relations $\twoheadrightarrow_{LL}$ and $\rightsquigarrow_{LL}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 5. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on LL processes and configurations that satisfies the rules in Figure 3 and in Figure 6.

LL configurations differ from HL configurations in two aspects. First, tagged processes (inside or outside memories) can be frozen, denoted $\lfloor \kappa : P \rfloor$, to indicate that they are participating to a rollback (rollback is no longer atomic). Second, LL configurations include notifications of the form $\mathsf{rl}\ \kappa$, used to notify a tagged process with key κ to enter a rollback.

Let us describe the LL rules. Communication rule L.Com is as before. The main idea for rollback is that when a memory pointed by a roll is marked (rule L.Start), a notification $\mathsf{rl}\ k$ is generated. This notification is propagated by rules L.Span and L.Branch. Rule L.Span also freezes threads inside memories, specifying that they will be eventually removed by the rollback. Rule L.Branch (where the predicate "κ occurs in M" means that either $M = \kappa : P$ or $M = [\mu; k']^{\circ}$ with $\kappa : P \in M$) is used when the target configuration has been split into multiple threads: a notification has to be sent to each of them. Rule L.Up is similar to L.Span, but it applies to tagged processes outside memories. It also stops the propagation of the rl notification. The main idea is that by using rules L.Span, L.Branch, and L.Up one is able to tag all the causal descendants of a marked memory. Finally, rule L.Stop rolls back a single computation step by removing a frozen process and freeing the content of the memory created with it. In the LL semantics a rollback request is thus executed incrementally, while it was atomic in the HL semantics (rule H.Roll). The LL semantics also exploits an extended structural congruence, adding axioms E.Gb1 and E.Gb2 to garbage collect rl notifications when they are no more needed, and extending axiom E.TagP to deal with frozen threads (axiom E.TagPFr).

We now show an example to clarify the semantics (each reduction is labeled by the name of the axiom used to derive it). Let $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$. We have:

$$\begin{array}{rl} M_0 & \twoheadrightarrow \nu k.\, [M_1; k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \\ (\text{L.Com}) & \twoheadrightarrow \nu k, l.\, [M_1; k] \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \\ (\text{L.Start}) & \rightsquigarrow \nu k, l.\, [M_1; k]^{\bullet} \mid [M_2; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ k \\ (\text{L.Span}) & \rightsquigarrow \nu k, l.\, [M_1; k]^{\bullet} \mid [M_2'; l] \mid (l : \mathsf{roll}\ k) \mid \mathsf{rl}\ l \\ (\text{L.Up}) & \rightsquigarrow \nu k, l.\, [M_1; k]^{\bullet} \mid [M_2'; l] \mid \lfloor (l : \mathsf{roll}\ k) \rfloor \\ (\text{L.Stop}) & \rightsquigarrow \nu k.\, [M_1; k]^{\bullet} \mid M_2' \\ (\text{L.Stop}) & \rightsquigarrow M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \end{array}$$

where:

$$M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \triangleright_\delta Y) \quad M_2' = \lfloor (k : c\langle \mathsf{roll}\ k\rangle) \rfloor \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$$

One can see that the rollback operation starts with the application of the rule L.Start, whose effects are (i) to mark the memory aimed by a roll process, and (ii) to generate a notification $\mathsf{rl}\ k$ to freeze its continuation. Since the continuation of the memory $[M_1; k]$ is contained in the memory $[M_2; l]$ then the rule L.Span is applied. So, the part of the memory containing the tag k gets frozen and a freeze notification $\mathsf{rl}\ l$ is generated. The notification eventually reaches the process $l : \mathsf{roll}\ k$ and freezes it (rule L.Up). Now, since there exists a memory whose continuation is a frozen process, we can apply the rule L.Stop, and free the configuration part of the memory (M_2'). Again, we have that the continuation of $[M_1; k]$ is a frozen process and by applying the rule L.STOP we can free the configuration M_1, obtaining the initial configuration. In general, a rollback of a step whose memory is tagged by k is performed by executing a top-down visit of its causal descendants, freezing them, followed by a bottom-up visit undoing the steps one at the time.

We can now state the correspondence result between the two semantics.

Theorem 3 (Correspondence between HL and LL). *For each roll-π HL consistent configuration M, $M\ {}_{HL}{\approx}^c_{LL}\ M$.*

Proof. The proof is quite long and technical, and relies on a several additional semantics used as intermediate steps from HL to LL. It can be found in [8]. □

This result can be easily formulated as full abstraction. In fact, the encoding j from HL configurations to LL configurations defined by the injection (HL configurations are a subset of LL configurations) is fully abstract.

Corollary 1 (Full abstraction). *Let j be the injection from HL (consistent) configurations to LL configurations and let M, N be two HL configurations. Then we have $j(M)\ {}_{LL}{\approx}^c_{LL}\ j(N)$ iff $M\ {}_{HL}{\approx}^c_{HL}\ N$.*

Proof. From Theorem 3 we have $M\ {}_{HL}{\approx}^c_{LL}\ j(M)$ and $N\ {}_{HL}{\approx}^c_{LL}\ j(N)$. The thesis follows by transitivity. □

The results above ensure that the loss of atomicity in rollback preserves the reachability of configurations yet does not make undesired configurations reachable.

5 Related Work and Conclusion

We have introduced in this paper a fine-grained undo capability for the asynchronous higher-order π-calculus, in the form of a rollback primitive. We present a simple but non-trivial high-level semantics for rollback, and we prove it both sound (rolling back brings a concurrent program back to a state that is a proper antecedent of the current one) and complete (rolling back can reach all antecedent states of the current one). We also present a lower-level distributed semantics for rollback, which we prove to be fully abstract with respect to the high-level one. The reversibility apparatus we exploit to support our rollback primitive is directly taken from our reversible HOπ calculus [9].

Undo or rollback capabilities in programming languages have been the subject of numerous previous works and we do not have the space to review them here; see [10] for an early survey in the sequential setting. Among the recent works that have considered undo or rollback capabilities for concurrent program execution, we can single out [3] where logging primitives are coupled with a notion of process group to serve as a basis for defining transaction abstractions, [12] which introduces a checkpoint abstraction for functional programs, and [7] which extends the actor model with constructs to create globally-consistent checkpoints.

Compared to these works, our rollback primitive brings immediate benefits: it provides a general semantics for undo operations which is not provided in [3]; thanks to the fine-grained causality tracking implied by our reversible substrate, our roll-π calculus does not suffer from uncontrolled cascading rollbacks (domino effect) which may arise with [12], and, in contrast to [7], provides a built-in guarantee that, in failure-free computations, rollback is always possible and reaches a consistent state (soundness of backward reduction).

Our low-level semantics for rollback, being a first refinement towards an implementation, is certainly related to distributed checkpoint and rollback schemes, in particular to the causal logging schemes discussed in the survey [6]. A thorough analysis of this relationship must be left for further study, however, as it requires a proper modeling of site and communication failures, as well as an explicit model for persistent data.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1) (2004)
2. Bennett, C.H.: Notes on the history of reversible computation. IBM Journal of Research and Development 32(1) (1988)
3. Chothia, T., Duggan, D.: Abstractions for fault-tolerant global computing. Theor. Comput. Sci. 322(3) (2004)
4. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
5. Danos, V., Krivine, J.: Transactions in RCCS. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 398–412. Springer, Heidelberg (2005)
6. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3) (2002)
7. Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: Proc. of POPL 2005. ACM, New York (2005)
8. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling reversibility in higher-order pi (TR), http://www.cs.unibo.it/~lanese/publications/fulltext/TR-rollpi.pdf.gz
9. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversing higher-order pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010)
10. Leeman, G.B.: A formal approach to undo operations in programming languages. ACM Trans. Program. Lang. Syst. 8(1) (1986)
11. Sangiorgi, D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD thesis CST–99–93, University of Edinburgh (1992)
12. Ziarek, L., Jagannathan, S.: Lightweight checkpointing for concurrent ML. J. Funct. Program. 20(2) (2010)

A Connector Algebra for P/T Nets Interactions*

Roberto Bruni[1], Hernán Melgratti[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Departamento de Computación, Universidad de Buenos Aires - Conicet, Argentina

Abstract. A quite flourishing research thread in the recent literature on component-based system is concerned with the algebraic properties of various kinds of connectors for defining well-engineered systems. In a recent paper, an algebra of stateless connectors was presented that consists of five kinds of basic connectors, plus their duals. The connectors can be composed in series or in parallel and employing a simple 1-state buffer they can model the coordination language Reo. Pawel Sobocinski employed essentially the same stateful extension of connector algebra to provide semantics-preserving mutual encoding with some sort of elementary Petri nets with boundaries. In this paper we show how the tile model can be used to extend Sobocinski's approach to deal with P/T nets, thus paving the way towards more expressive connector models.

1 Introduction

It is now widely acclaimed that software architectures are centred around three main kinds of elements, namely processing elements (also called *components*), data elements and connecting elements (also called *connectors*) [18]. The idea is to assemble simple, separately developed, components that exchange data items by synthesizing the appropriate glue code, i.e., by linking components via connectors. Connectors must take care of all those aspects that lie outside of the scopes of individual components. Thus, connectors are first class entities and assessing rigorous mathematical theories for them is of crucial relevance for the analysis of component-based systems.

Connectors can live at different levels of abstraction (architecture, software, processes) and several kinds of connectors have been studied in the literature [1, 13, 8, 5, 4]. Here we focus on the approach initiated in [7] and continued in [8], where a basic algebra of *stateless connectors* was presented. It consists of five kinds of basic connectors (plus their duals), namely symmetry, synchronization, mutual exclusion, hiding and inaction. The connectors can be composed in series or in parallel and the resulting circuits are equipped with a normal form axiomatization. These circuits are quite expressive: they can model the coordination aspects of the architectural design language CommUnity [13] and, using in addition a simple 1-state buffer, the coordination language Reo [1] (see [2]).

* Research supported by the EU Integrated Project 257414 ASCENS, the Italian MIUR Project IPODS (PRIN 2008), ANPCyT Project BID-PICT-2008-00319, and UBACyT 20020090300122.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 312–326, 2011.

In [22], Pawel Sobocinski employed essentially the same stateful extension of the connector algebra to compose Condition-Event (C/E) Petri nets (with consume/produce loops). Surprisingly enough, the proposed operations are quite different than those of CCS-like process calculi usually employed for making Petri nets compositional, and closer to approaches like Reo and the *tile model* [14]. Technically speaking, the contribution in [22] can be summarized as follows. C/E nets with boundaries are first introduced that can be composed in series and in parallel and come equipped with a bisimilarity semantics. Then, a suitable instance of the *wire calculus* from [21] is presented, called *Petri calculus*, that roughly models circuit diagrams with one-place buffers and interfaces. The first result enlightens a tight semantics correspondence: it is shown that a Petri calculus process can be defined for each net such that the translation preserves and reflects the semantics. The second result provides the converse translation, from Petri calculus to nets. Unfortunately, some problems arise in this direction that complicate a compositional definition of the encoding: Petri calculus processes must be normalized before the translation via a set of transformation rules that add new buffers to the circuit (and thus new places to the net).

In this paper we show how to exploit the tile model to provide an answer to the challenge posed by Sobocinski by the end of [22], namely to extend the correspondence result to deal with Place/Transition Petri nets with boundaries (where places can store more than one token and where arcs are weighted). Moreover, we are able to provide a more elegant (compositional) translation from the tile model to P/T nets that neither involves normalizing transformation, nor introduces additional places. The technical key to achieve the main result is the monoidality of the so-called vertical composition of tiles. For this reason, we conjecture that our results are unlikely to be achieved in the framework of [22], unless analogous structure is over-imposed on the Petri calculus.

Structure of the paper In § 2 we recall the basic notions of tile systems, Petri nets and [22]. In § 3 we introduce *P/T nets with boundaries.* In § 4 we define a tile system, called Petri tile calculus, where P/T nets with boundaries can be encoded via a semantics-preserving and -reflecting translation. In § 5 we show that the converse encoding is also possible and that it can be defined in a compositional way. This establishes a tight correspondence result: the Petri tile calculus and P/T nets with boundaries have the same expressive power. Related work is discussed in § 6 and some concluding remarks can be found in § 7.

2 Background

2.1 The Tile Model

The *tile model* [14] offers a flexible, rule-based semantic setting for concurrent systems [17,12,10,9,2]. A tile $A : s \xrightarrow[b]{a} t$ is a rewrite rule stating that the *initial configuration* s can evolve to the *final configuration* t via A, producing the *effect* b; but the step is allowed only if the 'arguments' of s can contribute by producing

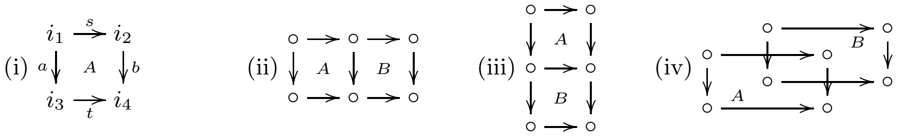


Fig. 1. Examples of tiles and their composition

a, which acts as the *trigger* of A (see Fig. 1(i)). Triggers and effects are called *observations* and tile vertices are called *interfaces*.

Tiles can be composed horizontally, in parallel, or vertically. The horizontal composition $A;B$ coordinates the evolution of the initial configuration of A with that of B, 'synchronizing' their rewrites (see Fig. 1(ii)). The vertical composition $A*B$ is the sequential composition of computations (see Fig. 1(iii)). The parallel composition $A \otimes B$ builds concurrent steps (see Fig. 1(iv)).

Roughly, the semantics of concurrent systems can be expressed via tiles when: i) configurations are equipped with sequential composition $s;t$ (defined when the output interface of s matches the input interface of t), with identities for each interface and with a monoidal tensor product $s \otimes t$ (associative, with unit and distributing over sequential composition); ii) observations have analogous structure $a;b$ and $a \otimes b$; iii) the interfaces of configurations and of observations are the same. Technically, we require that configurations and observations form two monoidal categories $\mathcal{H}$ and $\mathcal{V}$ with the same underlying set of objects.

Definition 1 (Tile system). *A* tile system *is a tuple* $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ *where* $\mathcal{H}$ *and* $\mathcal{V}$ *are monoidal categories over the same set of objects,* N *is the set of rule names and* $R\colon N \to \mathcal{H} \times \mathcal{V} \times \mathcal{V} \times \mathcal{H}$ *is a function such that for all* $A \in N$*, if* $R(A) = \langle s, a, b, t \rangle$*, then the sources and targets of* s, a, b, t *match as in Fig. 1(i).*

The categories $\mathcal{H}$ and $\mathcal{V}$ are typically those freely generated from some (sorted, hyper-)signatures $\Sigma_{\mathcal{H}}$ and $\Sigma_{\mathcal{V}}$, i.e., from sorted families of symbols $f\colon \tau_i \to \tau_o$. Their objects are words on the sort alphabet S, which must be common to the signatures $\Sigma_{\mathcal{H}}$ and $\Sigma_{\mathcal{V}}$, i.e., $\tau_i, \tau_o \in S^*$. Identity arrows $id_\tau : \tau \to \tau$ (and possibly other auxiliary arrows) are introduced by the free construction.

Our main contribution exploits the fact that, by the functoriality of the monoidal product imposed by the free construction, we have $(f \otimes g);(f' \otimes g') = (f;f') \otimes (g;g')$ for any arrows (either all configurations or all observations) f, f', g, g' such that $f;f'$ and $g;g'$ are well-defined. In particular, for $a : \tau_i \to \tau_o$ and $a' : \tau'_i \to \tau'_o$, we have $(id_{\tau_i} \otimes a');(a \otimes id_{\tau'_o}) = a \otimes a' = (a \otimes id_{\tau'_i});(id_{\tau_o} \otimes a')$.

Like rewrite rules in rewriting logic, tiles can be seen as sequents of *tile logic*: the sequent $s \xrightarrow[b]{a} t$ is *entailed* by $\mathcal{R}$, written $\mathcal{R} \vdash s \xrightarrow[b]{a} t$, if it can be obtained by horizontal, parallel, and vertical composition of some basic tiles in R plus identities tiles $id \xrightarrow[a]{a} id$ and $s \xrightarrow[id]{id} s$ (and possibly other auxiliary tiles). The "borders" of composed sequents are defined in Fig. 2.

$$\frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[c]{b} f}{s;h \xrightarrow[c]{a} t;f}\ (\mathtt{hor}) \qquad \frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[d]{c} f}{s \otimes h \xrightarrow[b\otimes d]{a\otimes c} t \otimes f}\ (\mathtt{par}) \qquad \frac{s \xrightarrow[b]{a} t \quad t \xrightarrow[d]{c} h}{s \xrightarrow[b;d]{a;c} h}\ (\mathtt{ver})$$

Fig. 2. Main inference rules for tile logic

Tiles express the reactive behavior of configurations in terms of trigger+effect labels. In this context, the usual notion of bisimilarity is called *tile bisimilarity* ($\simeq_{\mathrm{t}}$). Note that $s \simeq_{\mathrm{t}} t$ only if s and t have the same input-output interfaces.

Definition 2 (Tile bisimilarity). *Let $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ be a tile system. A symmetric relation $\sim_{\mathrm{t}}$ on configurations is called a* tile bisimulation *if whenever $s \sim_{\mathrm{t}} t$ and $\mathcal{R} \vdash s \xrightarrow[b]{a} s'$, then t' exists such that $\mathcal{R} \vdash t \xrightarrow[b]{a} t'$ and $s' \sim_{\mathrm{t}} t'$. The largest tile bisimulation is called* tile bisimilarity *and it is denoted by $\simeq_{\mathrm{t}}$.*

Lemma 1 (cfr. [14]). *A tile system $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ enjoys the basic source property if for each $A \in N$ if $R(A) = \langle s, a, b, t \rangle$, then $s \in \Sigma_{\mathcal{H}}$.*

If a tile system $\mathcal{R}$ enjoys the basic source property, then tile bisimilarity is a congruence for $\mathcal{R}$ (w.r.t. $\_;\_$ and $\_ \otimes \_$).

2.2 Nets with Boundaries

Petri nets [19] consist of *places* (i.e. resources types), which are repositories of *tokens* (i.e., resource instances), and *transitions* that remove and produce tokens.

Definition 3 (Net). *A* net *N is a 4-tuple $N = (S_N, T_N, {}^\circ{-}_N, {-}^\circ_N)$ where S_N is the (nonempty) set of places, $\mathtt{a}, \mathtt{a}', \ldots$, T_N is the set of transitions, $\mathtt{t}, \mathtt{t}', \ldots$ (with $S_N \cap T_N = \varnothing$), and the functions ${}^\circ{-}_N, {-}^\circ_N : T_N \to \wp_{\mathrm{f}}(S_N)$ assign finite sets of places, called respectively source and target, to each transition.*

Places of a *Place/Transition net* (P/T net) can hold zero, one or more tokens and arcs are weighted. The state of a P/T net is described in terms of *markings*, i.e., (finite) multisets of tokens available in the places of the net. Given a set S, a *multiset* over S is a function $m : S \to \mathbb{N}$ (where $\mathbb{N}$ is the set of natural numbers). We write $\mathcal{M}_S$ for the set of all finite multisets over S, $\varnothing$ for the empty multiset, $s \in S$ for the singleton $\{s\}$, and ns for the multiset with n instances of s. The multiset union $\oplus$ is defined such that $(m_1 \oplus m_2)(\mathtt{a}) = m_1(\mathtt{a}) + m_2(\mathtt{a})$ for all $\mathtt{a}$.

Definition 4 (P/T net). *A* marked place / transition Petri net *(P/T net) is a tuple $N = (S_N, T_N, {}^\circ{-}_N, {-}^\circ_N, m_{0N})$ where S_N is a set of places, T_N is a set of transitions, the functions ${}^\circ{-}_N, {-}^\circ_N : T_N \to \mathcal{M}_{S_N}$ assign respectively, preset and postset to each transition, and $m_{0N} \in \mathcal{M}_{S_N}$ is the initial marking.*

We often omit subscripts when they are clear from the context and denote $\mathtt{t} \in T$ simply as ${}^\circ\mathtt{t}[\rangle\mathtt{t}^\circ$. For $U : T \to \mathbb{N}$ we let ${}^\circ U = \bigoplus_{t \in U} {}^\circ t$ and $U^\circ = \bigoplus_{t \in U} t^\circ$.

Definition 5 (Semantics). *Let* $\mathtt{t} \in T$*,* $m, m' \in \mathcal{M}_S$ *and* $U \in \mathcal{M}_T$*. We let:*

$$m[\mathtt{t}\rangle m' \overset{\mathsf{def}}{=} \forall \mathtt{a} \in S : {}^{\circ}\mathtt{t}(\mathtt{a}) \leq m(\mathtt{a}) \textit{ and } m'(\mathtt{a}) = m(\mathtt{a}) - {}^{\circ}\mathtt{t}(\mathtt{a}) + \mathtt{t}^{\circ}(\mathtt{a})$$
$$m[U\rangle m' \overset{\mathsf{def}}{=} \forall \mathtt{a} \in S : {}^{\circ}U(\mathtt{a}) \leq m(\mathtt{a}) \textit{ and } m'(\mathtt{a}) = m(\mathtt{a}) - {}^{\circ}U(\mathtt{a}) + U^{\circ}(\mathtt{a})$$

Let N, N' be P/T nets. A pair $f = (f_S : S_N \to S_{N'}, f_T : T_N \to T_{N'})$ is a *net homomorphism* from N to N' (written $f : N \to N'$) if $f_S({}^{\circ}(t)_N) = {}^{\circ}(f_T(t))_{N'}$ and $f_S((t)^{\circ}_N) = (f_T(t))^{\circ}_{N'}$. Given $f : S \to S'$ (or even $f : S \to \mathcal{M}_{S'}$) we denote the homomorphic extension of f to $\mathcal{M}_S$ with the symbol f itself, i.e., we let $f(m_1 \oplus m_2) = f(m_1) \oplus f(m_2)$ and $f(\varnothing) = \varnothing$.

Definition 6 (Causal net and process). *A net* $K = (S_K, T_K, \delta_{0K}, \delta_{1K})$ *is a* causal net *(also called* deterministic occurrence net*) if it is* acyclic *and* $\forall t_0 \in T_K, \forall t_1 \neq t_0, \delta_{iN}(t_0) \cap \delta_{iN}(t_1) = \varnothing$*, for* $i = 0, 1$*.*

A process *for a P/T net* N *is a net morphism* P *from a causal net* K *to* N*.*

Two processes $P : K \to N$ and $P' : K' \to N$ are *isomorphic*, written $P \approx P'$, if there exists a net isomorphism $\psi : K \to K'$ such that $\psi; P' = P$. We let $[P]_{\approx}$ denote the equivalence class of P w.r.t. $\approx$.

Given a process $P : K \to N$, the set of *origins* and *destinations* of P are defined as $O(P) = {}^{\circ}K \cap S_K$ and $D(P) = K^{\circ} \cap S_K$, respectively. We write ${}^{\circ}P$ and P° for the multisets denoting the initial and final markings of the process, i.e. ${}^{\circ}P = P(O(P))$ and $P^{\circ} = P(D(P))$. Moreover, as isomorphisms respect initial and final markings, we say that $O(\xi) = {}^{\circ}P$, $D(\xi) = P^{\circ}$, for $\xi = [\![P]\!]_{\approx}$.

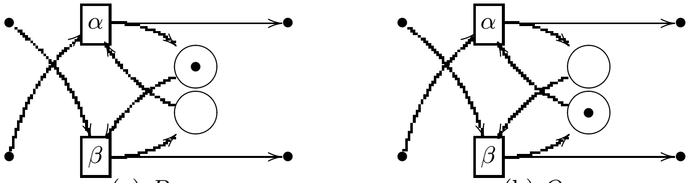
Definition 7 (Connected process). *A deterministic process* $P : K \to N$ *is a connected process if* T_K *is non-empty, and for all* $\mathtt{t}_0, \mathtt{t}_1 \in T_K$ *there exists an undirected path (w.r.t. the flow relation) connecting* $\mathtt{t}_0$ *and* $\mathtt{t}_1$*.*

The remaining of this section recalls the composable nets proposed in [22]. Due to space limitation, we refer to [22] for a detailed presentation. In the following we let $\underline{k}$, $\underline{l}$, $\underline{m}$, $\underline{n}$ range over finite ordinals: $\underline{n} \overset{\mathsf{def}}{=} \{0, 1, \ldots, n-1\}$.

Definition 8 (Nets with boundaries). *Let* $m, n \in \mathbb{N}$*. A net with boundaries* $N : m \to n$ *is a tuple* $N = (S, T, {}^{\circ}-, -^{\circ}, {}^{\bullet}-, -^{\bullet}, m_0)$ *where* $(S, T, {}^{\circ}-, -^{\circ}, m_0)$ *is a net and functions* ${}^{\bullet}- : T \to \wp_{\mathrm{f}}(\underline{m})$ *and* $-^{\bullet} : T \to \wp_{\mathrm{f}}(\underline{n})$ *assign transitions to the left and right boundaries of* N*.*

Figure 3 shows two different nets with boundaries. Places are circles and a marking is represented by the presence or absence of tokens; rectangles are transitions and arcs stand for pre and postset relations. The left interface (right interface) is depicted by points situated on the left (respectively, on the right). Figure 3(a) shows the net $P : 2 \to 2$ containing two places, two transitions and one token. Net Q is similar to P, but has a different initial marking.

Nets with boundaries can be composed in parallel and in series. Given $N : m \to n$ and $M : k \to l$, their tensor product is the net $N \otimes M : m + k \to n + l$ whose sets of places and transitions are the disjoint union of the corresponding



(a) P. (b) Q.

Fig. 3. Two nets with with boundaries

sets in N and M, whose maps ${}^\circ-, -^\circ, {}^\bullet-, -^\bullet$ are defined according to the maps in N and M and whose initial marking is $m_{0N} \oplus m_{0M}$. Intuitively, the tensor product corresponds to put the nets N and M side-by-side.

The sequential composition $N; M : m \to k$ of $N : m \to n$ and $M : n \to k$ is slightly more involved. Intuitively, transitions attached to the left or right boundaries can be seen as transition fragments, that can be completed by attaching other complementary fragments to that boundary. When two transition fragments in N share a boundary node, then they are two mutually exclusive options for completing a fragment of M attached to the same boundary node. Thus, the idea is to combine the transitions of N with that of M when they share a common boundary, as if their firings were synchronized. As in general several combinations are possible, only minimal synchronizations are selected, because the other can be recovered as concurrent firings.

2.3 Petri Calculus

Petri Calculus [22] extends the calculus of stateless connectors [8] with one-place buffers. Terms of the Petri Calculus are defined by the grammar in Fig. 4.

Any term has a unique associated sorting of the form (k, l) with $k, l \in \mathbb{N}$, that intuitively declare the left (input) interface and the right (output) interface. The type of constants are as follows: $\bigcirc$, $\odot$, and I have type $(1, 1)$, $\mathsf{X} : (2, 2)$, ∇ and $\wedge$ have type $(1, 2)$ and their duals Δ and $\vee$ have type $(2, 1)$, $\perp$ and $\downarrow$ have type $(1, 0)$ and their duals $\top$ and $\uparrow$ have type $(0, 1)$.

Inference rules for sorts are in Fig. 5. Operational semantics is defined by the rules in Fig. 6, where the labels of transitions are strings of 0 and 1, all transitions are sort-preserving, and if $P \xrightarrow[b]{a} Q$ with $P, Q : (n, m)$, then $|a| = n$ and $|b| = m$. Notably, bisimilarity induced by such a transition system is a congruence.

For example, let $P \stackrel{\mathsf{def}}{=} (\nabla \oplus \nabla); (\odot \oplus \mathsf{X} \oplus \bigcirc); (\Delta \oplus \Delta)$ and $Q \stackrel{\mathsf{def}}{=} (\nabla \oplus \nabla); (\bigcirc \oplus \mathsf{X} \oplus \odot); (\Delta \oplus \Delta)$. Clearly both P and Q have type $(2, 2)$. The only moves for P are $P \xrightarrow[00]{00} P$ and $P \xrightarrow[10]{01} Q$ while the only moves for Q are $Q \xrightarrow[00]{00} Q$ and $Q \xrightarrow[01]{10} P$. It is immediate to note that P and Q are the terms corresponding to the nets in Fig. 3 and that Q is bisimilar to $\mathsf{X}; P; \mathsf{X}$.

A close correspondence between nets with boundaries and Petri calculus terms is established in [22]. First, it is shown that any net $N : m \to n$ with initial marking X can be associated with a term $T_{N,X} : (m, n)$ that preserves and reflects the semantics of N. Conversely, for any term $t : (m, n)$ of the Petri

$$P ::= \ \bigcirc \mid \odot \mid \mathsf{I} \mid \mathsf{X} \mid \nabla \mid \Delta \mid \bot \mid \top \mid \wedge \mid \vee \mid \downarrow \mid \uparrow \mid P \oplus P \ \mid \ P;P$$

Fig. 4. Petri calculus grammar

$$\frac{\vdash P : (k,l) \quad \vdash R : (m,n)}{\vdash P \oplus R : (k+m, l+n)} \qquad \frac{\vdash P : (k,n) \quad \vdash R : (n,l)}{\vdash P;R : (k,l)}$$

Fig. 5. Sort inference rules

$$\bigcirc \xrightarrow[0]{1} \odot \quad \odot \xrightarrow[1]{0} \bigcirc \quad \odot \xrightarrow[1]{1} \odot \qquad \mathsf{I} \xrightarrow[1]{1} \mathsf{I} \qquad \nabla \xrightarrow[11]{1} \nabla \quad \Delta \xrightarrow[1]{11} \Delta \qquad \bot \xrightarrow{1} \bot \quad \top \xrightarrow[1]{} \top$$

$$\frac{a,b \in \{0,1\}}{\mathsf{X} \xrightarrow[ba]{ab} \mathsf{X}} \quad \frac{a \in \{0,1\}}{\wedge \xrightarrow[(1-a)a]{1} \wedge} \quad \frac{a \in \{0,1\}}{\vee \xrightarrow[1]{(1-a)a} \vee} \quad \frac{P \xrightarrow[c]{a} Q \quad R \xrightarrow[b]{c} S}{P;R \xrightarrow[b]{a} Q;S} \quad \frac{P \xrightarrow[b]{a} Q \quad R \xrightarrow[d]{c} S}{P \otimes R \xrightarrow[bd]{ac} Q \otimes S} \quad \frac{P : (m,n)}{P \xrightarrow[0^n]{0^m} P}$$

Fig. 6. Operational semantics for the Petri Calculus

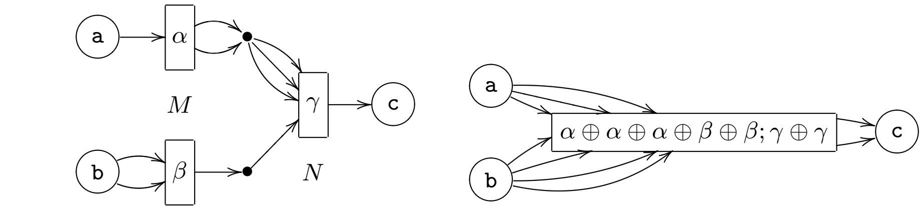
calculus there exists a bisimilar net $N_t : m \to n$. Due to space limitation we omit details here and refer the interested reader to [22]. We remark here that the encoding from Petri calculus to nets with boundaries has been defined only for a subclass of terms, called *composable*, which have been characterized structurally. The correspondence result holds because it is possible to transform any term t into a bisimilar composable term t' by using a suitable set of normalizing axioms that may add new filled places $\odot$ to t' and therefore to $N_{t'}$.

3 P/T Nets with Boundaries

This section defines a version of composable P/T following the line proposed in [22]. It is immediate to check that our definitions generalize the ones in [22].

Definition 9 (P/T net with boundaries). *Let $m, n \in \mathbb{N}$. A P/T net with boundaries $N : m \to n$ is a tuple $N = (S, T, {}^\circ-, -^\circ, {}^\bullet-, -^\bullet, m_0)$ such that $(S, T, {}^\circ-, -^\circ, m_0)$ is a P/T net and functions ${}^\bullet- : T \to \mathcal{M}_{\underline{m}}$ and $-^\bullet : T \to \mathcal{M}_{\underline{n}}$ assign transitions to the left and right boundaries of N.*

The notion of net homomorphism extends to P/T nets with the same boundaries: given $N, N' : m \to n$ is a pair $f = (f_S : S_N \to S_{N'}, f_T : T_N \to T_{N'})$ s.t. $f_S({}^\circ(t)_N) = {}^\circ(f_T(t))_{N'}$, $f_S((t)^\circ_N) = (f_T(t))^\circ_{N'}$, $f_S({}^\bullet(t)_N) = {}^\bullet(f_T(t))_{N'}$ and $f_S((t)^\bullet_N) = (f_T(t))^\bullet_{N'}$. A homomorphism is an isomorphism if its two components are bijections. We write $N \cong M$ if there is an isomorphism from N to M.



(a) Two P/T with boundaries M and N.

(b) Composition $M;N$.

Fig. 7. Composition of P/T with boundaries

In what follows we will use + to denote the disjoint union of sets. Given two functions $f : M \to R$ and $g : N \to R$, we also use f to denote the obvious extension $f' : M \to R + S$ and $f + g$ for the obvious function $h : M + N \to R$.

Definition 10 (Synchronization). *A synchronization between nets $M : l \to m$ and $N : m \to n$ is a connected process P of the following net $N + M = (S_N + S_M + \underline{l} + \underline{m} + \underline{n}, T_M + T_N, ({}^\circ -_M \cup {}^\bullet -_M) + ({}^\circ -_N \cup {}^\bullet -_N), (-^\circ_M \cup -^\bullet_M) + (-^\circ_N \cup -^\bullet_N))$ s.t. ${}^\circ P \cap \underline{m} = P^\circ \cap \underline{m} = \varnothing$.*

We write a synchronization $P : K \to M + N$ as $(U, V)_P$ with $U \in \mathcal{M}_{T_M}$, $V \in \mathcal{M}_{T_N}$ and $P(T_K) = U \oplus V$. We usually omit subscript P from $(U, V)_P$. Let

$$T_{M;N} \overset{\mathsf{def}}{=} \{(U, V)_P \mid (U, V)_P \text{ a synchronization of } M \text{ and } N\}$$

Define ${}^\circ -, -^\circ : T_{M;N} \to \mathcal{M}_{P_M + P_N}$ such that ${}^\circ(U, V)_P = {}^\circ P$ and $(U, V)^\circ_P = P^\circ$. Define ${}^\bullet - : T_{M;N} \to \underline{l}$ by ${}^\bullet(U, V) = {}^\bullet U$ and $-^\bullet : T_{M;N} \to \underline{n}$ by $(U, V)^\bullet = V^\bullet$.

Definition 11. *The composition of M and N, written $M;N : l \to n$, has: $P_{M;N} = P_M + P_N$; $T_{M;N}$ as the set of transitions defined above; ${}^\circ -, -^\circ : T_{M;N} \to \mathcal{M}_{P_M + P_N}$; and $m_{0M;N} = m_{0M} \oplus m_{0N}$.*

Figure 7(b) shows the sequential composition of the nets M and N depicted in Fig. 7(a). As for nets with boundaries, it can be shown that sequential composition is associative up-to isomorphism, i.e., given three nets $L : k \to l$, $M : l \to m$ and $N : m \to n$ then $(L; M); N \cong L; (M; N)$. Parallel composition (or tensor product) is defined analogously to nets with boundaries (see Section 2.2).

For any $k \in \mathbb{N}$, there is a bijection $\ulcorner - \urcorner : \mathcal{M}_{\underline{k}} \to \mathbb{N}^k$ with $\ulcorner m \urcorner_i = m(i)$.

Definition 12 (Semantics). *Let $N : m \to n$ and $m_0, m_0' \in \mathcal{M}_{P_N}$. We write*

$$(N, m_0) \xrightarrow[\beta]{\alpha} (N, m_0') \overset{\mathsf{def}}{=} \exists U_0 \dots U_k \in \mathcal{M}_{T_N} \textit{ s.t.}$$
$$m_0[U_0\rangle \dots [U_k\rangle m_0', \alpha = \ulcorner {}^\bullet U \urcorner \textit{ and } \beta = \ulcorner U^\bullet \urcorner \textit{ with } U = \oplus_i U_i$$

Lemma 2. *Let $M : l \to m$ and $N : m \to n$. $(M, m_0) \xrightarrow[\gamma]{\alpha} (M, m_0')$ and $(N, m_1) \xrightarrow[\beta]{\gamma} (N, m_1')$ for some γ iff $(M, m_0); (N, m_1) \xrightarrow[\beta]{\alpha} (M, m_0'); (N; m_1')$.*

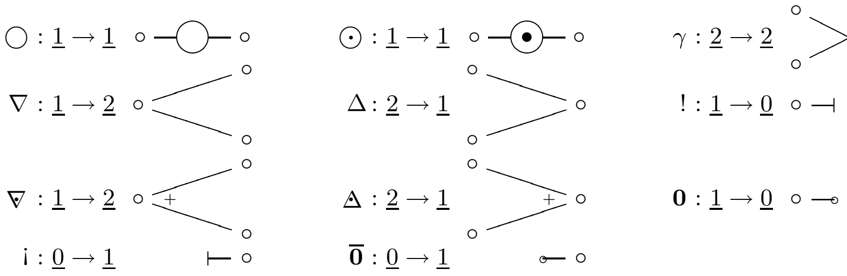

$\bigcirc : \underline{1} \to \underline{1}$
$\odot : \underline{1} \to \underline{1}$
$\gamma : \underline{2} \to \underline{2}$
$\nabla : \underline{1} \to \underline{2}$
$\Delta : \underline{2} \to \underline{1}$
$! : \underline{1} \to \underline{0}$
$\dot{\nabla} : \underline{1} \to \underline{2}$
+
$\dot{\Delta} : \underline{2} \to \underline{1}$
+
$\mathbf{0} : \underline{1} \to \underline{0}$
$¡ : \underline{0} \to \underline{1}$
$\overline{\mathbf{0}} : \underline{0} \to \underline{1}$

Fig. 8. Horizontal signature

4 Petri Tile Calculus

The horizontal signature of the tile system for modeling P/T nets with boundaries includes operators for basic places, tokens, mergers and replicators. The full horizontal signature of our tile system is in Fig. 8, together with the diagrammatic representation of each operator, which makes evident the duality of certain operators (e.g. ∇ and Δ). Notably, the symmetry γ is self-dual. We remark that the identity $id : \underline{1} \to \underline{1}$ is not part of the signature and it is added by the free construction to the monoidal category of configurations.

We find it useful to introduce the following notation, to be used in several definitions and lemmas: let $s : \underline{1} \to \underline{1}$ and $t : \underline{1} \to \underline{1}$, then we write $\langle s, t\rangle : \underline{1} \to \underline{1}$ as a shorthand for $\dot{\nabla}; (s \otimes t); \dot{\Delta}$. Moreover, we introduce a few (inductively defined) terms to be used in the following sections. (Their dual versions are defined analogously, but not given explicitly.) For $n > 0$ and $k \geq 0$:

$$\begin{array}{lll}
\bigcirc_n = \bigotimes_n \bigcirc & \odot_n = \bigotimes_n \odot & \gamma_1 = \gamma \quad \gamma_{n+1} = (\gamma_n \otimes id); (id_n \otimes \gamma) \\
id_n = \bigotimes_n id & !_n = \bigotimes_n ! & \nabla_1 = \nabla \quad \nabla_{n+1} = (\nabla \otimes \nabla_n); (id \otimes \gamma_n \otimes id_n) \\
\mathbf{0}_n = \bigotimes_n \mathbf{0} & d_n = ¡_n; \nabla_n & \dot{\nabla}_1 = \dot{\nabla} \quad \dot{\nabla}_{n+1} = (\dot{\nabla} \otimes \dot{\nabla}_n); (id \otimes \gamma_n \otimes id_n) \\
 & & \nabla^0_n = !_n \quad \nabla^{k+1}_n = \nabla_n; (\nabla^k_n \otimes id_n) \\
 & & \dot{\nabla}^0_n = \mathbf{0}_n \quad \dot{\nabla}^{k+1}_n = \dot{\nabla}_n; (\dot{\nabla}^k_n \otimes id_n)
\end{array}$$

Observations are strings $a \in \mathbb{N}^*$, with $a : \underline{|a|} \to \underline{|a|}$, e.g. $n : \underline{1} \to \underline{1}$ for any $n \geq 0$. Vertical composition of observations is defined as $(a_0 \ldots a_n); (b_0 \ldots b_n) = (a_0 + b_0) \ldots (a_n + b_n)$. The empty string ϵ is the unit, and the identity for each $\underline{n}$ is the string of 0s with length n.

Definition 13 (Petri tile calculus). *The Petri tile calculus is the tile system consisting of the basic tiles in Fig. 9(a) (duals omitted for brevity).*

We remind that for any configuration $t : \underline{m} \to \underline{n}$ we have always the *vertical indentity* tile $t \xrightarrow[b]{a} t$ where a (resp. b) is the string of 0s with length m (resp. n).

The properties of γ, ∇, Δ, $\dot{\nabla}$, $\dot{\Delta}$, $!$, $¡$, $\mathbf{0}$ and $\overline{\mathbf{0}}$ are well-known and have been studied in [8] under the name *stateless connectors*. There is one technical

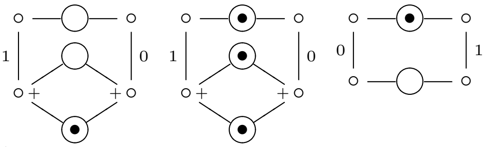
$$\bigcirc \xrightarrow[0]{1} \langle \bigcirc, \odot \rangle \qquad \nabla \xrightarrow[11]{1} \nabla \qquad ! \xrightarrow[\epsilon]{1} !$$

$$\odot \xrightarrow[0]{1} \langle \odot, \odot \rangle \qquad \dot{\nabla} \xrightarrow[10]{1} \dot{\nabla} \qquad \mathbf{0} \xrightarrow[\epsilon]{0} \mathbf{0}$$

$$\odot \xrightarrow[1]{0} \bigcirc \qquad \dot{\nabla} \xrightarrow[01]{1} \dot{\nabla}$$

$$\gamma \xrightarrow[yx]{xy} \gamma \text{ for } x, y \in \{1, 0\}$$

(a) Basic connectors



(b) Graphical representation of stateful basic tiles

Fig. 9. Tiles for ordinary connectors

difference, however, namely that the observation 0 is here the vertical identity, whereas in [8] it was interpreted as forced inaction. This difference has some consequences at the semantic level w.r.t. tile bisimilarity, as discussed below. One key property for them is that if $s \xrightarrow[\beta]{\alpha} t$ for s one of the above connector, then $t = s$ and α and β are suitably constrained. For example, $\gamma \xrightarrow[\beta]{\alpha} t$ iff $t = \gamma$, $\alpha = a_0 a_1$ and $\beta = a_1 a_0$. Moreover, identity id can also be regarded as a stateless connector, which are closed under sequential and monoidal composition.

There are some interesting laws that hold up-to-bisimilarity. For example, we remind that $\gamma; \gamma \simeq_{\mathrm{t}} id$ and that for all $s, t : \underline{1} \to \underline{1}$ we have $\gamma; (s \otimes t) \simeq_{\mathrm{t}} (t \otimes s); \gamma$. Below we note just a few useful equivalences (we remind that that their duals hold too and that $\otimes$ has precedence over ;):

$$\begin{array}{llll}
\nabla; (id \otimes \nabla) \simeq_{\mathrm{t}} \nabla; (\nabla \otimes id) & \nabla; (id \otimes !) \simeq_{\mathrm{t}} id & \nabla; \gamma \simeq_{\mathrm{t}} \nabla & \overline{\mathbf{0}}; \nabla \simeq_{\mathrm{t}} \overline{\mathbf{0}} \otimes \overline{\mathbf{0}} \\
\dot{\nabla}; (id \otimes \dot{\nabla}) \simeq_{\mathrm{t}} \dot{\nabla}; (\dot{\nabla} \otimes id) & \dot{\nabla}; (id \otimes \mathbf{0}) \simeq_{\mathrm{t}} id & \dot{\nabla}; \gamma \simeq_{\mathrm{t}} \dot{\nabla} & ¡; \dot{\nabla} \simeq_{\mathrm{t}} ¡ \otimes ¡ \\
\Delta; \nabla \simeq_{\mathrm{t}} id \otimes \nabla; \Delta \otimes id & \nabla; \Delta \simeq_{\mathrm{t}} id \simeq_{\mathrm{t}} \dot{\nabla}; \dot{\Delta} & & \dot{\Delta}; \dot{\nabla} \simeq_{\mathrm{t}} \dot{\nabla}_2; \dot{\Delta} \otimes \dot{\Delta}
\end{array}$$

Notably, this last bisimilarity does not hold in [8]. Moreover, contrary to [8], in our case $\dot{\nabla}; \Delta \not\simeq_{\mathrm{t}} \mathbf{0}; \overline{\mathbf{0}}$, because $\dot{\nabla}; \Delta \xrightarrow[11]{2} \dot{\nabla}; \Delta$ while $\mathbf{0}; \overline{\mathbf{0}}$ can only stay idle.

Since the Petri tile calculus satisfies the basic source property, then tile bisimilarity is a congruence and we can safely apply all the above equivalences within any larger term guaranteeing that the original term is bisimilar to the result, e.g., $\nabla; (\nabla; \Delta) \otimes id; \Delta \simeq_{\mathrm{t}} \nabla; id \otimes id; \Delta \simeq_{\mathrm{t}} \nabla; \Delta \simeq_{\mathrm{t}} id$.

Lemma 3. *Let $s, t, p : \underline{1} \to \underline{1}$. Then $\langle s, t \rangle \simeq_{\mathrm{t}} \langle t, s \rangle$ and $\langle s, \langle t, p \rangle \rangle \simeq_{\mathrm{t}} \langle \langle s, t \rangle, p \rangle$.*

4.1 P/T Nets as Tiles

Consider a P/T net $N = (S, T, {}^{\circ}-, -^{\circ}, m)$. W.l.o.g. assume $S_N = \underline{p}$ and $T_N = \underline{t}$. Then, the marking is encoded as follows

$$[\![m]\!]_m = \bigotimes_{i<p} [\![m(i)]\!] : \underline{p} \to \underline{p}$$

where $\llbracket n \rrbracket : \underline{1} \to \underline{1}$ with $n \in \mathbb{N}$ is defined as follows

$$\llbracket 0 \rrbracket = \bigcirc \qquad \llbracket k+1 \rrbracket = \langle \llbracket k \rrbracket, \odot \rangle = \dot{\nabla} ; (\llbracket k \rrbracket \otimes \odot) ; \dot{\Delta}$$

Intuitively $\bigcirc$ represent a place and $\odot$ a token, so that, e.g., $\llbracket 2 \rrbracket = \langle \langle \bigcirc, \odot \rangle, \odot \rangle$ represents two tokens in a place. Note however that when a token is consumed, then the space occupied is left vacant, e.g., $\llbracket 2 \rrbracket \xrightarrow[1]{0} t$ with either $t = \langle \langle \bigcirc, \odot \rangle, \bigcirc \rangle$ or $t = \langle \langle \bigcirc, \bigcirc \rangle, \odot \rangle$. Nevertheless, we note that $\langle \_, \_ \rangle$ is associative and commutative up to tile bisimilarity and thus $\langle \langle \bigcirc, \odot \rangle, \bigcirc \rangle \simeq_{\mathrm{t}} \langle \langle \bigcirc, \bigcirc \rangle, \odot \rangle$. Moreover, any number of empty places combined in mutual exclusion is bisimilar to one empty place, e.g., $\langle \bigcirc, \bigcirc \rangle \simeq_{\mathrm{t}} \bigcirc$. Thus, e.g., $\langle \langle \bigcirc, \odot \rangle, \bigcirc \rangle \simeq_{\mathrm{t}} \llbracket 1 \rrbracket$.

Next, we give a structural characterization of the configurations reachable from $\llbracket n \rrbracket$. For $n, a, b \in \mathbb{N}$ such that $n + a \geq b$, we let $\llbracket n \rrbracket^a_b$ denote the set of terms obtained from $\llbracket n \rrbracket$ by inserting a instances of $\odot$ and replacing b instances of $\odot$ with $\bigcirc$. Formally, we let $\langle n \rangle$ denote the set of any combination of n tokens, i.e., $\langle 1 \rangle = \{ \odot \}$ and $\langle n \rangle = \{ \langle t_1, t_2 \rangle \mid \exists n_1, n_2 > 0, n_1 + n_2 = n, t_1 \in \langle n_1 \rangle, t_2 \in \langle n_2 \rangle \}$ for $n > 1$. Similarly, we let $\langle n \rangle_0 = \langle n \rangle$ and $\langle 1 \rangle_1 = \{ \bigcirc \}$ and $\langle n \rangle_b = \{ \langle t_1, t_2 \rangle \mid \exists n_1, n_2 > 0, n_1 + n_2 = n, \exists b_1 \leq n_1, \exists b_2 \leq n_2, b_1 + b_2 = b, t_1 \in \langle n_1 \rangle_{b_1}, t_2 \in \langle n_2 \rangle_{b_2} \}$ for $n \geq b > 0$ and $n \neq 1$. Finally, we let $\llbracket n \rrbracket^0_0 = \{ \llbracket n \rrbracket \}$, $\llbracket 0 \rrbracket^{a+1}_b = \llbracket 1 \rrbracket^a_b$ and $\llbracket n+1 \rrbracket^a_b = \{ \langle t_1, t_2 \rangle \mid \exists a_1, a_2, a_1 + a_2 = a, \exists b_1, b_2, b_1 + b_2 = b, n + a_1 \geq b_1, 1 + a_2 \geq b_2, t_1 \in \llbracket n \rrbracket^{a_1}_{b_1}, t_2 \in \langle 1 + a_2 \rangle_{b_2} \rangle \}$. We extend this notation to markings, by letting

$$\llbracket m, m_a, m_b \rrbracket_m = \left\{ \bigotimes\nolimits_{i<p} t_i \mid t_i \in \llbracket m(i) \rrbracket^{m_a(i)}_{m_b(i)} \right\}$$

Notably, $\llbracket n \rrbracket^a_b \simeq_{\mathrm{t}} \llbracket n + a - b \rrbracket$ and therefore for any $t \in \llbracket m, m_a, m_b \rrbracket_m$ we have $t \simeq_{\mathrm{t}} \bigotimes_{i<p} \llbracket m(i) + m_a(i) - m_b(i) \rrbracket$

Lemma 4. *Let $m_0 \in \mathcal{M}_{\underline{p}}$. $\llbracket m_0 \rrbracket_m \xrightarrow[b_0 \ldots b_{p-1}]{a_0 \ldots a_{p-1}} t$ iff $t \in \llbracket m_0, m_a, m_b \rrbracket_m$ with $m_a(i) = a_i$, $m_b(i) = b_i$ and $m_0(i) + m_a(i) \geq m_b(i)$ for $i \in \underline{p}$.*

We use λ_f with $f : \underline{k} \to \mathcal{M}_{\underline{l}}$ to denote the tile $\lambda_f : \underline{l} \to \underline{k}$ defined as follows

$$\lambda_f = \dot{\nabla}^k_l ; \bigotimes\nolimits_{j<k} \llbracket f(j) \rrbracket_\lambda : \underline{l} \to \underline{k}$$

where $\llbracket m \rrbracket_\lambda : \underline{l} \to \underline{1}$ for $m \in \mathcal{M}_{\underline{l}}$ is defined as follows

$$\llbracket m \rrbracket_\lambda = \bigotimes\nolimits_{i<l} (\dot{\nabla}^{m(i)}_1 ; \Delta^{m(i)}_1) ; \Delta^l_1$$

Lemma 5. $\lambda_f \xrightarrow[\beta]{\alpha} t$ *iff $t = \lambda_f$, $\alpha = \ulcorner a' \urcorner$, $\beta = \ulcorner b' \urcorner$ and $f(b') = a'$.*

Similarly, we use ρ_f with $f : \underline{k} \to \mathcal{M}_{\underline{l}}$ to denote the following tile $\rho_f : \underline{k} \to \underline{l}$

$$\rho_f = \bigotimes\nolimits_{j<k} \llbracket f(j) \rrbracket_\rho ; \dot{\Delta}^k_l$$

where $\llbracket m \rrbracket_\rho : \underline{1} \to \underline{l}$ is defined by $\llbracket m \rrbracket_\rho = \nabla^l_1 ; \bigotimes_{i<l} (\nabla^{m(i)}_1 ; \dot{\Delta}^{m(i)}_1)$.

Lemma 6. $\rho_f \xrightarrow[\beta]{\alpha} t$ *iff $t = \rho_f$, $\alpha = \ulcorner a' \urcorner$, $\beta = \ulcorner b' \urcorner$ and $f(a') = b'$.*

Definition 14 (Encoding). *The encoding of a P/T net with boundaries* $N = (\underline{p}, \underline{t}, {}^{\circ}-, -^{\circ}, {}^{\bullet}-, -^{\bullet}, m_0)$ *as a tile is defined as follows*

$$[\![m_0]\!]_N = (d_t \otimes \lambda_{\bullet -}); (id_t \otimes (\Delta_t; \rho_{-\circ}; [\![m_0]\!]_m; \lambda_{\circ -}; \nabla_t)); (e_t \otimes \rho_{-\bullet})$$

We write $[\![m_0, m_a, m_b]\!]_N$ to denote the class of bisimilar representations of N with marking m_1 such that $m_0(i) + m_a(i) - m_b(i) = m_1(i)$ for $i \in \underline{p}$. Formally,

$$\begin{aligned}[\![m_0, m_a, m_b]\!]_N = \{r \mid\ & m \in [\![m_0, m_a, m_b]\!]_m \text{ and} \\ & r = (d_t \otimes \lambda_{\bullet -}); (id_t \otimes (\Delta_t; \rho_{-\circ}; m; \lambda_{\circ -}; \nabla_t)); (e_t \otimes \rho_{-\bullet})\}\end{aligned}$$

We now define the Petri tile terms encoding the behaviour of the nets in Fig. 7 (for space limitation we show terms that are bisimilar to the ones generated by the encoding, but simpler). Term $M_1 = \text{¡}; \bigcirc; \nabla; \dot{\Delta}$ stands for transition α and $M_2 = \text{¡}; \bigcirc; \dot{\nabla}; \Delta$ for β. Then, $M = (M_1 \otimes M_2)$. Similarly, $N = (\dot{\nabla}^3_1 \otimes id); \Delta^4_1; \bigcirc; !$. Finally, the term for the net $M; N$ is bisimilar to $((\text{¡}; \bigcirc; \dot{\nabla}^3_1) \otimes (\text{¡}; \bigcirc; \dot{\nabla}^4_1)); \Delta^7_1; \nabla; \dot{\Delta}; \bigcirc; !$.

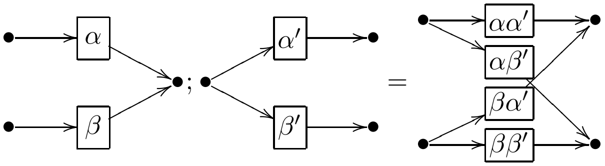
Theorem 1. *Let* N *be a finite net with boundaries, then*

- *if* $m_0 \xrightarrow[\beta]{\alpha} m_1$ *then* $\exists Q, m_a, m_b$ *s.t.* $[\![m_0]\!]_N \xrightarrow[\beta]{\alpha} Q$, $Q \in [\![m_0, m_a, m_b]\!]_N$ *and* $m_0(i) + m_a(i) - m_b(i) = m_1(i)$ *for* $i \in \underline{p}$.
- *if* $[\![m_0]\!]_N \xrightarrow[\beta]{\alpha} Q$ *then* $\exists Q, m_a, m_b, m_1$ *such that* $Q \in [\![m_0, m_a, m_b]\!]_N$, $m_0(i) + m_a(i) - m_b(i) = m_1(i)$ *for* $i \in \underline{p}$ *and* $m_0 \xrightarrow[\beta]{\alpha} m_1$.

5 Petri Tile Calculus as P/T Nets with Boundaries

The encoding of basic tiles is shown in Fig. 10. Rules are analogous to the ones proposed in [22]. In addition, our encoding is homomorphic w.r.t. ';' and $\oplus$, e.g., $\{[t_1; t_2]\} = \{[t_1]\}; \{[t_2]\}$ and $\{[t_1 \oplus t_2]\} = \{[t_1]\} \oplus \{[t_2]\}$. We remark that our encoding is compositional and we can avoid normalizing terms before encoding them. Consider the encoding of term $\dot{\nabla}; \dot{\Delta}$, which is problematic for [22]:

$\{[\dot{\nabla}; \dot{\Delta}]\} = \{[\dot{\nabla}]\}; \{[\dot{\Delta}]\} =$



When considering the string 11 over the left interface, we have the following behaviours for $\dot{\Delta}; \dot{\nabla}$: $\dot{\Delta}; \dot{\nabla} \xrightarrow[20]{11} \dot{\Delta}; \dot{\nabla}$, $\dot{\Delta}; \dot{\nabla} \xrightarrow[02]{11} \dot{\Delta}; \dot{\nabla}$ and $\dot{\Delta}; \dot{\nabla} \xrightarrow[11]{11} \dot{\Delta}; \dot{\nabla}$. It is easy to notice that $\{[\dot{\Delta}; \dot{\nabla}]\} \xrightarrow[20]{11} \{[\dot{\Delta}; \dot{\nabla}]\}$ (by firing $\alpha\alpha'$ and $\beta\alpha'$), $\{[\dot{\Delta}; \dot{\nabla}]\} \xrightarrow[02]{11} \{[\dot{\Delta}; \dot{\nabla}]\}$ (using $\alpha\beta'$ and $\beta\beta'$) and $\{[\dot{\Delta}; \dot{\nabla}]\} \xrightarrow[11]{11} \{[\dot{\Delta}; \dot{\nabla}]\}$ (by firing either $\alpha\alpha'$ and $\beta\beta'$ or $\alpha\beta'$ and $\beta\alpha'$). From this simple fact, we also learned how to "repair" the encoding in [22] (see Section 6).

Theorem 2. *For any connector* t, $t \sim \{[t]\}$.

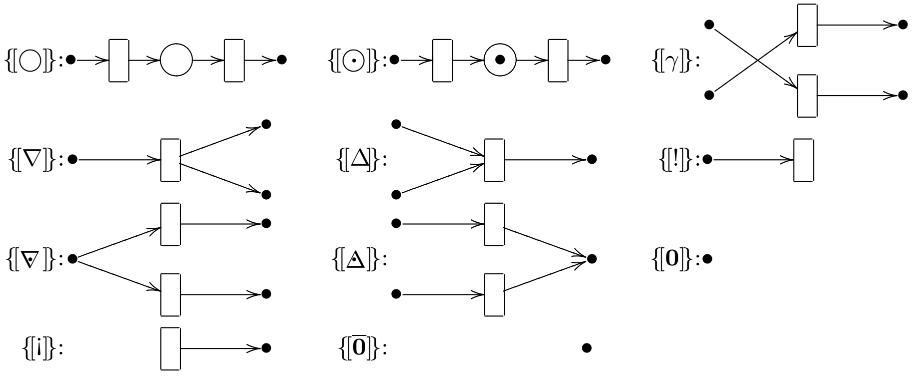


Fig. 10. Encoding of basic tiles into P/T nets

6 Related Work

Different studies about primitive forms of connectors have appeared in the literature. Due to space limitation, we will just mention the most prominent ones and we will postpone a detailed comparison to the full version of this paper. Connectors have been studied as key aspect of coordination languages like Reo [1], which provides a set of primitive connectors modelling synchronous / asynchronous / lossy channels and the asynchronous one-place buffer. Complex forms of interactions are defined as combinations of basic forms of connectors. Similarly, connectors are a fundamental notion of architectural languages. In particular, CommUnity [13] presents a formal treatment of architectural connectors in a categorical setting. A formal link between tiles and Reo and CommUnity connectors is presented in [8]. Connectors considering prioritized forms of interactions have been addressed in [5], where the relation with [8] is also mentioned (for the non prioritized case). Our approach to connectors is much indebted to [23,7].

Tiles resemble Plotkin's SOS inference rules [20], but take inspiration also from Structured Transition Systems [11] and context systems [15]. The Tile Model also extends rewriting logic [16] (in the non-conditional case) by taking into account rewrite with side effects and rewrite synchronization. While in this paper we exploit horizontal connectors only, in [9] it is shown how to benefit from the interplay of connectors in both the horizontal and vertical dimensions for defining causal semantics.

Operators of the Petri tile calculus are in one-to-one correspondence with operators of the Petri Calculus of [22]. The semantics of unary operators except for buffers (i.e., ○ and ⊙) coincide in both calculus. Rules for buffers in the Petri calculus allows for at most one token in any place while buffers in the Petri tile calculus may contain an unbounded number of tokens (this can be traced to the rules that define the semantics of ○ and ⊙). Thanks to vertical composition of tiles we can easily deal with computations that are problematic in the Petri calculus, like $\dot{\Delta}; \dot{\nabla} \xrightarrow[11]{11} \dot{\Delta}; \dot{\nabla}$, which is not possible to obtain in [22], because ∇ and Δ serializes the access to the shared interface and their vertical composition misses the functoriality axiom of tensor product. One simple fix to their counterexamples, that allows to get rid of term normalization before producing the net just consists in mapping ∇ to a net with two transitions α and β with the same left interface and distinct right interface, both having a self-loop on the same (internal) place p. This way, the problematic step $\xrightarrow[11]{11}$ is banned also from the net. If instead, one wants to avoid introducing additional places in the net, then we can just take the semantics rules for the calculus in [22] (places have capacity one, they are not unbounded) and make vertical composition monoidal, with 0 as identity. This is in fact a key semantic difference between the wire calculus [21] and the tile model. Finally, we mention that nets with boundaries are very similar to the open nets in [3].

7 Conclusions

In theoretical computer science, it is very frequent that quite different representations can be reconciled by showing that they can be mutually encoded one in the other with tight semantics correspondence: thus, in the end, they have the same expressive power and represent the same abstract concept.

In this paper, we have contributed to the above thread by showing that a suitable class of tile models, called Petri tile calculus, has the same expressive power as a composable version of Petri nets, called P/T nets with boundaries. Our result gives some insights on how to improve the analogous result in [22] (by exploiting compositionality and a simpler translation). Moreover, given the correspondence between similar tile models and other approaches to connectors [2,6,8], the algebraic properties of the tile model can serve to relate formal frameworks that are otherwise very different in style and nature (CommUnity, Reo, Petri nets), and it is now possible to establish a hierarchy of connectors just by selecting different subsets of the basic connectors from the Petri tile calculus.

In this respect, an interesting line of research is the study of prioritised version of tiles, such that local priorities can be attributed to the basic elements to guarantee that some global order of preference in the reactive behaviour is obtained. In fact global priorities are often introduced in ad hoc ways in connectors to prune some unwanted behaviours, but then it is not clear under which conditions they can be distributed over basic connectors preserving the overall behaviour. We plan to investigate suitable classes of tile models where either this is always made possible by assuming suitable structural constraints, or when non-distributable global priorities are assigned, then the problem can be identified and a relaxed assignment of priorities can be suggested.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. in Comp. Science 14(3), 329–366 (2004)

2. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for reo. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 37–55. Springer, Heidelberg (2009)
3. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processe. Math. Struct. in Comp. Science 15(1), 1–35 (2005)
4. Barbosa, M.A., Barbosa, L.S.: Specifying software connectors. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 52–67. Springer, Heidelberg (2005)
5. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Computers 57(10), 1315–1330 (2008)
6. Bliudze, S., Sifakis, J.: Causal semantics for the algebra of connectors. Formal Methods in System Design 36(2), 167–194 (2010)
7. Bruni, R., Gadducci, F., Montanari, U.: Normal forms for algebras of connection. Theor. Comput. Sci. 286(2), 247–292 (2002)
8. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theor. Comput. Sci. 366(1-2), 98–120 (2006)
9. Bruni, R., Montanari, U.: Dynamic connectors for concurrency. Theor. Comput. Sci. 281(1-2), 131–176 (2002)
10. Bruni, R., Montanari, U., Rossi, F.: An interactive semantics of logic programming. TPLP 1(6), 647–690 (2001)
11. Corradini, A., Montanari, U.: An algebraic semantics for structured transition systems and its application to logic programs. Theoret. Comput. Sci. 103, 51–106 (1992)
12. Ferrari, G.L., Montanari, U.: Tile formats for located and mobile systems. Inf. Comput. 156(1-2), 173–235 (2000)
13. Fiadeiro, J.L., Maibaum, T.S.E.: Categorical semantics of parallel program design. Sci. Comput. Program. 28(2-3), 111–138 (1997)
14. Gadducci, F., Montanari, U.: The tile model. In: Proof, Language, and Interaction, pp. 133–166. The MIT Press, Cambridge (2000)
15. Larsen, K.G., Xinxin, L.: Compositionality through an operational semantics of contexts. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 526–539. Springer, Heidelberg (1990)
16. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. 96, 73–155 (1992)
17. Montanari, U., Rossi, F.: Graph rewriting, constraint solving and tiles for coordinating distributed systems. Applied Categorical Structures 7(4), 333–370 (1999)
18. Perry, D.E., Wolf, E.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17, 40–52 (1992)
19. Petri, C.: Kommunikation mit Automaten. PhD thesis, Institut für Instrumentelle Mathematik, Bonn (1962)
20. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
21. Sobocinski, P.: A non-interleaving process calculus for multi-party synchronisation. In: ICE 2009. EPTCS, vol. 12, pp. 87–98 (2009)
22. Sobociński, P.: Representations of petri net interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 554–568. Springer, Heidelberg (2010)
23. Stefanescu, G.: Reaction and control i. mixing additive and multiplicative network algebras. Logic Journal of the IGPL 6(2), 348–369 (1998)

Vector Addition System Reversible Reachability Problem

Jérôme Leroux

LaBRI, Université Bordeaux 1, CNRS

Abstract. The reachability problem for vector addition systems is a central problem of net theory. This problem is known to be decidable but the complexity is still unknown. Whereas the problem is EXPSPACE-hard, no elementary upper bounds complexity are known. In this paper we consider the reversible reachability problem. This problem consists to decide if two configurations are reachable one from each other. We show that this problem is EXPSPACE-complete. As an application of the introduced materials we characterize the reversibility domains of a vector addition system.

1 Introduction

Vector addition systems (VASs) or equivalently Petri nets are one of the most popular formal methods for the representation and the analysis of parallel processes [EN94]. Their reachability problem is central since many computational problems (even outside the realm of parallel processes) reduce to the reachability problem. Sacerdote and Tenney provided in [ST77] a partial proof of decidability of this problem. The proof was completed in 1981 by Mayr [May81] and simplified by Kosaraju [Kos82] from [ST77, May81]. Ten years later [Lam92], Lambert provided a further simplified version based on [Kos82]. This last proof still remains difficult and the upper-bound complexity of the corresponding algorithm is just known to be non-primitive recursive. Nowadays, the exact complexity of the reachability problem for VASs is still an open-problem. The problem is known to be EXPSPACE-hard [CLM76] but even the existence of an elementary upper-bound complexity is open.

Recently, in [Ler11] we provided a new proof of the reachability problem based on the notion of *production relations* inspired by Hauschildt [Hau90]. That proof shows that reachability sets are *almost semilinear*, a class of sets introduced in that paper that extends the class of Presburger sets. An application of that result was provided; we proved that a final configuration is not reachable from an initial one if and only if there exists a forward inductive invariant definable in the Presburger arithmetic that contains the initial configuration but not the final one. Since we can decide if a Presburger formula denotes a forward inductive invariant, we deduce that there exist checkable certificates of non-reachability in the Presburger arithmetic. In particular, there exists a simple algorithm for deciding the general VAS reachability problem based on two semi-algorithms. A first one that tries to prove the reachability by enumerating finite sequences of actions and a second one that tries to prove the non-reachability by enumerating Presburger formulas. The Presburger inductive invariants presented in that

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 327–341, 2011.

paper are obtained by over approximating production relations thanks to strongly connected subreachability graphs (called *witness graph* and recalled in Section 6). As a direct consequence, configurations in these graphs are reachable one from each other.

In this paper we consider the reversible reachability problem that consists to decide if two configurations are reachable one from each other. We prove that this problem is EXPSPACE-complete. This result extends known result for the subclasses of reversible and cyclic vector addition systems [BF97]. We also prove that the general *coverability problem* reduces to the reversible reachability problem (see Section 3). As an application of the introduced materials we characterize the reversibility domains of a vector addition system in the last Section 11.

2 Projected Vectors

In this paper, some components of vectors in $\mathbb{Z}^d$ are projected away. In order to avoid multiple dimensions, we introduce an additional element $\star \not\in \mathbb{Z}$, the set $\mathbb{Z}_\star = \mathbb{Z} \cup \{\star\}$, and the set $\mathbb{Z}_I^d$ of vectors $\boldsymbol{z} \in \mathbb{Z}_\star^d$ such that $I = \{i \mid \boldsymbol{z}(i) = \star\}$. Operations on $\mathbb{Z}$ are extended component-wise into operations on $\mathbb{Z}_I^d$ by interpreting $\star$ as a projected component. More formally we denote by $\boldsymbol{z}_1 + \boldsymbol{z}_2$ where $\boldsymbol{z}_1, \boldsymbol{z}_2 \in \mathbb{Z}_I^d$ the vector $\boldsymbol{z} \in \mathbb{Z}_I^d$ defined by $\boldsymbol{z}(i) = \boldsymbol{z}_1(i) + \boldsymbol{z}_2(i)$ for every $i \not\in I$. Symmetrically given $\boldsymbol{z} \in \mathbb{Z}_I^d$ and an integer $k \in \mathbb{Z}$, we denote by $k\boldsymbol{z}$ the vector in $\mathbb{Z}_I^d$ defined by $(k\boldsymbol{z})(i) = k(\boldsymbol{z}(i))$ for every $i \not\in I$. The relation $\leq$ is extended over $\mathbb{Z}_*^d$ component-wise by $\boldsymbol{z}_1 \leq \boldsymbol{z}_2$ if $\boldsymbol{z}_2(i) \neq \star$ then $\boldsymbol{z}_1(i) \neq \star$ and in this case $\boldsymbol{z}_1(i) \leq \boldsymbol{z}_2(i)$.

Example 2.1. We have $k(\star, 1) = (\star, k)$ even if $k = 0$. We also have $(\star, 5) - (\star, 2) = (\star, 3)$ and $(\star, 1) + (\star, 2) = (\star, 3)$. We have $\cdots \leq -1 \leq 0 \leq 1 \leq \cdots \leq \star$.

The *projection* of a vector $\boldsymbol{z} \in \mathbb{Z}_I^d$ over a set $L \subseteq \{1, \ldots, d\}$ of indexes is the vector in $\mathbb{Z}_{I \cup L}^d$ defined by $\pi_L(\boldsymbol{z})(i) = \boldsymbol{z}(i)$ for every $i \not\in L$. The projection of a set $\boldsymbol{Z} \subseteq \mathbb{Z}_I^d$ over L is defined as expected by $\pi_L(\boldsymbol{Z}) = \{\pi_L(\boldsymbol{z}) \mid \boldsymbol{z} \in \boldsymbol{Z}\}$.

Example 2.2. Let $L = \{1\}$. We have $\pi_L(1000, 1) = (\star, 1)$ and $\pi_L(4, \star) = (\star, \star)$. We also have $\pi_L(\{(2, 0), (1, 1), (2, 0)\}) = \{(\star, 0), (\star, 1), (\star, 2)\}$.

Let $\boldsymbol{z} \in \mathbb{Z}_I^d$. We denote by $||\boldsymbol{z}||_\infty$ the natural number equals to 0 if $I = \{1, \ldots, d\}$ and equals to $\max_{i \not\in I} |\boldsymbol{z}(i)|$ otherwise. Given a finite set $\boldsymbol{Z} \subseteq \mathbb{Z}_I^d$ we denote by $||\boldsymbol{Z}||_\infty$ the natural number $\max_{\boldsymbol{z} \in \boldsymbol{Z}} ||\boldsymbol{z}||_\infty$ if $\boldsymbol{Z}$ is non empty and 0 is $\boldsymbol{Z}$ is empty.

3 Vector Addition Systems

A *Vector Addition System* (*VAS*) is a finite set $\boldsymbol{A} \subseteq \mathbb{Z}^d$. Vectors $\boldsymbol{a} \in \boldsymbol{A}$ are called *actions* and vectors $\boldsymbol{c} \in \mathbb{N}_\star^d$ with $\mathbb{N}_\star = \mathbb{N} \cup \{\star\}$ are called *configurations*. A configuration in $\mathbb{N}^d$ is said to be *standard* and we denote by $\mathbb{N}_I^d$ the set of configurations $\boldsymbol{c} \in \mathbb{N}_\star^d$ such that $I = \{i \mid \boldsymbol{c}(i) = \star\}$. Given a word $\sigma = \boldsymbol{a}_1 \ldots \boldsymbol{a}_k$ of actions $\boldsymbol{a}_j \in \boldsymbol{A}$ we denote by $\Delta(\sigma)$ the vector in $\mathbb{Z}^d$ defined by $\Delta(\sigma) = \sum_{j=1}^{k} \boldsymbol{a}_j$. This vector is called the *displacement* of σ. We also introduce the vector $\Delta_I(\sigma) = \pi_I(\Delta(\sigma))$. A *run* ρ from a configuration $\boldsymbol{x} \in \mathbb{N}_I^d$ to a configuration $\boldsymbol{y} \in \mathbb{N}_I^d$ *labelled* by a word $\sigma = \boldsymbol{a}_1 \ldots \boldsymbol{a}_k$ of actions $\boldsymbol{a}_j \in \boldsymbol{A}$

is a non-empty word $\rho = \boldsymbol{c}_0 \dots \boldsymbol{c}_k$ of configurations $\boldsymbol{c}_j \in \mathbb{N}_I^d$ such that $\boldsymbol{c}_0 = \boldsymbol{x}, \boldsymbol{c}_k = \boldsymbol{y}$ and such that $\boldsymbol{c}_j = \boldsymbol{c}_{j-1} + \pi_I(\boldsymbol{a}_j)$ for every $j \in \{1, \dots, k\}$. Note that in this case ρ is unique and $\boldsymbol{y} - \boldsymbol{x} = \Delta_I(\sigma)$. This run is denoted by $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{y}$. The set I is called the set of *projected components* of ρ. The *projection* $\pi_L(\rho)$ of a run $\rho = \boldsymbol{c}_0 \dots \boldsymbol{c}_k$ over a set of indexes $L \subseteq \{1, \dots, d\}$ is defined as expected as the run $\pi_L(\rho) = \pi_L(\boldsymbol{c}_0) \dots \pi_L(\boldsymbol{c}_k)$. Observe that if ρ is the run $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{y}$ then $\pi_L(\rho)$ is the run $\pi_L(\boldsymbol{x}) \xrightarrow{\sigma} \pi_L(\boldsymbol{y})$. The following lemma provides a simple way to deduce a converse result.

Lemma 3.1. *Let L be a set of indexes and $\mathbf{c}$ be a configuration such that there exists a run from $\pi_L(\boldsymbol{c})$ labelled by a word σ. If $\boldsymbol{c}(i) \geq |\sigma| \, ||\boldsymbol{A}||_\infty$ for every $i \in L$ then there exists a run from $\mathbf{c}$ labelled by σ.*

Proof. Let $\boldsymbol{c} \in \mathbb{N}_I^d$ be a configuration such that there exists a path from $\pi_L(\boldsymbol{c})$ labelled by a word $\sigma = \boldsymbol{a}_1 \dots \boldsymbol{a}_k$ where $\boldsymbol{a}_j \in \boldsymbol{A}$. Let us introduce the vector $\boldsymbol{c}_j = \boldsymbol{c} + \pi_I(\boldsymbol{a}_1 + \dots + \boldsymbol{a}_j)$. Since there exists a run from $\pi_L(\boldsymbol{c})$ labelled by σ we deduce that $\pi_L(\boldsymbol{c}_j) \in \mathbb{N}_{I \cup L}^d$. Observe that for every $j \in \{0, \dots, k\}$ and for every $i \notin I$ we have $\boldsymbol{c}_j(i) \geq \boldsymbol{c}(i) - |\sigma| \, ||\boldsymbol{A}||_\infty$. In particular if $\boldsymbol{c}(i) \geq |\sigma| \, ||\boldsymbol{A}||_\infty$ for every $i \in L \backslash I$ we deduce that $\boldsymbol{c}_j \in \mathbb{N}_I^d$. Therefore $\rho = \boldsymbol{c}_0 \dots \boldsymbol{c}_k$ is a run from $\boldsymbol{c}$ labelled by σ. □

Example 3.2. $\rho = (2,0)(1,1)(0,2)$ is the run $(2,0) \xrightarrow{(-1,1)(-1,1)} (0,2)$. Let $L = \{1\}$ and observe that $\pi_L(\rho) = (\star,0)(\star,1)(\star,2)$ is the run $(\star,0) \xrightarrow{(-1,1)(-1,1)} (\star,2)$.

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two standard configurations. When there exists a run from $\boldsymbol{x}$ to $\boldsymbol{y}$ we say that $\boldsymbol{y}$ is *reachable* from $\boldsymbol{x}$ and if there also exists a run from $\boldsymbol{y}$ to $\boldsymbol{x}$ we say that $(\boldsymbol{x}, \boldsymbol{y})$ is in the *reversible reachability relation*. The problem of deciding this last property is called the *reversible reachability problem*. This problem is shown to be EXPSPACE-hard by introducing the *coverability problem*. Given two standard configurations $\boldsymbol{x}$ and $\boldsymbol{y}$ we say that $\boldsymbol{y}$ is *coverable* by $\boldsymbol{x}$ if there exists a standard configuration in $\boldsymbol{y} + \mathbb{N}^d$ reachable from $\boldsymbol{x}$. The coverability problem is known to be EXPSPACE-complete [CLM76, Rac78]. We reduce the coverability problem as follows. We first observe that we can add to a vector addition system $\boldsymbol{A}$ additional actions of the form $(0, \dots, 0, -1, 0, \dots, 0)$ without modifying the coverability problem. Thanks to this transformation a standard configuration $\boldsymbol{y}$ is coverable from a standard configuration $\boldsymbol{x}$ if and only if $\boldsymbol{y}$ is reachable from $\boldsymbol{x}$. We introduce the VAS $\boldsymbol{V}$ in dimension $d{+}2$ defined by $\boldsymbol{V} = ((0,0) \times \boldsymbol{A}) \cup \{(-1,1,-\boldsymbol{y}), (1,-1,\boldsymbol{x})\}$. Observe that $(1,0,\boldsymbol{x})$ and $(0,1,\boldsymbol{0})$ are in the reversible reachability relation of $\boldsymbol{V}$ if and only if $\boldsymbol{y}$ is coverable from $\boldsymbol{x}$ in $\boldsymbol{A}$. As a direct consequence, the reversible reachability problem is EXPSPACE-hard.

4 Subreachability Graphs

A *subreachability graph* is a graph $G = (\boldsymbol{Q}, T)$ where $\boldsymbol{Q} \subseteq \mathbb{N}_I^d$ is a non empty finite set of configurations called *states* and $T \subseteq \boldsymbol{Q} \times \boldsymbol{A} \times \boldsymbol{Q}$ is a finite set of triples $(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y}) \in \boldsymbol{Q} \times \boldsymbol{A} \times \boldsymbol{Q}$ satisfying $\boldsymbol{x} \xrightarrow{\boldsymbol{a}} \boldsymbol{y}$ called *transitions*. The set I is called the set of *projected components* of G and the subreachability graph is said to be *standard* if I is empty. A *witness graph* is a strongly connected subreachability graph (see Fig. 1 for examples).

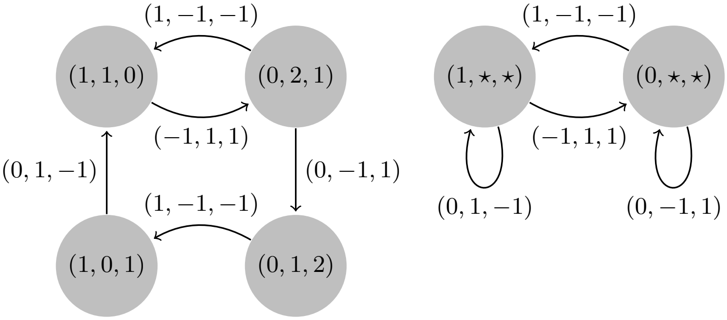


Fig. 1. A subreachability graph G and the subreachability graph $\pi_L(G)$ with $L = \{2, 3\}$

The *projection* $\pi_L(t)$ of a transition $t = (\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y})$ over a set of indexes $L \subseteq \{1, \dots, d\}$ is defined by $\pi_L(t) = (\pi_L(\boldsymbol{x}), \boldsymbol{a}, \pi_L(\boldsymbol{y}))$ and the projection of the set of transitions T is defined by $\pi_L(T) = \{\pi_L(t) \mid t \in T\}$. The projection $\pi_L(G)$ of a subreachability graph $G = (\boldsymbol{Q}, T)$ is the subreachability graph $\pi_L(G) = (\pi_L(\boldsymbol{Q}), \boldsymbol{A}, \pi_L(T))$.

Example 4.1. A standard subreachability graph $G = (\boldsymbol{Q}, T)$ and the subreachability graph $\pi_L(G)$ projected over the set $L = \{2, 3\}$ are depicted in Fig. 1.

A *path* in a subreachability graph G from a configuration $\boldsymbol{x} \in \boldsymbol{Q}$ to a configuration $\boldsymbol{y} \in \boldsymbol{Q}$ labelled by a word $\sigma = \boldsymbol{a}_1 \dots \boldsymbol{a}_k$ of actions $\boldsymbol{a}_j \in \boldsymbol{A}$ is a word $p = t_1 \dots t_k$ of transitions $t_j \in T$ of the form $t_j = (\boldsymbol{c}_{j-1}, \boldsymbol{a}_j, \boldsymbol{c}_j)$ with $\boldsymbol{c}_0 = \boldsymbol{x}$ and $\boldsymbol{c}_k = \boldsymbol{y}$. We observe that the word p is unique. This path is denoted by $\boldsymbol{x} \xrightarrow{\sigma}_G \boldsymbol{y}$. Let us observe that in this case $\rho = \boldsymbol{c}_0 \dots \boldsymbol{c}_k$ is the unique run $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{y}$. In particular if a path $\boldsymbol{x} \xrightarrow{\sigma}_G \boldsymbol{y}$ exists then the run $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{y}$ also exists. Note that conversely if there exists a run $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{y}$ then there exists a subreachability G such that $\boldsymbol{x} \xrightarrow{\sigma}_G \boldsymbol{y}$. Such a G is obtained by introducing the set of states $\boldsymbol{Q} = \{\boldsymbol{c}_0, \dots, \boldsymbol{c}_k\}$ and the set of transitions $T = \{t_1, \dots, t_k\}$ where $t_j = (\boldsymbol{c}_{j-1}, \boldsymbol{a}_j, \boldsymbol{c}_j)$. A path $\boldsymbol{x} \xrightarrow{\sigma}_G \boldsymbol{y}$ is called a *cycle* if $\boldsymbol{x} = \boldsymbol{y}$. The cycle is said to be *simple* if $\boldsymbol{c}_{j_1} = \boldsymbol{c}_{j_2}$ with $j_1 < j_2$ implies $j_1 = 0$ and $j_2 = k$. The projection $\pi_L(p)$ of a path $p = t_1 \dots t_k$ in G over a set of indexes $L \subseteq \{1, \dots, d\}$ is the path $\pi_L(p) = \pi_L(t_1) \dots \pi_L(t_k)$ in $\pi_L(G)$. Observe that the projection of a path $\boldsymbol{x} \xrightarrow{\sigma}_G \boldsymbol{y}$ over L is the path $\pi_L(\boldsymbol{x}) \xrightarrow{\sigma}_{\pi_L(G)} \pi_L(\boldsymbol{y})$. The *Parikh image* of a path is the function $\mu : T \to \mathbb{N}$ defined by $\mu(t)$ is the number of occurrences of t in this path. A cycle is said to be *total* if its Parikh image μ satisfies $\mu(t) \geq 1$ for every $t \in T$.

Example 4.2. Let us come back to the standard witness graph G depicted in Fig. 1. Let us consider the cycle $(1, 1, 0) \xrightarrow{(-1,1,1)(1,-1,-1)}_G (1, 1, 0)$ in G. Its projection over $L = \{2, 3\}$ is the cycle $(1, \star, \star) \xrightarrow{(-1,1,1)(1,-1,-1)}_{\pi_L(G)} (1, \star, \star)$ in the witness graph $\pi_L(G)$ also depicted in Fig. 1.

A word $\sigma \in \boldsymbol{A}^*$ is said to be *forward iterable* from a configuration $\boldsymbol{c}$ if there exists a run $\boldsymbol{c} \xrightarrow{\sigma} \boldsymbol{y}$ such that $\boldsymbol{c} \leq \boldsymbol{y}$. In this case the configuration $\boldsymbol{c}_\star = \pi_L(\boldsymbol{c})$ where

$L = \{i \mid \boldsymbol{c}(i) \neq \boldsymbol{y}(i)\}$ is called the *forward limit* of σ from $\boldsymbol{c}$. We observe that σ is forward iterable from $\boldsymbol{c}$ if and only if for every $n \in \mathbb{N}$ there exists a run $\boldsymbol{c} \xrightarrow{\sigma^n} \boldsymbol{y}_n$. In that case L is the minimal set of indexes such that $\pi_L(\boldsymbol{y}_n)$ does not depend on n. Symmetrically σ is said to be *backward iterable* from a configuration $\boldsymbol{c}$ if there exists a run $\boldsymbol{x} \xrightarrow{\sigma} \boldsymbol{c}$ such that $\boldsymbol{c} \leq \boldsymbol{x}$. In this case the configuration $\boldsymbol{c}_\star = \pi_L(\boldsymbol{c})$ where $L = \{i \mid \boldsymbol{c}(i) \neq \boldsymbol{x}(i)\}$ is called the *backward limit* of σ from $\boldsymbol{c}$.

Example 4.3. The action $\boldsymbol{a} = (0, -1, 1)$ is forward iterable from $\boldsymbol{x} = (0, \star, 0)$ since $(0, \star, 0) \xrightarrow{\boldsymbol{a}} (0, \star, 1)$. Observe that in this case $(0, \star, 0) \xrightarrow{\boldsymbol{a}^n} (0, \star, n)$ for every $n \in \mathbb{N}$. The forward limit of $\boldsymbol{a}$ from $(0, \star, 0)$ is $(0, \star, \star)$.

A configurations $\boldsymbol{c}$ is said to be *forward pumpable* by a cycle $\boldsymbol{q} \xrightarrow{\sigma}_G \boldsymbol{q}$ if σ is forward iterable from $\boldsymbol{c}$ with a forward limit equals to $\boldsymbol{q}$. Note that in this case $\boldsymbol{q}$ is unique since it satisfies $\boldsymbol{q} = \pi_I(\boldsymbol{c})$ where I is the set of projected components of G. Symmetrically a configuration $\boldsymbol{c}$ is said to be *backward pumpable* by a cycle $\boldsymbol{q} \xrightarrow{\sigma}_G \boldsymbol{q}$ if σ is backward iterable from $\boldsymbol{c}$ with a backward limit equals to $\boldsymbol{q}$.

Example 4.4. Let us come back to the witness graph $\pi_L(G)$ depicted in Fig. 1. Observe that $(0, \star, 0)$ is forward pumpable by $(0, \star, \star) \xrightarrow{(0,-1,1)}_{\pi_L(G)} (0, \star, \star)$.

5 Outline

The reminder of this paper is a proof that the reversible reachability problem is in EXPSPACE. We prove that if a pair $(\boldsymbol{x}, \boldsymbol{y})$ of standard configurations are in the reversible reachability relation then there exist runs from $\boldsymbol{x}$ to $\boldsymbol{y}$ and from $\boldsymbol{y}$ to $\boldsymbol{x}$ with lengths bounded by a number double exponential in the size of $(\boldsymbol{x}, \boldsymbol{A}, \boldsymbol{y})$. Using the fact that NEXPSPACE=EXPSPACE, and that double exponential numbers can be stored in exponential space, one obtain the EXPSPACE upper bound. These "short" runs are obtained as follows.

Theorem 6.3 gives a bound on the size of the Parikh image of a cycle in a witness graph to achieve a particular displacement vector, using a result of Pottier [Pot91]. This result is used in Section 7, which considers the special case of reversible witness graphs in which each path can be followed by another path such that the total displacement is zero. In Theorem 7.3 it is shown that a reversible witness graph possesses a "short" total cycle that has a zero displacement.

Section 9 takes an arbitrary witness graph G and asserts the existence of a set of indexes J such that the witness graph $\pi_J(G)$ has a "small" number of states and such that states of G that are not "too" large are pumpable by "short" cycles in $\pi_J(G)$.

The development culminates with the main result in Section 10. There, it is shown that $\boldsymbol{x}$ and $\boldsymbol{y}$ are two states of a reversible witness graph G. One then uses the result from Section 9 to generate a reversible witness graph $\pi_J(G)$. Most of the work involves showing how to replace arbitrary path between $\boldsymbol{x}$ and $\boldsymbol{y}$ by "short" paths by exploiting the fact that $\boldsymbol{x}$ and $\boldsymbol{y}$ are pumpable to move from $\pi_J(G)$ back to G.

6 Displacement Vectors

A *displacement vector* of a witness graph G is a finite sum of vectors of the form $\Delta(\sigma) = \sum_{j=1}^{k} \boldsymbol{a}_j$ where $\sigma = \boldsymbol{a}_1 \dots \boldsymbol{a}_k$ is a word labelling a cycle in G. We denote by $\boldsymbol{Z}_G$ the set of *displacement vectors*. Observe that $\boldsymbol{Z}_G$ is a submonoid of $(\mathbb{Z}^d, +)$. Displacement vectors are related to *Kirchhoff functions* as follows. A *Kirchhoff function* for a witness graph $G = (\boldsymbol{Q}, T)$ is a function $\mu : T \to \mathbb{N}$ such that the functions $\mathrm{in}(\mu), \mathrm{out}(\mu) : \boldsymbol{Q} \to \mathbb{N}$ defined bellow are equal.

$$\mathrm{in}(\mu)(\boldsymbol{x}) = \sum_{t \in T \cap (\boldsymbol{Q} \times \boldsymbol{A} \times \{\boldsymbol{x}\})} \mu(t) \qquad \mathrm{out}(\mu)(\boldsymbol{x}) = \sum_{t \in T \cap (\{\boldsymbol{x}\} \times \boldsymbol{A} \times \boldsymbol{Q})} \mu(t)$$

A Kirchhoff function $\mu : T \to \mathbb{N}$ is said to be *total* if $\mu(t) \geq 1$ for every $t \in T$.

Lemma 6.1 (Euler's Lemma). *A function μ is a Kirchhoff function for a witness graph G if and only if μ is a finite sum of Parikh images of cycles in G. In particular a function μ is a total Kirchhoff function if and only if μ is the Parikh image of a total cycle.*

As a direct consequence of the Euler's Lemma, we deduce that a vector $\boldsymbol{z} \in \mathbb{Z}^d$ is a displacement vector of G if and only if there exists a Kirchhoff function μ for G satisfying the following equality:

$$\boldsymbol{z} = \sum_{t = (\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y}) \in T} \mu(t) \boldsymbol{a}$$

In this case $\boldsymbol{z}$ is called the *displacement* of μ.

Example 6.2. Let us come back to the witness graph $\pi_L(G)$ depicted in Fig. 1. A function $\mu : \pi_L(T) \to \mathbb{N}$ is a Kirchhoff function for $\pi_L(G)$ if and only if $\mu(t_1) = \mu(t_2)$ where $t_1 = ((1, \star, \star), (-1, 1, 1), (0, \star, \star))$ and $t_2 = ((0, \star, \star), (1, -1, -1), (1, \star, \star))$. In particular the set of displacement vectors of $\pi_L(G)$ satisfies $\boldsymbol{Z}_{\pi_L(G)} = \{\boldsymbol{z} \in \mathbb{Z}^3 \mid \boldsymbol{z}(1) = 0 \wedge \boldsymbol{z}(2) + \boldsymbol{z}(3) = 0\}$.

The following theorem shows that the displacement vectors $\boldsymbol{z} \in \boldsymbol{Z}_G$ are displacement of Kirchhoff functions μ for G such that $||\mu||_\infty = \max_{t \in T} \mu(t)$ is bounded by a polynomial in $|\boldsymbol{Q}|$, $||\boldsymbol{A}||_\infty$, and $||\boldsymbol{z}||_\infty$.

Theorem 6.3. *Vectors $\boldsymbol{z} \in \boldsymbol{Z}_G$ are displacement of Kirchhoff functions μ such that the following inequality holds where $q = |\boldsymbol{Q}|$, $a = ||\boldsymbol{A}||_\infty$, and $m = ||\boldsymbol{z}||_\infty$:*

$$||\mu||_\infty \leq (q^{d+1} a (1 + 2a)^d + m)^d$$

Proof. We first recall a "Frobenius theorem" proved in [Pot91]. Let $H \in \mathbb{Z}^{d \times n}$ be a matrix and let us denote by $h_{i,j}$ for each $i \in \{1, \dots, d\}$ and $j \in \{1, \dots, n\}$ the element of H at position (i, j). We denote by $||H||_{1,\infty}$ the natural number $\max_{1 \leq i \leq d} \sum_{j=1}^{n} |h_{i,j}|$. Given a vector $\boldsymbol{v} \in \mathbb{N}^n$, we introduce the natural number $||\boldsymbol{v}||_1 = \sum_{j=1}^{n} \boldsymbol{v}(j)$. Let $\boldsymbol{V}$ be the set of vectors $\boldsymbol{v} \in \mathbb{N}^n$ such that $H\boldsymbol{v} = \boldsymbol{0}$. Recall that $\boldsymbol{V}$ is a submonoid of $(\mathbb{N}^n, +)$ generated by the finite set $\min(\boldsymbol{V} \backslash \{\boldsymbol{0}\})$ of minimal elements for $\leq$. From [Pot91] we deduce that vectors $\boldsymbol{v} \in \min(\boldsymbol{V}\backslash\{\boldsymbol{0}\})$ satisfy the following inequality where r is the rank of H:

$$||\boldsymbol{v}||_1 \leq (1 + ||H||_{1,\infty})^r$$

Observe that if $a = 0$ then $\boldsymbol{z} = \boldsymbol{0}$ and the theorem is proved with the Kirchhoff function μ defined by $\mu(t) = 0$ for every $t \in T$. So we can assume that $a \geq 1$. Since every cycle labelled by a word σ can be decomposed into a finite sequence of simple cycles labelled by words $\sigma_1, \dots, \sigma_k$ such that $\Delta(\sigma) = \sum_{j=1}^{k} \Delta(\sigma_j)$ we deduce that the set of displacement vectors $\boldsymbol{Z}_G$ is the submonoid of $(\mathbb{Z}^d, +)$ generated by the set $\boldsymbol{Z}$ of non-zero vectors $\boldsymbol{z} = \Delta(\sigma)$ where σ is the label of a simple cycle. Since the length of a simple cycle is bounded by the cardinal q of $\boldsymbol{Q}$, we get $||\boldsymbol{Z}||_\infty \leq qa$. As a corollary we deduce that the cardinal k of $\boldsymbol{Z}$ is bounded by $k \leq (1 + 2qa)^d - 1$ (the -1 comes from the fact that vectors in $\boldsymbol{Z}$ are non-zero).

Let us consider a vector $\boldsymbol{z} \in \boldsymbol{Z}_G$ and let us introduce a whole enumeration $\boldsymbol{z}_1, \dots, \boldsymbol{z}_k$ of the vectors in $\boldsymbol{Z}$ and the following set $\boldsymbol{V}$ where $n = k + 1$:

$$\boldsymbol{V} = \{\boldsymbol{v} \in \mathbb{N}^n \mid \bigwedge_{i=1}^{d} \sum_{j=1}^{k} \boldsymbol{v}(j)\boldsymbol{z}_j(i) - \boldsymbol{v}(n)\boldsymbol{z}(i) = 0\}$$

We observe that $\boldsymbol{V}$ is associated to a matrix $H \in \mathbb{Z}^{d \times n}$. The rank of H is bounded by d and $||H||_{1,\infty} \leq kqa + m$. We deduce from the Frobenius theorem that vectors $\boldsymbol{v} \in \min(\boldsymbol{V}\backslash\{\boldsymbol{0}\})$ satisfy the following inequality:

$$||\boldsymbol{v}||_1 \leq (1 + kqa + m)^d \leq (q^{d+1}a(1 + 2a)^d + m)^d$$

Since $\boldsymbol{z} \in \boldsymbol{Z}_G$, there exists a vector $\boldsymbol{v} \in \boldsymbol{V}$ such that $\boldsymbol{v}(n) = 1$. In particular there exists another vector $\boldsymbol{v} \in \min(\boldsymbol{V}\backslash\{\boldsymbol{0}\})$ such that $\boldsymbol{v}(n) = 1$. Observe that for every $j \in \{1, \dots, k\}$ there exists a function λ_j that is the Parikh image of a simple cycle such that $\boldsymbol{z}_j$ is the displacement of λ_j. We introduce the Kirchhoff function $\mu = \sum_{j=1}^{k} \boldsymbol{v}(j)\lambda_j$. Since $\boldsymbol{v} \in \boldsymbol{V}$ and $\boldsymbol{v}(n) = 1$ we deduce that the displacement of μ is $\boldsymbol{z}$. The theorem is proved by observing that $\mu(t) = \sum_{j=1}^{k} \boldsymbol{v}(j)\lambda_j(t) \leq ||\boldsymbol{v}||_1$ since $\lambda_j(t) \in \{0, 1\}$. □

7 Reversible Witness Graphs

A witness graph G is said to be *reversible* if for every path $\boldsymbol{x} \xrightarrow{u}_G \boldsymbol{y}$ there exists a path $\boldsymbol{y} \xrightarrow{v}_G \boldsymbol{x}$ such that $\Delta(u) + \Delta(v) = \boldsymbol{0}$. Observe that standard witness graphs are reversible since the condition $\Delta(u) + \Delta(v) = \boldsymbol{0}$ is implied by the two paths.

Example 7.1. Witness graphs depicted in Fig. 1 are reversible. The witness graph $G = (\{\star\}, \{(\star, 1, \star)\})$ is not reversible.

Let us recall that a submonoid $\boldsymbol{Z}$ of $(\mathbb{Z}^d, +)$ is said to be a *subgroup* if $-\boldsymbol{z} \in \boldsymbol{Z}$ for every $\boldsymbol{z} \in \boldsymbol{Z}$. The following lemma provides two characterizations of the reversible witness graphs.

Lemma 7.2. *A witness graph G is reversible if and only if $\boldsymbol{Z}_G$ is a subgroup of $(\mathbb{Z}^d, +)$ if and only if the zero vector is the displacement of a total Kirchhoff function.*

Proof. Assume first that G is reversible and let us prove that $\boldsymbol{Z}_G$ is a subgroup of $(\mathbb{Z}^d, +)$. Let us consider a cycle $\boldsymbol{x} \xrightarrow{u}_G \boldsymbol{x}$. Since G is reversible, there exists a cycle $\boldsymbol{x} \xrightarrow{v}_G \boldsymbol{x}$ such that $\Delta(u) + \Delta(v) = \boldsymbol{0}$. We deduce that $-\boldsymbol{Z}_G = \boldsymbol{Z}_G$ since vectors in $\boldsymbol{Z}_G$ are finite sums of vectors $\Delta(u)$ where u is the label of a cycle in G. Therefore $\boldsymbol{Z}_G$ is a subgroup of $\mathbb{Z}^d$.

Now let us assume that $\boldsymbol{Z}_G$ is a subgroup of $(\mathbb{Z}^d, +)$ and let us prove that the zero vector is the displacement of a total Kirchhoff function. Since G is strongly connected, there exists a total cycle $\boldsymbol{x} \xrightarrow{u}_G \boldsymbol{x}$. Observe that $\boldsymbol{z} = \Delta(u)$ is in $\boldsymbol{Z}_G$. Since $\boldsymbol{Z}_G$ is a subgroup we deduce that $-\boldsymbol{z} \in \boldsymbol{Z}_G$. Hence $-\boldsymbol{z}$ is the displacement of a Kirchhoff function λ. Let λ' be Parikh image of $\boldsymbol{x} \xrightarrow{u}_G \boldsymbol{x}$ and observe that $\mu = \lambda + \lambda'$ is a total Kirchhoff function. Moreover the displacement of μ is $-\boldsymbol{z} + \boldsymbol{z} = \boldsymbol{0}$.

Finally, let us assume that the zero vector is the displacement of a total Kirchhoff function μ and let us prove that G is reversible. Let us consider a path $\boldsymbol{x} \xrightarrow{u}_G \boldsymbol{y}$. Since G is strongly connected, there exists a path $\boldsymbol{y} \xrightarrow{\alpha}_G \boldsymbol{x}$. Let us consider the Parikh image λ of the cycle $\boldsymbol{x} \xrightarrow{u\alpha}_G \boldsymbol{x}$ and let $m = 1 + ||\lambda||_\infty$. We observe that $\mu' = m\mu - \lambda$ is a total Kirchhoff function and the Euler's Lemma shows that μ' is the Parikh image of a cycle $\boldsymbol{x} \xrightarrow{\beta}_G \boldsymbol{x}$. From $\mu' = m\mu - \lambda$ we deduce that $\Delta(\beta) = m\boldsymbol{0} - \Delta(u\alpha)$. Let us consider $v = \alpha\beta$ and observe that $\boldsymbol{y} \xrightarrow{v}_G \boldsymbol{x}$ and $\Delta(u) + \Delta(v) = \boldsymbol{0}$. Thus G is reversible. □

The following theorem shows that if G is a reversible witness graph then the zero vector is the displacement of a total Kirchhoff function μ such $||\mu||_\infty$ can be bounded by a polynomial in $|\boldsymbol{Q}|$ and $||\boldsymbol{A}||_\infty$.

Theorem 7.3. *Let G be a reversible witness graph. The zero vector is the displacement of a total Kirchhoff function μ such that the following inequality holds where $q = |\boldsymbol{Q}|$ and $a = ||\boldsymbol{A}||_\infty$:*

$$||\mu||_\infty \leq (q(1+2a))^{d(d+1)}$$

Proof. Since G is strongly connected, every transition $t \in T$ occurs in at least one simple cycle. We denote by λ_t the Parikh image of such a simple cycle and we introduce the Kirchhoff function $\lambda = \sum_{t \in T} \lambda_t$. We have $\lambda(t) \in \{1, \ldots, |T|\}$ for every $t \in T$. We introduce the displacement $\boldsymbol{z}$ of λ. Since G is reversible, we deduce that $-\boldsymbol{z}$ is the displacement vector of a Kirchhoff function for G. As $||\boldsymbol{z}||_\infty \leq |T|qa$, $|T| \leq q|\boldsymbol{A}|$, and $|\boldsymbol{A}| \leq (1+2a)^d$ we deduce that $||\boldsymbol{z}||_\infty \leq q^2 a(1+2a)^d$. Theorem 6.3 shows that $-\boldsymbol{z}$ is the displacement of a Kirchhoff function λ' satisfying the following inequalities:

$$||\lambda'||_\infty \leq (q^d a(1+2a)^d + q^2 a(1+2a)^d)^d \leq (q^{d+1} 2a(1+2a)^d)^d$$

Let us consider the total Kirchhoff function $\mu = \lambda + \lambda'$. Observe that the displacement of μ is the zero vector and since $||\lambda||_\infty \leq |T| \leq q(1+2a)^d \leq (q^{d+1}(1+2a)^d)^d$ we get the theorem with:

$$||\mu||_\infty \leq (q^{d+1} 2a(1+2a)^d)^d + (q^{d+1}(1+2a)^d)^d \leq (q(1+2a))^{d(d+1)}$$

□

8 Extractors

In this section we introduce a way for extracting "large" components of configurations. An *extractor* is a non increasing sequence $\lambda = (\lambda_n)_{1 \leq n \leq d}$ of natural numbers $\lambda_n \in \mathbb{N}$. Let $\boldsymbol{X} \subseteq \mathbb{N}_I^d$. An *excluding* set for $(\lambda, \boldsymbol{X})$ is a set of indexes J such that $\boldsymbol{x}(i) < \lambda_{|J|+1}$ for every $i \notin J$ and for every $\boldsymbol{x} \in \boldsymbol{X}$. Since λ is non increasing we deduce that the class of excluding sets for a couple $(\lambda, \boldsymbol{X})$ is stable by intersection. As this class contains $\{1, \dots, d\}$ we deduce that there exists a *unique minimal excluding set* J for $(\lambda, \boldsymbol{X})$. By minimality of this set we deduce that for every $i \in J$ there exists $\boldsymbol{x} \in \boldsymbol{X}$ such that $\boldsymbol{x}(i) \geq \lambda_{|J|}$. We denote $\lambda(\boldsymbol{X})$ the set $\pi_J(\boldsymbol{X})$ where J is the minimal excluding set for $(\lambda, \boldsymbol{X})$. A set $\boldsymbol{X} \subseteq \mathbb{N}_I^d$ is said to be *normalized* for λ if $\lambda(\boldsymbol{X}) = \boldsymbol{X}$. As a direct consequence of the following lemma we deduce that $\lambda(\boldsymbol{X})$ is normalized for λ for every set $\boldsymbol{X} \subseteq \mathbb{N}_I^d$.

Lemma 8.1. *Let $\boldsymbol{X} \subseteq \mathbb{N}_I^d$ and let L be a set of indexes included in the minimal excluding set of $(\lambda, \boldsymbol{X})$. Then $\lambda(\boldsymbol{X}) = \lambda(\pi_L(\boldsymbol{X}))$.*

Proof. Note that if $\boldsymbol{X}$ is empty the result is immediate so we can assume that $\boldsymbol{X}$ is non empty. Let J be the minimal excluding set of $(\lambda, \boldsymbol{X})$ and observe that J is an excluding set for $\boldsymbol{X}' = \pi_L(\boldsymbol{X})$. In particular the minimal excluding set J' for $\boldsymbol{X}'$ satisfies $J' \subseteq J$. Since J' is an excluding set of $(\lambda, \boldsymbol{X}')$ we deduce that $\boldsymbol{x}'(i) < \lambda_{|J'|+1}$ for every $i \notin J'$. Hence $\pi_L(\boldsymbol{x})(i) < \lambda_{|J'|+1}$ for every $\boldsymbol{x} \in \boldsymbol{X}$. As $\boldsymbol{x} \leq \pi_L(\boldsymbol{x})$ we deduce that J' is an excluding set of $(\lambda, \boldsymbol{X})$. By minimality of J we get the other inclusion $J \subseteq J'$. Thus $J = J'$ and we have proved that $\lambda(\boldsymbol{X}) = \lambda(\pi_L(\boldsymbol{X}))$. □

Example 8.2. Let $\lambda = (5, 3, 2)$ be an extractor. We have $\lambda(\{(1, 8, 1)\}) = \{(1, \star, 1)\}$, $\lambda(\{(1, 8, 1), (3, 1, 1)\}) = \{(\star, \star, 1)\}$.

9 Pumpable Configurations

In this section we show that for arbitrary witness graph G, there exists a set J of indexes such that the number of states of $\pi_J(G)$ is "small" and such that states with "small" size of G are pumpable by "short" cycles of $\pi_J(G)$. The proof of this result is inspired by the Rackoff ideas [Rac78]. All other results or definitions introduced in this section are not used in the sequel.

Theorem 9.1. *Let G be a witness graph and let $s \in \mathbb{N}_{>0}$ be a positive integer. We introduce the positive integer $x = (1 + ||\boldsymbol{A}||_\infty)s$. There exists a set of indexes J such that the number of states of $\pi_J(G)$ is bounded by x^{d^d} and such that every state $\boldsymbol{q} \in \boldsymbol{Q}$ such that $||\boldsymbol{q}||_\infty < s$ is forward and backward pumpable by cycles of $\pi_J(G)$ with lengths bounded by dx^{d^d}.*

Such a set J is obtained by introducing the class of *adapted extractors*. An extractor λ is said to be *adapted* if the following inequality holds for every $n \in \{2, \dots, d\}$:

$$\lambda_{n-1} \geq \lambda_n^{d-n+1} ||\boldsymbol{A}||_\infty + \lambda_n$$

Lemma 9.2. *Let λ be an adapted extractor, G be a witness graph with a set of states $\boldsymbol{Q} \subseteq \mathbb{N}_I^d$, and let J be the minimal excluding set for $(\lambda, \boldsymbol{Q})$. For every state $\boldsymbol{q} \in \boldsymbol{Q}$ there exists a run $\boldsymbol{q} \xrightarrow{u} \boldsymbol{y}$ such that $\pi_J(\boldsymbol{q}) \xrightarrow{u}_{\pi_J(G)} \pi_J(\boldsymbol{y})$ and such that the bounds $|u| \leq \sum_{|I|<n\leq|J|} \lambda_n^{d+1-n}$, and $\boldsymbol{y}(j) \geq \lambda_{|J|}$ for every $j \in J$ hold.*

Proof. Since $\boldsymbol{Q} \subseteq \mathbb{N}_I^d$ we deduce that $I \subseteq J$. We introduce a parameter $k \in \mathbb{N}$ and we prove the lemma by induction over k under the constraint $|J| - |I| \leq k$. Observe that if $k = 0$ then $I = J$ and the property is proved with $u = \epsilon$ and $\boldsymbol{y} = \boldsymbol{q}$. Assume the property proved for a natural number $k \in \mathbb{N}$ and let us consider a witness graph G with a set of projected components I such that $|J| - |I| \leq k + 1$ where J is the minimal excluding set for $(\lambda, \boldsymbol{Q})$. We consider a state $\boldsymbol{q} \in \boldsymbol{Q}$.

We say that a state $\boldsymbol{p} \in \boldsymbol{Q}$ is normalized if $\{\boldsymbol{p}\}$ is normalized for λ, i.e $\lambda(\{\boldsymbol{p}\}) = \{\boldsymbol{p}\}$ or equivalently $\boldsymbol{p}(i) < \lambda_{|I|+1}$ for every $i \notin I$. Observe that if every state $\boldsymbol{p} \in \boldsymbol{Q}$ is normalized then $\lambda(\boldsymbol{Q}) = \boldsymbol{Q}$ and in particular $J = I$ and the property is proved. So we can assume that there exists a state in $\boldsymbol{p} \in \boldsymbol{Q}$ that is not normalized. Since G is strongly connected, there exists a path $\boldsymbol{q} \xrightarrow{\sigma}_G \boldsymbol{p}$ with a minimal length such that $\boldsymbol{p}$ is not normalized. Let us observe that the number of states in $\boldsymbol{Q}$ that are normalized is bounded by $\lambda_{|I|+1}^{d-|I|}$. By minimality of the length of σ we deduce that $|\sigma| \leq \lambda_{|I|+1}^{d-|I|}$.

We introduce the minimal excluding set K for $(\lambda, \{\boldsymbol{p}\})$. Observe that I is strictly included in K since $\boldsymbol{p}$ is not normalized. Moreover K is included in J since J is an excluding set for $(\lambda, \{\boldsymbol{p}\})$. Lemma 8.1 shows that J is the minimal excluding set of $(\lambda, \pi_K(\boldsymbol{Q}))$. Observe that $|J| - |K| < |J| - |I| \leq k + 1$. By applying the induction on the witness graph $\pi_K(G)$ and the state $\pi_K(\boldsymbol{p})$, we deduce that there exists a run $\pi_K(\boldsymbol{p}) \xrightarrow{u} \boldsymbol{y}$ such that $\pi_J(\boldsymbol{p}) \xrightarrow{u}_{\pi_J(G)} \pi_J(\boldsymbol{y})$ with $|u| \leq \sum_{|K|<n\leq|J|} \lambda_n^{d+1-n}$ and such that $\boldsymbol{y}(j) \geq \lambda_{|J|}$ for every $j \in J$. We introduce the word $v = \sigma u$. Since λ is an adapted extractor we deduce that $\lambda_{|K|} \geq ||\boldsymbol{A}||_\infty \sum_{|K|<n\leq|J|} \lambda_n^{d+1-n}$. From $\boldsymbol{p}(k) \geq \lambda_{|K|}$ for every $k \in K$ we deduce that $\boldsymbol{p}(k) \geq ||\boldsymbol{A}||_\infty |u|$. Since there exists a run from $\pi_K(\boldsymbol{p})$ labelled by u, Lemma 3.1 shows that there exists a run $\boldsymbol{p} \xrightarrow{u} \boldsymbol{z}$. For every $k \in K$ we have $\boldsymbol{z}(k) \geq \lambda_{|J|}$ if $\boldsymbol{p}(k) = \star$ and $\boldsymbol{z}(k) \geq \boldsymbol{p}(k) - ||\boldsymbol{A}||_\infty |u| \geq \lambda_{|J|}$ otherwise since λ is an adapted extractor. As $\boldsymbol{p} \xrightarrow{u} \boldsymbol{z}$ we deduce that $\pi_K(\boldsymbol{p}) \xrightarrow{u} \pi_K(\boldsymbol{z})$. In particular $\pi_K(\boldsymbol{z}) = \boldsymbol{y}$. Let $j \in J \backslash K$. From the previous equality we get $\boldsymbol{z}(j) = \boldsymbol{y}(j)$. Moreover since $\boldsymbol{y}(j) \geq \lambda_{|J|}$ we get $\boldsymbol{z}(j) \geq \lambda_{|J|}$. We have proved that $\boldsymbol{z}(j) \geq \lambda_{|J|}$ for every $j \in J$. Hence the induction is proved. □

Now let us prove Theorem 9.1. We consider a witness graph G with a set of states $\boldsymbol{Q} \subseteq \mathbb{N}_I^d$. We also consider a positive integer $s \in \mathbb{N}_{>0}$ and we introduce the positive integers $a = ||\boldsymbol{A}||_\infty$ and $x = (1+a)s$. Let λ be the adapted extractor defined by $\lambda_d = s$ and the following induction for every $n \in \{2, \ldots, d\}$:

$$\lambda_{n-1} = \lambda_n^d (1 + ||\boldsymbol{A}||_\infty)$$

An immediate induction provides $\lambda_n^{d+1-n} \leq x^{d^d}$ for every $n \in \{1, \ldots, d\}$. We introduce the minimal excluding set J for $(\lambda, \boldsymbol{Q})$. Observe that the number of states in $\pi_J(\boldsymbol{Q})$ is bounded by $\lambda_{|J|+1}^{d-|J|}$. Hence $|\pi_J(\boldsymbol{Q})| \leq x^{d^d}$. Let us consider $\boldsymbol{q} \in \boldsymbol{Q}$ such that $||\boldsymbol{q}||_\infty < s$. Lemma 9.2 shows that there exists a run $\boldsymbol{q} \xrightarrow{\sigma} \boldsymbol{x}$ with $\boldsymbol{x}(j) \geq \lambda_{|J|}$ for every $j \in J$ such that $\pi_J(\boldsymbol{q}) \xrightarrow{\sigma}_{\pi_J(G)} \pi_J(\boldsymbol{x})$ and such that:

$$|\sigma| \leq \sum_{n=1}^{|J|} \lambda_n^{d+1-n}$$

Since $\pi_J(G)$ is strongly connected there exists a path $\pi_J(\boldsymbol{x}) \xrightarrow{u}_{\pi_J(G)} \pi_J(\boldsymbol{q})$. We can assume that the length of u is minimal. In particular $u = \epsilon$ if $J = \{1, \ldots, d\}$ and $|u| \leq \lambda_{|J|+1}^{d-|J|}$ otherwise. In both case $|\sigma u| \leq dx^{d^d}$. Since λ is an adapted extractor we deduce that $\boldsymbol{x}(j) \geq |u| \, ||\boldsymbol{A}||_\infty$ for every $j \in J$ and by applying Lemma 3.1 we deduce that there exists a run $\boldsymbol{x} \xrightarrow{u} \boldsymbol{y}$. Since $\pi_J(\boldsymbol{x}) \xrightarrow{u} \pi_J(\boldsymbol{q})$ we deduce that $\boldsymbol{y}(j) = \boldsymbol{q}(j)$ for every $j \notin J$. Moreover if $j \in J \backslash I$ since $\boldsymbol{y}(j) \geq s$ and $s > ||\boldsymbol{q}||_\infty$ we get $\boldsymbol{y}(j) > \boldsymbol{q}(j)$. We deduce that $\boldsymbol{q} \leq \boldsymbol{y}$ and $J \backslash I = \{i \mid \boldsymbol{q}(i) \neq \boldsymbol{y}(i)\}$. Therefore $\boldsymbol{q}$ is forward pumpable by the cycle $\pi_J(\boldsymbol{q}) \xrightarrow{\sigma u}_{\pi_J(G)} \pi_J(\boldsymbol{q})$.

Symmetrically we prove the backward case. We have proved Theorem 9.1.

10 Deciding The Reversibility Problem

In this section, the reversible reachability problem is proved to be EXPSPACE-complete. The proof is inspired by the Kosaraju ideas [Kos82]. A word $\alpha \in \boldsymbol{A}^*$ is said to be *reversible* on a configuration $\boldsymbol{c}$ if there exists a word $\beta \in \boldsymbol{A}^*$ such that $\boldsymbol{c} \xrightarrow{\alpha\beta} \boldsymbol{c}$ and $\Delta(\alpha) + \Delta(\beta) = \boldsymbol{0}$. Note that if $\boldsymbol{c}$ is a standard configuration the last condition is implied by the first one.

Theorem 10.1. *Let $\alpha \in \boldsymbol{A}^*$ be a reversible word on a configuration **c**. There exists another word $\alpha' \in \boldsymbol{A}^*$ reversible on **c** such that $\Delta(\alpha) = \Delta(\alpha')$ and such that:*

$$|\alpha'| \leq 17d^2 x^{15d^{d+2}}$$

where $x = (1 + 2||\boldsymbol{A}||_\infty)\,(1 + ||\boldsymbol{p}||_\infty + ||\Delta(\alpha)||_\infty)$.

Let us assume that $\alpha \in \boldsymbol{A}^*$ is a reversible word on a configuration $\boldsymbol{c}$. There exists a word $\beta \in \boldsymbol{A}^*$ such that the run $\boldsymbol{p} \xrightarrow{\alpha\beta} \boldsymbol{p}$ satisfies $\Delta(\alpha) + \Delta(\beta) = \boldsymbol{0}$. From this run we extract a unique witness graph $G = (\boldsymbol{Q}, T)$ such that $\boldsymbol{p} \xrightarrow{\alpha\beta}_G \boldsymbol{p}$ is a total cycle. In particular the Parikh image of this cycle is a total Kirchhoff function proving that G is reversible by Lemma 7.2.

Let I be the set of projected components of G. We introduce $a = ||\boldsymbol{A}||_\infty$ and $s = 1 + ||\boldsymbol{p}||_\infty + ||\Delta(\alpha)||_\infty$. Let $\boldsymbol{q} = \boldsymbol{p} + \Delta_I(\alpha)$. We have $||\boldsymbol{q}||_\infty \leq ||\boldsymbol{p}||_\infty + ||\Delta(\alpha)||_\infty < s$. Let us introduce $x = (1 + 2a)s$. Theorem 9.1 shows that there exists a set of indexes J such that $\pi_J(G)$ has at most x^{d^d} states and such that $\boldsymbol{p}$ forward pumpable by a cycle $\pi_J(\boldsymbol{p}) \xrightarrow{v}_{\pi_J(G)} \pi_J(\boldsymbol{p})$ and $\boldsymbol{q}$ is backward pumpable by a cycle $\pi_J(\boldsymbol{q}) \xrightarrow{w}_{\pi_J(G)} \pi_J(\boldsymbol{q})$ such that $|v|, |w| \leq dx^{d^d}$. In particular $\Delta_I(v)$ and $-\Delta_I(w)$ are two vectors in $\{\boldsymbol{c} \in \mathbb{N}_I^d \mid \boldsymbol{c}(i) \neq 0 \Leftrightarrow i \in J\}$. We deduce that for every $n \in \mathbb{N}$ we have:

$$\boldsymbol{p} \xrightarrow{v^n} \boldsymbol{p} + n\Delta_I(v) \qquad \boldsymbol{q} - n\Delta_I(w) \xrightarrow{w^n} \boldsymbol{q}$$

Since the witness graph G is reversible, Lemma 7.2 shows that $\pi_J(G)$ is reversible. From Theorem 7.3 we deduce that the zero vector is the displacement of a total Kirchhoff function μ for $\pi_J(G)$ satisfying:

$$||\mu||_\infty \leq (x^{d^d}(1+2a))^{d(d+1)} \leq x^{4d^{d+2}}$$

Note that $|\pi_J(T)| \leq |\pi_J(\boldsymbol{Q})|\;|A| \leq x^{d^d} x^d \leq x^{2d^d}$

Lemma 10.2. *There exists a cycle* $\pi_J(\boldsymbol{q}) \xrightarrow{u}_{\pi_J(G)} \pi_J(\boldsymbol{q})$ *such that* $\Delta(v) + \Delta(u) + \Delta(w) = \mathbf{0}$ *and:*

$$|u| \leq 3d\, x^{7d^{d+2}}$$

Proof. Let μ_v, μ_w be the Parikh images of $\pi_J(\boldsymbol{p}) \xrightarrow{v}_{\pi_J(G)} \pi_J(\boldsymbol{p})$ and $\pi_J(\boldsymbol{q}) \xrightarrow{w}_{\pi_J(G)} \pi_J(\boldsymbol{q})$. We introduce the function $\lambda = (1 + 2dx^{d^d})\mu - (\mu_v + \mu_w)$. Observe that λ is a Kirchhoff function for $\pi_J(G)$ satisfying $\lambda(t) \geq (1 + 2dx^{d^d}) - 2dx^{d^d} \geq 1$ for every $t \in \pi_J(T)$. The Euler's Lemma shows that λ is the Parikh image of a total cycle $\pi_J(\boldsymbol{q}) \xrightarrow{u}_{\pi_J(G)} \pi_J(\boldsymbol{q})$. Observe that $\Delta(u) = (1 + 2dx^{d^d})\mathbf{0} - (\Delta(v) + \Delta(w))$. Hence $\Delta(v) + \Delta(u) + \Delta(w) = \mathbf{0}$. The length of u is bounded by:

$$|u| = \sum_{t \in \pi_J(T)} (1 + 2dx^{d^d})\mu(t) - (\mu_v(t) + \mu_w(t)) \leq 3dx^{d^d} ||\mu||_\infty |\pi_J(T)| \leq 3dx^{7d^{d+2}} \quad \square$$

Lemma 10.3. *There exists a path* $\pi_J(\boldsymbol{p}) \xrightarrow{\tilde{\alpha}}_{\pi_J(G)} \pi_J(\boldsymbol{q})$ *such that* $\Delta(\tilde{\alpha}) = \Delta(\alpha)$ *and:*

$$|\tilde{\alpha}| \leq 2x^{7d^{d+2}}$$

Proof. Since $\pi_J(G)$ is strongly connected, there exists a path $\pi_J(\boldsymbol{q}) \xrightarrow{\tilde{\beta}}_{\pi_J(G)} \pi_J(\boldsymbol{p})$. We can assume that $|\tilde{\beta}|$ is minimal. In particular $|\tilde{\beta}| < x^{d^d}$. Moreover, we know that $\pi_J(\boldsymbol{p}) \xrightarrow{\alpha}_{\pi_J(G)} \pi_J(\boldsymbol{q})$. Observe that $\alpha\tilde{\beta}$ is the label of a cycle in $\pi_J(G)$. Hence $\boldsymbol{z} = \Delta(\alpha) + \Delta(\tilde{\beta})$ is the displacement of a Kirchhoff function for G. We have $||\boldsymbol{z}||_\infty \leq ||\Delta(\alpha)||_\infty + ||\Delta(\tilde{\beta})||_\infty \leq s + |\tilde{\beta}|a$ we get $||\boldsymbol{z}||_\infty \leq s + x^{d^d} a \leq x^{d^d}(1+a)$. Theorem 6.3 shows that $\boldsymbol{z}$ is the displacement of a Kirchhoff function θ for G such that:

$$||\theta||_\infty \leq ((x^{d^d})^{d+1} a(1+2a)^d + x^{d^d}(1+a))^d \leq x^{4d^{d+2}}$$

We introduce the Parikh image f of the path $\pi_J(\boldsymbol{q}) \xrightarrow{\tilde{\beta}}_{\pi_J(G)} \pi_J(\boldsymbol{p})$. Let us add to the strongly connected graph $\pi_J(G)$ an additional transition $t_\bullet$ from $\pi_J(\boldsymbol{q})$ to $\pi_J(\boldsymbol{p})$ and let $G_\bullet$ be this new graph and $T_\bullet = \pi_J(T) \cup \{t_\bullet\}$ be its set of transitions. Functions θ, μ and f are extended over $T_\bullet$ by $\theta(t_\bullet) = \mu(t_\bullet) = f(t_\bullet) = 0$. We also introduce the Parikh image $f_\bullet$ of $t_\bullet$, i.e. $f_\bullet(t_\bullet) = 1$ and $f_\bullet(t) = 0$ for every $t \in \pi_J(T)$. Let us observe that $g = \theta + x^{d^d}\mu - f + f_\bullet$ is a Kirchhoff function for $G_\bullet$. Since $f(t) < x^{d^d}$ we deduce that $g(t) \geq 1$ for every $t \in \pi_J(T)$. Euler's Lemma shows that g is the Parikh image of a total cycle. Since $g(t_\bullet) = 1$ we deduce that g is the Parikh image of a cycle of the form $(\pi_J(\boldsymbol{p}) \xrightarrow{\tilde{\alpha}} \pi_J(\boldsymbol{q}))\, t_\bullet$. By definition of g we get $\Delta(\tilde{\alpha}) = \boldsymbol{z} + x^{d^d}\mathbf{0} - \Delta(\tilde{\beta})$. Hence $\Delta(\tilde{\alpha}) = \boldsymbol{z} - \Delta(\tilde{\beta})$. Since $\boldsymbol{z} = \Delta(\alpha) + \Delta(\tilde{\beta})$ we get $\Delta(\tilde{\alpha}) = \Delta(\alpha)$. The following inequalities provide the lemma:

$$|\tilde{\alpha}| \leq |\pi_J(T)|\;||\theta||_\infty + x^{d^d} |\pi_J(T)|\;||\mu||_\infty \leq 2x^{7d^{d+2}} \quad \square$$

Lemma 10.4. *For every $n \geq |u|a$ we have:*

$$\boldsymbol{q} + n\Delta_I(v) \xrightarrow{u^n} \boldsymbol{q} - n\Delta_I(w)$$

Proof. Let $n \geq |u|a$. We introduce the sequence $(\boldsymbol{x}_k)_{0 \leq k \leq n}$ of configurations $\boldsymbol{x}_k = \boldsymbol{q} + (n-k)\Delta_I(v) - k\Delta_I(w)$. Since $\pi_J(\boldsymbol{x}_k) = \pi_J(\boldsymbol{q})$ we deduce that there exists a run from $\pi_J(\boldsymbol{x}_k)$ labelled by u. Moreover as $\Delta_I(v)(j) \geq 1$ and $-\Delta_I(w)(j) \geq 1$ for every $j \in J$, we deduce that $\boldsymbol{x}_k(j) \geq n \geq |u_p|a$ for every $j \in J$. Lemma 3.1 shows that there exists a run from $\boldsymbol{x}_k$ labelled by u. Since $\Delta(v) + \Delta(u) + \Delta(w) = \boldsymbol{0}$ we get $\boldsymbol{x}_k \xrightarrow{u} \boldsymbol{x}_{k+1}$. □

Lemma 10.5. *For every $n \geq |\tilde{\alpha}|a$ we have:*

$$\boldsymbol{p} + n\Delta_I(v) \xrightarrow{\tilde{\alpha}} \boldsymbol{q} + n\Delta_I(v)$$

Proof. Observe that $\pi_J(\boldsymbol{p} + n\Delta_I(v)) = \pi_J(\boldsymbol{p})$ and $\pi_J(\boldsymbol{p}) \xrightarrow{\tilde{\alpha}} \pi_J(\boldsymbol{q})$. Moreover for every $j \in J$ we have $(\boldsymbol{p} + n\Delta_I(v))(j) \geq n \geq |\tilde{\alpha}|a$. From Lemma 3.1 we deduce that there exists a run from $\boldsymbol{p} + n\Delta_I(v)$ labelled by $\tilde{\alpha}$. From $\boldsymbol{p} \xrightarrow{\alpha} \boldsymbol{q}$ we deduce that $\boldsymbol{p} + \Delta_I(\alpha) = \boldsymbol{q}$. Since $\Delta(\alpha) = \Delta(\tilde{\alpha})$ we deduce that $\boldsymbol{p} + \Delta_I(\tilde{\alpha}) = \boldsymbol{q}$. We deduce the run $\boldsymbol{p} + n\Delta_I(v) \xrightarrow{\tilde{\alpha}} \boldsymbol{q} + n\Delta_I(v)$. □

Finally, let $n = a\max\{|\tilde{\alpha}|, |u|\}$. We have proved that $\boldsymbol{p} \xrightarrow{\alpha'} \boldsymbol{q}$ where $\alpha' = v^n\tilde{\alpha}u^nw^n$. Note that $\Delta(\alpha') = \Delta(\alpha)$ since $\Delta(\tilde{\alpha}) = \Delta(\alpha)$ and $\Delta(v) + \Delta(u) + \Delta(w) = \boldsymbol{0}$. We deduce that $\Delta(\alpha') = \Delta(\alpha)$. As $\boldsymbol{q} \xrightarrow{\beta} \boldsymbol{p}$ with $\Delta(\alpha) + \Delta(\beta) = \boldsymbol{0}$ we deduce that α' is reversible on $\boldsymbol{p}$. Note that $n \leq a3dx^{7d^{d+2}} \leq 3dx^{8d^{d+2}}$. Hence we have:

$$|\alpha'| \leq 2x^{7d^{d+2}} + 3dx^{8d^{d+2}}(2dx^{d^d} + 3dx^{7d^{d+2}}) \leq 17d^2x^{15d^{d+2}}$$

We have proved Theorem 10.1.

Corollary 10.6. *Two standard configurations $\boldsymbol{p}, \boldsymbol{q}$ are is the same strongly connected component of a standard subreachability graph if and only if there exist runs $\boldsymbol{p} \xrightarrow{\alpha} \boldsymbol{q}$ and $\boldsymbol{q} \xrightarrow{\beta} \boldsymbol{p}$ such that:*

$$|\alpha|, |\beta| \leq 17d^2x^{15d^{d+2}}$$

where $x = (1 + 2||\boldsymbol{A}||_\infty)(1 + 2\max\{||\boldsymbol{p}||_\infty, ||\boldsymbol{q}||_\infty\})$.

Theorem 10.7. *The reversible reachability problem is EXPSPACE-complete.*

11 Application : Reversibility Domains

During the execution of a VAS some actions are reversible and some not. More precisely, let $\boldsymbol{D_a}$ be the set of standard configurations $\boldsymbol{c}$ such that there exists a word α satisfying $\boldsymbol{c} \xrightarrow{a} \boldsymbol{c} + \boldsymbol{a} \xrightarrow{\alpha} \boldsymbol{c}$. We observe that the set $\boldsymbol{D_a}$ is an upward closed set for the order $\leq$. In fact $\boldsymbol{c} \xrightarrow{a} \boldsymbol{c} + \boldsymbol{a} \xrightarrow{\alpha} \boldsymbol{c}$ implies the same thing by replacing $\boldsymbol{c}$ with a standard configuration $\boldsymbol{x} \in \boldsymbol{c} + \mathbb{N}^d$. So $\boldsymbol{D_a}$ is characterized by its finite set of minimal elements $\min(\boldsymbol{D_a})$ for $\leq$. As an application of Theorem 10.1, we obtain the following result.

Theorem 11.1. *Configurations* $\boldsymbol{c} \in \min(\boldsymbol{D_a})$ *satisfy the following inequality where* $a = ||\boldsymbol{A}||_\infty$.

$$||\boldsymbol{c}||_\infty \leq (102d^2a^2)^{(15d^{d+2})^{d+2}}$$

Proof. Observe that if $a = 0$ we are done since in this case $\boldsymbol{c} = \boldsymbol{0}$. So we can assume that $a \geq 1$. We introduce the extractor $\lambda = (\lambda_1, \ldots, \lambda_d)$ defined by $\lambda_{d+1} = a$ and the following induction for $n \in \{1, \ldots, d+1\}$:

$$\lambda_{n-1} = 17d^2(6a\lambda_n)^{15d^{d+2}}$$

Let $\boldsymbol{c} \in \min(\boldsymbol{D_a})$ and let $\boldsymbol{d} = \boldsymbol{c} + \boldsymbol{a}$. Let us consider the minimal excluding set I for $(\lambda, \{\boldsymbol{d}\})$. By minimality of I we have $\boldsymbol{d}(i) < \lambda_{|I|+1}$ for every $i \not\in I$ and $\boldsymbol{d}(i) \geq \lambda_{|I|}$ for every $i \in I$. We consider the standard configuration $\boldsymbol{y}$ defined by $\boldsymbol{y}(i) = \lambda_{|I|}$ if $i \in I$ and $\boldsymbol{y}(i) = \boldsymbol{d}(i)$ if $i \not\in I$. Let us consider $\boldsymbol{q} = \pi_I(\boldsymbol{c})$ and $\boldsymbol{p} = \pi_I(\boldsymbol{d})$. Since $\boldsymbol{c} \in \boldsymbol{D_a}$ there exists a run $\boldsymbol{d} \xrightarrow{\alpha} \boldsymbol{c}$. In particular $\boldsymbol{p} \xrightarrow{\alpha} \boldsymbol{q} \xrightarrow{\boldsymbol{a}} \boldsymbol{p}$ with $\Delta(\alpha) + \Delta(\boldsymbol{a}) = \boldsymbol{0}$. We deduce that α is reversible on $\boldsymbol{p}$ and Theorem 10.1 shows that there exists a word α' such that $\boldsymbol{p} \xrightarrow{\alpha'} \boldsymbol{q}$, $\Delta(\alpha') = \Delta(\alpha)$ and:

$$|\alpha'| \leq 17d^2x^{15d^{d+2}}$$

where $x = (1+2a)(1 + ||\boldsymbol{p}||_\infty + ||\boldsymbol{a}||_\infty)$. Note that $||\boldsymbol{p}||_\infty \leq \lambda_{|I|+1} - 1$. We deduce that $x \leq (1+2a)(\lambda_{|I|+1} + a) \leq 6a\lambda_{|I|+1}$ since $1 \leq a$ and $a \leq \lambda_{|I|+1}$. Hence $a|\alpha'| \leq \lambda_{|I|}$ thanks to the induction defining λ. Since $\pi_I(\boldsymbol{y}) = \boldsymbol{p}$ we deduce that there exists a run from $\pi_I(\boldsymbol{y})$ labelled by α'. As $\boldsymbol{y}(i) \geq \lambda_{|I|} \geq a|\alpha'|$ for every $i \in I$, Lemma 3.1 shows that there exists a run $\boldsymbol{y} \xrightarrow{\alpha'} \boldsymbol{x}$. Since $\Delta(\alpha') = \Delta(\alpha) = -\boldsymbol{a}$ we deduce that $\boldsymbol{x} = \boldsymbol{y} - \boldsymbol{a}$. From $\boldsymbol{y} \leq \boldsymbol{d}$ we get $\boldsymbol{x} \leq \boldsymbol{c}$ by subtracting $\boldsymbol{a}$. Moreover as $\boldsymbol{x} \xrightarrow{\boldsymbol{a}} \boldsymbol{y} \xrightarrow{\alpha'} \boldsymbol{x}$ we deduce that $\boldsymbol{x} \in \boldsymbol{D_a}$. By minimality of $\boldsymbol{c}$ we get $\boldsymbol{c} = \boldsymbol{x}$. Hence $\boldsymbol{c} = \boldsymbol{y} - \boldsymbol{a}$. In particular $||\boldsymbol{c}||_\infty \leq \lambda_{|I|} + a \leq \lambda_0 + a$. Finally let us get a bound on λ_0. We get the equality $\lambda_{n-1} = c\lambda_n^e$ by introducing $e = 15d^{d+2}$ and $c = 17d^2(6a)^e$. Hence $\lambda_0 \leq (ca)^{e^{d+1}} \leq (102d^2a^2)^{e^{d+2}}$ and from $e^{d+2} \leq (15d^{d+2})^{d+2}$ we are done. □

12 Conclusion

The reversible reachability problem is proved to be EXPSPACE-complete in this paper. The proof is inspired by the Rackoff and Kosaraju ideas [Rac78, Kos82]. We have introduced the domain of reversibility $\boldsymbol{D_a}$ of every action $\boldsymbol{a} \in \boldsymbol{A}$. Observe that the reflexive and transitive closure of the following relation R is a congruence and from [BF97] we deduce that this relation is definable in the Presburger arithmetic. That means there exist a Preburger formula ϕ that exactly denotes the pair $(\boldsymbol{x}, \boldsymbol{y})$ of standard configurations in the reversible reachability relation. As a future work we are interested in characterizing precisely the size of such a formula (we already derive an elementary bound from [BF97] and Theorem 11.1).

$$R = \bigcup_{\boldsymbol{a} \in \boldsymbol{A}} \{(\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{a}) \mid \boldsymbol{x} \in \boldsymbol{D_a}\}$$

Such a formula will provide a first hint on the structure of the *production relations* introduced in [Ler11] for solving the general vector addition system reachability problem.

References

[BF97] Bouziane, Z., Finkel, A.: Cyclic petri net reachability sets are semi-linear effectively constructible. Electr. Notes Theor. Comput. Sci. 9 (1997)

[CLM76] Cardoza, E., Lipton, R.J., Meyer, A.R.: Exponential space complete problems for petri nets and commutative semigroups: Preliminary report. In: STOC 1976, pp. 50–54. ACM, New York (1976)

[EN94] Esparza, J., Nielsen, M.: Decidability issues for petri nets - a survey. Bulletin of the European Association for Theoretical Computer Science 52, 245–262 (1994)

[Hau90] Hauschildt, D.: Semilinearity of the Reachability Set is Decidable for Petri Nets. PhD thesis, University of Hamburg (1990)

[Kos82] Rao Kosaraju, S.: Decidability of reachability in vector addition systems (preliminary version). In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC 1982), San Francisco, California, USA, May 5-7, pp. 267–281. ACM, New York (1982)

[Lam92] Lambert, J.L.: A structure to decide reachability in petri nets. Theoretical Computer Science 99(1), 79–104 (1992)

[Ler11] Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, pp. 307–316. ACM, New York (2011)

[May81] Mayr, E.W.: An algorithm for the general petri net reachability problem. In: Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation, STOC 1981, Milwaukee, Wisconsin, USA, May 11-13, pp. 238–246. ACM, New York (1981)

[Pot91] Pottier, L.: Minimal solutions of linear diophantine systems: Bounds and algorithms. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 162–173. Springer, Heidelberg (1991)

[Rac78] Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoretical Computer Science 6(2) (1978)

[ST77] Sacerdote, G.S., Tenney, R.L.: The decidability of the reachability problem for vector addition systems (preliminary version). In: Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colorado, USA, May 2-4, pp. 61–76. ACM, New York (1977)

Efficient Contextual Unfolding*

César Rodríguez[1], Stefan Schwoon[1], and Paolo Baldan[2]

[1] LSV, ENS Cachan & CNRS, INRIA Saclay, France
[2] Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy

Abstract. A contextual net is a Petri net extended with read arcs, which allow transitions to check for tokens without consuming them. Contextual nets allow for better modelling of concurrent read access than Petri nets, and their unfoldings can be exponentially more compact than those of a corresponding Petri net. A constructive but abstract procedure for generating those unfoldings was proposed in earlier work; however, no concrete implementation existed. Here, we close this gap providing two concrete methods for computing contextual unfoldings, with a view to efficiency. We report on experiments carried out on a number of benchmarks. These show that not only are contextual unfoldings more compact than Petri net unfoldings, but they can be computed with the same or better efficiency, in particular with respect to the place-replication encoding of contextual nets into Petri nets.

1 Introduction

Petri nets are a means for reasoning about concurrent, distributed systems. They explicitly express notions such as concurrency, causality, and independence.

The unfolding of a Petri net is, essentially, an acyclic version of the net in which loops have been unrolled. The unfolding is infinite in general, but for finite-state Petri nets one can construct a finite complete prefix of it that completely represents the behaviour of the system, and whose acyclic structure permits easier analyses. This prefix is typically much smaller than the number of reachable markings because an unfolding exploits the inherently concurrent nature of the underlying system; loosely speaking, the more concurrency there is in the net, the more advantages unfoldings have over reachability-graph techniques.

Petri net unfoldings may serve as a basis for further analyses. There is a large body of work describing their construction, their properties, and their use in various fields (see, e.g., [6] for an extensive survey).

However, Petri nets are not well-suited to model concurrent read access, that is, multiple actions requiring non-exclusive access to one common resource. Consequently, the unfolding technique becomes inefficient in such situations. It is possible to mitigate this problem with a place-replication (PR) encoding [17]. Here, a resource with n readers is duplicated n times, and each reader obtains a "private" copy. However, the resulting unfolding may still be exponential in n.

* Supported by Fundación Caja Madrid, the MIUR project SisteR, and the University of Padua project AVIAMO.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 342–357, 2011.

Contextual nets explicitly model concurrent read accesses and address this problem. They extend Petri nets with *read arcs*, allowing an action to check for the presence of a resource without consuming it. They have been used, e.g., to model concurrent database access [13], concurrent constraint programs [12], priorities [9], and asynchronous circuits [17]. Their accurate representation of concurrency makes contextual unfoldings up to exponentially smaller in the presence of multiple readers, which promises to yield more efficient analysis procedures.

While the properties and construction of ordinary Petri net unfoldings are well-understood, research on how to construct and exploit the properties of contextual unfoldings has been lacking so far. Contextual unfoldings are introduced in [17,1], and a first unfolding procedure for a restricted subclass can be found in [17]. A general but non-constructive procedure is proposed in [18].

A constructive, general solution was finally given in [3], at the price of making the underlying theory notably more complicated. In particular, computing a complete prefix required to annotate every event e with a subset of its histories, where roughly speaking, a history of e is a set of events that must precede e in any execution. However, it remained unclear whether the approach could be implemented with reasonable efficiency, and how. For 1-safe nets, the interest of computing a complete contextual prefix was doubtful: while the prefix can be exponentially smaller than the complete prefix of the corresponding PR-encoding, the intermediate product used to produce it has asymptotically the same size. More precisely, the number of histories in the contextual prefix matches the number of events in the PR-prefix (for general k-safe nets, this is not the case).

In [2], first theoretical advances towards an efficient implementation were made, proposing to annotate not only events, but conditions with histories. This gave rise to a binary concurrency relation, a concept that mimics a crucial element of efficient Petri unfolding tools [16,10]. However, an implementation was still lacking, so the above doubts persisted.

In this paper, we address these open issues with the following contributions:

- We provide new approaches to two key elements of an unfolding tool: the computation of possible extensions and maintaining a concurrency relation.
- We generalise the results in [3,2] in order to deal with a slight generalization of the adequate orders from [7]. Although not very surprising, this extension is quite relevant in practice as it drastically reduces the resulting prefixes.
- We implemented both approaches, aiming for efficiency. The resulting tool, called Cunf [14], matches dedicated Petri net unfolders like Mole [16] on pure Petri nets and additionally handles contextual unfoldings. The development of such a tool was non-trivial: First, the new unfolder is not a simple extension of an existing one because the presence of histories influences the data structures at every level. Secondly, even a Petri unfolder has complicated data structures, and its computation requires to solve subproblems that are computationally hard in principle [8].
- We ran the tool on a set of benchmarks and report on the experiments, for both approaches. In particular, it turns out that, even for 1-safe nets, our construction of contextual unfoldings is faster than that for PR-unfoldings.

Apart from details of the prefix computation, our main message is that efficient contextual unfolding is possible and performs better than the PR-encoding, even for 1-safe nets. Contextual nets and their unfoldings therefore have a rightful place in research on concurrency, including from an efficiency point of view. A full version of this paper including all the proofs can be found at [15].

2 Basic Notions

A *contextual net (c-net)* is a tuple $N = \langle P, T, F, C, m_0 \rangle$, where P and T are disjoint sets of *places* and *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and $C \subseteq P \times T$ is the *context relation*. A pair $(p, t) \in C$ is called *read arc*. Any function $m \colon P \to \mathbb{N}$ is called a *marking*, and m_0 is the *initial marking*. A *Petri net* is a c-net without any read arcs.

For $x \in P \cup T$, we call ${}^\bullet x := \{y \in P \cup T \mid (y, x) \in F\}$ the *preset* of x and $x^\bullet := \{y \in P \cup T \mid (x, y) \in F\}$ the *postset* of x. The *context* of a place p is defined as $\underline{p} := \{\, t \in T \mid (p, t) \in C \,\}$, and the context of a transition t as $\underline{t} := \{\, p \in P \mid (p, t) \in C \,\}$. These notions are extended to sets in the usual fashion.

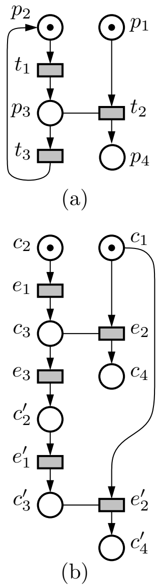
A marking m is *n-safe* if $m(p) \leq n$ for all $p \in P$. A set $A \subseteq T$ of transitions is *enabled* at m if for all $p \in P$,

$$m(p) \geq |p^\bullet \cap A| + \begin{cases} 1 & \text{if } \underline{p} \cap A \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Such A can *occur* or *be executed*, leading to a new marking m', where $m'(p) = m(p) - |p^\bullet \cap A| + |{}^\bullet p \cap A|$ for all $p \in P$. We call $\langle m, A, m' \rangle$ a *step* of N.

A finite sequence of transitions $\sigma = t_1 \ldots t_n \in T^*$ is a *run* if there exist markings $m_1, \ldots, m_n$ such that $\langle m_{i-1}, \{t_i\}, m_i \rangle$ is a step for $1 \leq i \leq n$, and m_0 is the initial marking of N; if such a run exists, m_n is said to be *reachable*. A c-net N is said to be n-safe if every reachable marking of N is n-safe.

Fig. 1 (a) depicts a 1-safe c-net. Read arcs are drawn as undirected lines. For t_2, we have $\{p_1\} = {}^\bullet t_2$, $\{p_3\} = \underline{t_2}$ and $\{p_4\} = t_2^\bullet$.



(a)

(b)

Fig. 1. (a) A 1-safe c-net; and (b) an unfolding prefix

General assumptions. We restrict our interest to finite 1-safe c-nets and treat markings as sets of places. Furthermore, for any c-net $N = \langle P, T, C, F, m_0 \rangle$ we assume for all transitions $t \in T$ that ${}^\bullet t \cap \underline{t} = \emptyset$; notice that transitions violating this condition can never fire in 1-safe nets.

2.1 Encodings of Contextual Nets

A c-net N can be encoded into a Petri net whose reachable markings are in one-to-one correspondence with those of N. We treat two such encodings, and

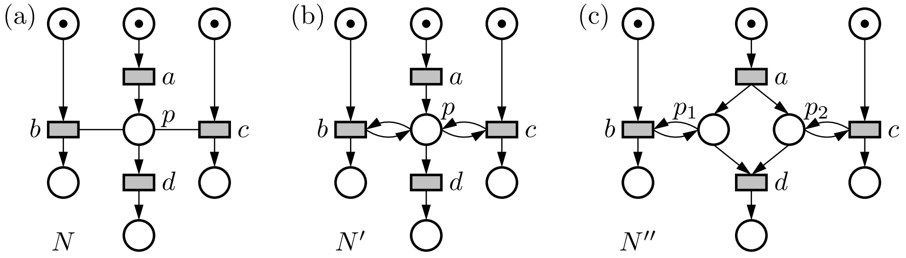


Fig. 2. C-net N, its *plain encoding* N' and its *Place-Replication encoding* N''

illustrate them by the c-net N in Fig. 2 (a). Place p has two transitions b, c in its context, modelling a situation where, e.g., two processes are read-accessing a common resource modelled by p. Note that step $\{b, c\}$ can occur in N.

Plain encoding. Given a c-net N, the *plain encoding* of N is the net N' obtained by replacing every read arc (p, t) in the context relation by a read/write loop $(p, t), (t, p)$ in the flow relation. The net N' has the same reachable markings as N; it also has the same runs but not the same steps as N. An example can be found in Fig. 2 (b). Note that the step $\{b, c\}$ can no longer occur in N', as the firings of $\{b\}$ and $\{c\}$ are sequentialized.

PR-encoding. The *place-replication (PR-) encoding* [17] of a c-net N is a Petri net N'' in which we substitute every place p read by $n \geq 1$ transitions $t_1, \ldots, t_n$ by places $p_1, \ldots, p_n$, updating the flow relation of N'' as follows. For $i \in \{1, \ldots, n\}$,

1. transition t_i consumes and produces place p_i, i.e., $p_i \in {}^\bullet t_i$ and $p_i \in t_i^\bullet$;
2. any transition t producing p in N produces p_i in N'', i.e., $p_i \in t^\bullet$;
3. any transition t consuming p in N consumes p_i in N'', i.e., $p_i \in {}^\bullet t$.

A PR-encoding is depicted in Fig. 2 (c). Reachable markings, runs, and steps of N'' are in one-to-one correspondence to those of N.

3 Contextual Unfoldings and Their Prefixes

In this section, we mostly recall basic definitions from [3] concerning unfoldings. We fix a 1-safe c-net $N = \langle P, T, F, C, m_0 \rangle$ for the rest of the section. Intuitively, the unfolding of N is a safe acyclic c-net where loops of N are "unrolled"; in general, this structure is infinite.

Definition 1. *The unfolding of N, written $\mathcal{U}_N$, is a c-net $(B, E, G, D, \widehat{m}_0)$ equipped with a mapping $f \colon (B \cup E) \to (P \cup T)$, which we extend to sets and sequences in the usual way. We call the elements of B* conditions*, and those of E* events*; f maps conditions to places and events to transitions.*

Conditions will take the form $\langle p, e' \rangle$, where $p \in P$ and $e' \in E \cup \{\bot\}$, and events will take the form $\langle t, M \rangle$, where $t \in T$ and $M \subseteq B$. We shall assume

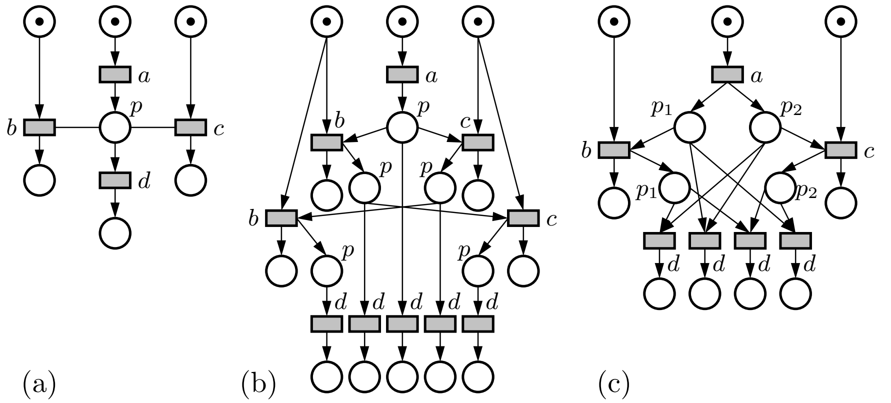


Fig. 3. Unfoldings of N, N', and N'' from Fig. 2

$f(\langle p,e'\rangle) = p$ and $f(\langle t,M\rangle) = t$, respectively. A set M of conditions is called concurrent*, written $conc(M)$, iff $\mathcal{U}_N$ has a reachable marking M' s.t. $M' \supseteq M$.*

$\mathcal{U}_N$ is the smallest net containing the following elements:

- *if $p \in m_0$, then $\langle p, \bot\rangle \in B$ and $\langle p, \bot\rangle \in \widehat{m}_0$;*
- *for any $t \in T$ and disjoint pair of sets $M_1, M_2 \subseteq B$ such that $conc(M_1 \cup M_2)$, $f(M_1) = {}^\bullet t$, $f(M_2) = \underline{t}$, we have $e := \langle t, M_1 \cup M_2\rangle \in E$, and for all $p \in t^\bullet$, we have $\langle p, e\rangle \in B$. Moreover, G and D are such that ${}^\bullet e = M_1$, $\underline{e} = M_2$, and $e^\bullet = \{\, \langle p,e\rangle \mid p \in t^\bullet \,\}$.*

Fig. 3 shows unfoldings of the nets from Fig. 2, where f is indicated by the labels of conditions and events. In this case, the c-net is isomorphic to its unfolding; crucially, it is smaller than the unfoldings of its two encodings. Call events labelled by b and c "readers", and events labelled by d "consumers". If, in Fig. 2, we replaced b, c by n transitions reading from p, there would be n readers and one consumer in the contextual unfolding; $\mathcal{O}(n!)$ readers *and* consumers in the plain unfolding; and n readers but 2^n consumers in the PR-unfolding.

$\mathcal{U}_N$ represents all possible behaviours of N, and, in particular m is reachable in N iff some $\widehat{m}$ with $f(\widehat{m}) = m$ is reachable in $\mathcal{U}_N$. Intuitively, the plain unfolding explodes because it represents the step $\{b, c\}$ of the c-net by two runs; and the cycles in the PR-encoding mean more consuming events for the PR-unfolding.

Definition 2. *The* causality relation *on $\mathcal{U}_N$, denoted $<$, is the transitive closure of $G \cup \{\, (e,e') \in E \times E \mid e^\bullet \cap \underline{e'} \neq \emptyset \,\}$. For $x \in B \cup E$, we write $[x]$ for the set of* causes *of x, defined as $\{\, e \in E \mid e \leq x \,\}$, where $\leq$ is the reflexive closure of $<$.*

In Fig. 1 (b), we have, e.g., $c_2 < e_1$, $e_1 < e_2$, and $c_2 < e_2$. The causality relation between a pair of events $e < e'$ captures the intuition that e must occur before e' in any run that fires e'.

Definition 3. *A set $X \subseteq E$ is called* causally closed *if $[e] \subseteq X$ for all $e \in X$. A* prefix *of $\mathcal{U}_N$ is a net $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0 \rangle$ such that $E' \subseteq E$ is causally closed, $B' = \widehat{m}_0 \cup (E')^\bullet$, and G', D' are the restrictions of G, D to $(B' \cup E')$.*

In other words, a prefix is a causally-closed subnet of $\mathcal{U}_N$. Surely, if $\mathcal{P}$ is a prefix and $\widehat{m}$ a marking reachable in it, then $f(\widehat{m})$ is reachable in N. We are interested in computing a prefix for which the inverse also holds.

Definition 4. *A prefix $\mathcal{P}$ is called* complete *if for all markings m, m is reachable in N iff there exists a marking $\widehat{m}$ reachable in $\mathcal{P}$ such that $f(\widehat{m}) = m$.*

A complete prefix thus preserves all behavioural information about N, while being typically smaller than its reachability graph; yet its acyclic structure makes the reachability problem easier than for N itself [11]. Moreover, as we saw in Fig. 3, a contextual unfolding is more succinct than its corresponding Petri net unfolding. Other papers, e.g., [17], consider a slightly stronger notion of completeness imposing that not only reachable markings, but also firable transitions have a representative in the prefix. That would not affect the results in this paper.

4 Constructing Finite Prefixes

In this section, we make inroads on how to construct a finite prefix. The material from this section mostly recalls elements from [3], with minor modifications. We fix a net N and its unfolding $\mathcal{U}_N$ as in Section 3.

Consider events e_2 and e_3 in Fig. 1 (b). Clearly, $e_2 < e_3$ does not hold. However, any run that fires *both* e_2 and e_3 will fire e_2 before e_3 (since e_3 consumes c_3). This situation arises due to read arcs and motivates the next definition.

Definition 5. *Two events $e, e' \in E$ are in* asymmetric conflict*, written $e \nearrow e'$, iff (i) $e < e'$, or (ii) $\underline{e} \cap {}^\bullet e' \neq \emptyset$, or (iii) $e \neq e'$ and ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$. For a set of events $X \subseteq E$, we write $\nearrow_X$ to denote the relation $\nearrow \cap (X \times X)$.*

Asymmetric conflict can be thought of as a scheduling constraint: if both e, e' occur in a run, then e must occur first. Note that in case (iii) this is vacuously the case, as e, e' cannot both occur. Thus, by condition (iii) $\nearrow$ subsumes the symmetric conflicts known from Petri net unfoldings as loops of length two.

Definition 6. *A* configuration *of $\mathcal{U}_N$ is a finite, causally closed set of events $\mathcal{C}$ such that $\nearrow_{\mathcal{C}}$ is acyclic. $Conf(\mathcal{U}_N)$ denotes the set of all such configurations.*

A set of events is a configuration iff all its events can be ordered to form a run that respects the scheduling constraints given by $\nearrow$. We say that configuration $\mathcal{C}$ *evolves* to configuration $\mathcal{C}'$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, iff $\mathcal{C} \subseteq \mathcal{C}'$ and $\neg(e' \nearrow e)$ for all $e \in \mathcal{C}$ and $e' \in \mathcal{C}' \setminus \mathcal{C}$. Intuitively, a run of $\mathcal{C}$ can be extended into a run of $\mathcal{C}'$.

Configurations $\mathcal{C}$, $\mathcal{C}'$ are said to be in *conflict*, written $\mathcal{C} \# \mathcal{C}'$, when there is no configuration $\mathcal{C}''$ verifying $\mathcal{C} \sqsubseteq \mathcal{C}''$ and $\mathcal{C}' \sqsubseteq \mathcal{C}''$. Note that if two configurations are *not* in conflict, then their union is a configuration.

The *cut* of a configuration $\mathcal{C}$ is the marking reached in $\mathcal{U}_N$ by a run of $\mathcal{C}$. We define $\mathsf{Cut}(\mathcal{C}) := (\widehat{m}_0 \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$. The *marking* of $\mathcal{C}$ is its image through f: $\mathsf{Mark}(\mathcal{C}) := f(\mathsf{Cut}(\mathcal{C}))$.

Definition 7. *Let e be an event. If $\mathcal{C}$ is a configuration with $e \in \mathcal{C}$, we define the configuration $\mathcal{C}[\![e]\!] := \{\, e' \in \mathcal{C} \mid e'(\nearrow_{\mathcal{C}})^* e \,\}$ as the* history of e in $\mathcal{C}$. *Moreover, $Hist(e) := \{\, \mathcal{C}[\![e]\!] \mid \mathcal{C} \in Conf(\mathcal{U}_N) \wedge e \in \mathcal{C} \,\}$ is the set of histories of e.*

While in Petri net unfoldings each event has exactly one history, a contextual unfolding may have multiple (even infinitely many) histories per event. For instance, in Fig. 1 (b) $Hist(e_3) = \{\{e_1, e_3\},\ \{e_1, e_2, e_3\}\}$. To compute a complete prefix, one annotates events with a finite subset of their histories.

Definition 8. *An* enriched event *is a pair $\langle e, H\rangle$ where $e \in E$ and $H \in Hist(e)$. A* closed enriched prefix (CEP) *of $\mathcal{U}_N$ is a pair $\mathcal{E} = \langle \mathcal{P}, \chi\rangle$ such that $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0\rangle$ is a prefix and $\chi\colon E' \to 2^{2^E}$ satisfies for all $e \in E'$ (i) $\emptyset \neq \chi(e) \subseteq Hist(e)$, and (ii) $H \in \chi(e)$ and $e' \in H$ imply $H[\![e']\!] \in \chi(e')$. For an enriched event $\langle e, H\rangle$, we write $\langle e, H\rangle \in \mathcal{E}$ if $e \in E'$ and $H \in \chi(e)$.*

In [3], a complete prefix of $\mathcal{U}_N$ is constructed by a saturation procedure that adds one enriched event at a time until there remains no addition that would "contribute" new markings. We concretize this idea in the following:

Definition 9. *Let $\mathcal{E}$ be a CEP. An enriched event $\langle e, H\rangle$ is a* possible extension *of $\mathcal{E}$ iff $\langle e', H[\![e']\!]\rangle \in \mathcal{E}$ for all $e' \in H$, $e' \neq e$, but $\langle e, H\rangle \notin \mathcal{E}$.*

Let $\prec$ be a partial order among configurations verifying that $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C} \neq \mathcal{C}'$ implies $\mathcal{C} \prec \mathcal{C}'$. We extend $\prec$ to enriched events by $\langle e, H\rangle \prec \langle e', H'\rangle$ if $H \prec H'$. Given a fixed $\prec$, a tuple $\langle e, H\rangle$ is called *cutoff* iff there exists an enriched event $\langle e', H'\rangle$ such that $\mathsf{Mark}(H') = \mathsf{Mark}(H)$ and $\langle e', H'\rangle \prec \langle e, H\rangle$. Thus, $\prec$ parametrizes the following informal algorithm:

Algorithm 1.

- Start with the CEP that contains just $\widehat{m}_0$;
- Then, in each iteration, add a non-cutoff $\prec$-minimal possible extension.
- If no non-cutoff possible extensions remain, terminate.

Whether Algorithm 1 terminates with a *complete* prefix depends on the choice of $\prec$. It was shown in [3,2] that the procedure above yields a complete prefix if $\prec$ is the partial order due to McMillan [11]. However, it is known for Petri net unfoldings that using a total, so-called *adequate* order as defined in [7] can result in up to exponentially smaller complete prefixes.

Proposition 1. *Let N be a 1-safe c-net. If $\prec$ is adequate, then Algorithm 1 terminates with a CEP $\mathcal{E} = \langle \mathcal{P}, \chi\rangle$ such that $\mathcal{P}$ is a complete prefix of $\mathcal{U}_N$.*

5 Two Approaches to Possible Extensions and Concurrency

We now turn to the question of how to implement Algorithm 1 efficiently, for constructing unfoldings in practice. The main computational problem is to identify the possible extensions at each iteration of the procedure. Let N and $\mathcal{U}_N$ be as in the previous sections.

For Petri net unfolders (which do not deal with histories) this involves identifying sets M of conditions such that $conc(M)$ and $f(M) = {}^\bullet t$ for some $t \in T$ (compare Definition 1). For Petri nets, it is known that $conc(M)$ holds iff $conc(\{c_1, c_2\})$ for all pairs $c_1, c_2 \in M$. Possible extensions can therefore be identified by repeatedly consulting a *binary* relation on conditions. Moreover, this binary relation can be computed efficiently and incrementally during prefix construction. This idea is exploited by existing tools such as Mole [16] or Punf [10].

The above statement about $conc(\cdot)$ was shown to be invalid for contextual unfoldings in [3]. However, one can define a binary relation with similar properties on conditions enriched with histories.

Definition 10. *Let c be a condition. A* generating history *of c is $\emptyset$ if $c \in \widehat{m}_0$, or $H \in Hist(e)$, where $\{e\} = {}^\bullet c$. A* reading history *of c is any $H \in Hist(e)$ such that $e \in \underline{c}$. A* history *of c is any of its generating or reading histories or $H_1 \cup H_2$, where H_1 and H_2 are histories of c verifying $\neg(H_1 \# H_2)$. In the latter case, the history is called* compound*.*

If H is a history of c, we call $\langle c, H\rangle$ an *enriched condition*, called generating, reading, or compound condition, according to H[1]. For a CEP $\mathcal{E} = \langle \mathcal{P}, \chi\rangle$, we say $\langle c, H\rangle \in \mathcal{E}$ if H is built from histories in χ. The mapping f is extended to enriched events and conditions by $f(\langle e, H\rangle) = f(e)$ and $f(\langle c, H\rangle) = f(c)$.

Definition 11. *Two enriched conditions $\langle c, H\rangle, \langle c', H'\rangle$ are called* concurrent*, written $\langle c, H\rangle \parallel \langle c', H'\rangle$, iff $\neg(H \# H')$ and $c, c' \in \mathsf{Cut}(H \cup H')$.*

In Section 5.1, we discuss how $\parallel$ helps to compute possible extensions. In Section 5.2 we then discuss how to update $\parallel$ during the unfolding construction.

5.1 Computing Possible Extensions

We discuss two ways of computing possible extensions. The first, called "lazy", avoids constructing compound conditions (see Definition 10), reducing the number of enriched conditions considered. The second, "eager" approach does use compound conditions, saving work while computing possible extensions instead. The lazy approach was introduced in [2] for the McMillan order, but holds also for the total order of [7]. The eager approach is proposed for the first time here.

[1] In [2], generating histories were called *causal*; we find the term generating more suggestive. The definition of compound histories is new and does not appear in [2].

Lazy Approach. The lazy approach [2] is based on the observation that the history associated with an event can be constructed by taking generating and read histories for places in the pre-set and generating histories for places in the context. This is expressed by the following proposition:

Proposition 2. *[2] The pair $\langle e, H\rangle$ with $f(e) = t$ is an enriched event iff there exist sets X_p, X_c of enriched conditions such that*

1. *$f(X_p) = {}^\bullet t$ and $f(X_c) = \underline{t}$;*
2. *$X_p \cup X_c$ contains exactly one generating condition for every $c \in ({}^\bullet e \cup \underline{e})$;*
3. *X_p contains generating or reading conditions, X_c generating conditions;*
4. *for all $\rho, \rho' \in X_p \cup X_c$ we have $\rho \parallel \rho'$;*
5. *finally, $H = \bigcup_{\langle c,H'\rangle \in X_p \cup X_c} H'$.*

Proposition 2 allows to identify new possible extensions whenever a prefix is extended with new enriched conditions. Compound conditions are avoided at the price of combining generating and reading conditions as stated in items 2–4 for every possible extension.

Eager approach. The eager approach, instead of attempting to combine generating and reading histories when computing a possible extension, explicitly produces all types of enriched conditions, including compound ones. This means more enriched conditions, but on the other hand less work when computing possible extensions.

Proposition 3. *The pair $\langle e, H\rangle$ with $f(e) = t$ is an enriched event iff there exist sets X_p, X_c of enriched conditions such that*

1. *$f(X_p) = {}^\bullet t$ and $f(X_c) = \underline{t}$;*
2. *$X_p \cup X_c$ contains exactly one enriched condition for every $c \in ({}^\bullet e \cup \underline{e})$;*
3. *X_p contains arbitrary enriched conditions, X_c generating conditions;*
4. *for all $\rho, \rho' \in X_p \cup X_c$ we have $\rho \parallel \rho'$;*
5. *finally, $H = \bigcup_{\langle c,H'\rangle \in X_p \cup X_c} H'$.*

Notice that $|X_p| = |{}^\bullet t|$ in Proposition 3 whereas no such bound exists in Proposition 2. Like the latter, Proposition 3 allows to identify new possible extensions upon addition of new enriched conditions.

5.2 Updating the Concurrency Relation

We face the problem of keeping up to date the concurrency relation on enriched conditions when the unfolding grows by the insertion of new enriched events.

In [2] an approach is proposed, based on the introduction of another binary relation on enriched conditions, called subsumption. Intuitively, $\langle c, H\rangle$ *subsumes* $\langle c', H'\rangle$, written $\langle c, H\rangle \propto \langle c', H'\rangle$, when in the history H there is an event that reads condition c', with history H', and c' is not consumed by H. This means that when taking the enriched condition $\langle c, H\rangle$ we are also implicitly taking $\langle c', H'\rangle$. For instance, in Fig. 1(b), $\langle c_4, \{e_1, e_2\}\rangle$ subsumes $\langle c_3, \{e_1, e_2\}\rangle$. When a new enriched event is inserted in the unfolding, subsumption plays a role in updating the concurrency relation. Assume that the inserted event is $\langle e, H\rangle$ and that it is created using sets X_c, X_p (see Proposition 2 or Proposition 3). Then the enriched conditions generated by $\langle e, H\rangle$ are concurrent with an enriched condition ρ already in the prefix iff $X_p \cup X_c \cup \{\rho\}$ is pairwise concurrent and it satisfies suitable closure properties w.r.t subsumption.

Here we show that for 1-safe nets the result below holds, which allows to update the concurrency relation for a new generating or reading conditions inserted in the unfolding, in a simpler way, without the need of computing subsumption.

Proposition 4. *In Algorithm 1, let $\mathcal{E}$ be the current CEP, where $\langle e, H\rangle$ is the last addition thanks to sets X_c, X_p as per Proposition 2 or Proposition 3. We denote by $Y_p = e^\bullet \times \{H\}$ and $Y_c = \underline{e} \times \{H\}$ the generating and reading conditions created by the addition of $\langle e, H\rangle$. Let $\rho \in Y_p \cup Y_c$, and let $\rho' = \langle c', H'\rangle \in \mathcal{E}$ be any other enriched condition. Then $\rho \parallel \rho'$ iff*

$$\rho' \in Y_p \cup Y_c \;\vee\; (c' \notin {}^\bullet e \;\wedge\; \forall \rho_1 \in X_p \cup X_c : (\rho_1 \parallel \rho') \;\wedge\; \underline{{}^\bullet e} \cap H' \subseteq H)$$

Then the concurrency relation can be transferred to compound conditions on the basis of the result below.

Proposition 5. *Let $\rho = \langle c, H_1 \cup H_2\rangle$ be a compound condition of $\mathcal{E}$, where $\rho_1 = \langle c, H_1\rangle$, $\rho_2 = \langle c, H_2\rangle$ are enriched conditions verifying $\neg(H_1 \# H_2)$. Let $\rho' \in \mathcal{E}$ be any enriched condition. Then $\rho \parallel \rho'$ iff $\rho_1 \parallel \rho' \wedge \rho_2 \parallel \rho'$.*

5.3 Discussion: Lazy vs. Eager Approach

In order to discover possible extensions of the form $\langle e, H\rangle$, both approaches consider combinations of generating and reading histories for conditions $c \in {}^\bullet e$.

Consider Proposition 2. For every possible extension, the lazy approach takes one generating and possibly multiple reading histories for c, all of which must be concurrent. If the events in $\underline{c}$ have many different histories, or $\underline{c}$ is large, then many different combinations need to be checked for concurrency.

The eager approach (Proposition 3) takes exactly one enriched condition of arbitrary type, including compound, for c. Compound histories are a set of concurrent reading histories (Definition 10); thus a compound condition represents pre-computed information needed to identify possible extensions.

We consider two examples where eager beats lazy and vice versa. In Fig. 4 (a), condition c has a sequence of n readers and hence $n+1$ histories $\{e_1, \ldots, e_i\}$, for $i = 0, \ldots, n$. For each history H of c', eager simply combines H with the $n+1$ histories for c, while lazy checks all 2^n subsets of $e_1, \ldots, e_n$ to find these $n+1$ compound histories. If c' has many histories, eager becomes largely superior. Of course, an intelligent strategy may help lazy to avoid exploring all 2^n subsets one by one. However, even with a good strategy, lazy still has to enumerate at least the same combinations as eager; and since the problem of identifying the useful subsets is NP-complete [8], there will always be instances where lazy becomes inefficient, whatever strategy is employed.

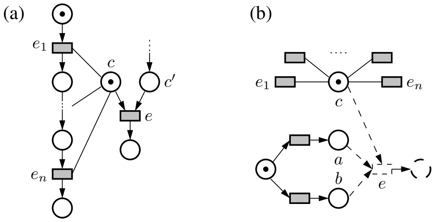


Fig. 4. Good examples for the eager (a) and the lazy (b) approach.

On the other hand, consider Fig. 4 (b). Again, c has n readers, this time yielding 2^n histories. Suppose that $f(c)$ is an input place of some transition t. Now, if t also has $f(a)$ and $f(b)$ in its preset, then no t-labelled event e will ever be generated in the unfolding, and all histories of c are effectively useless. Since those compound conditions also appear in the computation of the concurrency relation, they become a liability in terms of both memory and execution time. The lazy approach does not suffer from this problem here.

Both approaches therefore have their merits, and we implemented them both. We shall report on experiments in Section 7. Concerning Section 5.2, we only retained the new approach, which is clearly better than that of [2].

6 Efficient Prefix Construction

We implemented the procedure from Algorithm 1, using the methods proposed in Section 5. The resulting tool, called Cunf, is publicly available [14]. Cunf expects as input a 1-safe c-net and produces as output a complete unfolding prefix.

Notice that efficient tools exist for the unfolding of Petri nets such as Mole [16] or Punf [10]. While we profited much from the experiences gained from developing Mole, Cunf is not a simple extension of Mole. The issues of asymmetric conflict and histories permeate every aspect of the construction so that we went for a completely new implementation in C, comprising some 4,000 lines of code.

Here, we review some features such as data structures and implementation details, relevant to handling the complications imposed by contextual unfoldings, that helped to produce an efficient tool. Experiments are reported in Section 7.

The history graph. Cunf needs to maintain enriched events and conditions, i.e. tuples $\langle e, H\rangle$ or $\langle c, H\rangle$, where H is a history. We store these in a graph structure, maintained while the enriched prefix $\mathcal{E}$ evolves. Formally, the *history graph* associated with $\mathcal{E}$ is a directed graph $\mathcal{H}_\mathcal{E}$ whose nodes are the enriched events of $\mathcal{E}$, and with edges $\langle e, H\rangle \to \langle e', H'\rangle$ iff $e' \in H$ and $H' = H[\![e']\!]$ and either (i) $(e'^\bullet \cup \underline{e'}) \cap {}^\bullet e \neq \emptyset$ or (ii) $e'^\bullet \cap \underline{e} \neq \emptyset$. Each node $\langle e, H\rangle$ is labelled by e.

Intuitively, $\mathcal{H}_\mathcal{E}$ has an edge between two enriched events $\langle e, H\rangle$ and $\langle e', H'\rangle$ iff some enriched condition $\langle c, H'\rangle$ has been used to construct $\langle e, H\rangle$ (in the sense of Proposition 2 or Proposition 3).

This structure allows Cunf to perform many operations efficiently: every additional enriched event enlarges the graph by just one node plus some edges; common parts of histories are shared. We can easily enumerate the events in $H \in \chi(e)$ by following the edges from node $\langle e, H \rangle$, and $\mathcal{H}_\mathcal{E}$ implicitly represents the relation $\sqsubset$. Given an event e, we can enumerate the histories in $\chi(e)$ by keeping a list of nodes in $\mathcal{H}_\mathcal{E}$ labelled by e. Given a condition c, we can enumerate its generating and reading histories similarly.

Compound conditions are stored in a shared-tree-like structure, where leaves represent reading histories and internal nodes compound histories. An internal node has two children, one of which is a leaf, the other either internal or a leaf. One easily sees that a compound history of c corresponds, w.l.o.g., to a union $H_1 \cup \cdots \cup H_n$ of reading histories. Every internal node represents such a union, and the structure allows sharing if one compound history contains another.

Possible extensions. Cunf behaves similar to Mole or other unfolders in its flow of logic, but its actions are on enriched events and conditions. We start with a prefix containing just $\widehat{m}_0$ and identify the initial possible extensions. As long as the set of possible extensions is non-empty, we choose a "minimal" extension and add it unless it is a cutoff. For "minimal", we use the adequate order $\prec_F$ from [7]. Adding $\langle e, H \rangle$ means adding H to $\chi(e)$, creating e first if necessary. The addition of $\langle e, H \rangle$ will give rise to various types of enriched conditions for whom we compute the concurrency relation (see below). Whenever we add an enriched condition ρ, we attempt to find possible extensions, i.e. sets X_p, X_c matching the conditions in Propositions 2 or 3 such that $X_p \cup X_c$ includes ρ, where, in order to implement condition 4, we use the precomputed binary concurrency relation. Upon identifying a possible extension $\langle e, H \rangle$, we immediately compute its marking, information relevant to deciding $\prec_F$, and certain lists $r(H), s(H)$ during two linear traversals of H. Details on $r(H)$ and $s(H)$ are given below.

Concurrency relation. The relation $\|$ on the enriched conditions $\mathcal{E}$ can be stored and updated whenever new possible extensions are appended to $\mathcal{E}$. We detail now how Propositions 4 and 5 are used to efficiently compute this update.

Let $c(\rho)$ denote the set of enriched conditions ρ' verifying $\rho \,\|\, \rho'$. The relation $\|$ is generally sparse, and Cunf stores $c(\rho)$ as a list. However, for the purpose of the following, $c(\rho)$ could also be a row in a matrix representing $\|$.

For reading and generating conditions ρ (Proposition 4), Cunf initially sets $c(\rho)$ to $Y_p \cup Y_c$. Next, it computes the intersection of $c(\rho')$ for all $\rho' \in X_p \cup X_c$, and filters out those $\langle c', H' \rangle$ for which $\underline{{}^\bullet e} \cap H' \not\subseteq H$ holds. In order to compute this condition without actually traversing H and H', we use the sets $r(H)$ and $s(H)$ computed earlier (see above). These are defined as $r(H) := \{\, e' \in H \mid \underline{e'} \cap \mathsf{Cut}(H) \neq \emptyset \,\}$ and $s(H) := \{\, e' \in H \mid e' \in \underline{{}^\bullet e} \,\}$. Then $\underline{{}^\bullet e} \cap H' \not\subseteq H$ holds iff $\underline{{}^\bullet e} \setminus s(H) \cap r(H') \neq \emptyset$, which can be computed traversing $\underline{{}^\bullet e}$ and $s(H)$ one time, and checking $r(H')$ for every ρ'. Note that, while the other steps have their counterparts in Petri net unfoldings, this step is new and specific to c-nets. However, we find that this implementation keeps the overhead very small.

As for compound conditions ρ built using ρ_1 and ρ_2 (Proposition 5), Cunf computes $c(\rho)$ as the intersection of $c(\rho_1)$ and $c(\rho_2)$.

Certain enriched conditions $\rho = \langle c, H \rangle$ need not to be included in the concurrency relation. It is safe, for instance, to leave $c(\rho)$ empty if ρ is generating and $f(c)^\bullet \cup \underline{f(c)} = \emptyset$, or if H is a cutoff. We can also avoid computing $c(\rho)$ if ρ is reading or compound and $f(c)^\bullet = \emptyset$, even if $\underline{f(c)} \neq \emptyset$.

7 Experiments

In order to experimentally evaluate our tool, we performed a series of experiments. We were interested in the following questions:

- Is the contextual unfolding procedure efficient?
- What is the size of the unfoldings, compared to Petri net unfoldings?
- How do the various approaches (lazy, eager, PR, plain encoding) compare?

Concerning the second and third point, contextual unfoldings may be up to exponentially more succinct than Petri net unfoldings, and we could contrive examples showing arbitrarily large discrepancies. To get more realistic numbers, we took a set of 1-safe nets provided in [4]. This collects nets with various characteristics that allowed to test practically all aspects of our implementation.

For each net N in the set, we first obtained the c-net N' by substituting pairs of arcs (p, t) and (t, p) in N by read arcs. Evidently, the plain encoding of N' is N. Secondly, we obtained the PR-encoding N'' of N'.

We first ran both Mole [16] and Cunf on the nets N and N'', which are ordinary Petri nets without read arcs. Naturally, both tools compute the same result; the object of this exercise was to establish whether Cunf was working reasonably efficient on known examples. Indeed, its running times were always within 70% and 140% of those of Mole, the differences due to minor implementation choices. To abstract from these details, we used Cunf for all further comparisons.

We then used Cunf to produce complete unfoldings of the plain net N, the PR-encoding N'', and of N' using both lazy and eager methods and the order $\prec_F$

Table 1. Experimental results

	Plain		PR		Contextual				Ratios		
Net	Events	t_P	Events	t_R	Av. $\underline{t}$	Events	t_L	t_E	t_E/t_P	t_E/t_R	t_E/t_L
bds_1.sync	12900	0.51	4302	0.26	1.22	1866	0.14	0.14	0.27	0.54	1.00
byzagr4_1b	14724	3.40	8044	5.30	0.92	8044	3.41	2.90	0.85	0.55	0.85
dpd_7.sync	10457	0.88	10457	0.99	0.77	10457	0.92	0.91	1.03	0.92	0.99
elevator_4	16856	2.01	16856	504.77	1.19	16856	1.27	1.26	0.63	>0.01	0.99
ftp_1.sync	83889	76.74	50928	113.38	1.05	50928	34.25	34.21	0.45	0.30	1.00
furnace_4	146606	40.39	100260	43.52	0.85	95335	23.48	18.34	0.45	0.42	0.78
key_4.fsa	67954	2.21	21742	4.30	0.37	4754	2036.66	6.33	2.86	1.47	>0.01
q_1.sync	10722	1.21	10722	2.18	0.90	10722	1.13	1.13	0.93	0.52	1.00
rw_12.sync	98361	3.95	98361	7.64	0.99	98361	4.52	3.10	0.78	0.41	0.69
rw_1w3r	15401	0.38	14982	0.69	0.48	14490	0.45	0.45	1.18	0.65	1.00
rw_2w1r	9241	0.30	9241	8.95	0.76	9241	0.43	0.40	1.33	0.04	0.93

from [7]. Table 1 summarizes the results. For all approaches, we list the number of events in the complete prefix and the running times (in seconds) of the eager approach. For c-nets, we additionally list the running time t_L of the lazy method, since only in proper c-nets this time differs from eager. Notice that the number of events in lazy and eager is the same; moreover, the number of *enriched* events in lazy and eager equals the number of events in PR (compare the discussion in the introduction). The average transition context size is provided for the c-nets, as well as three ratios comparing our running times.

We make the following observations:

– In all examples that we tried, the eager approach was always at least as fast as the lazy approach; an effect similar to the one in Fig. 4 (b) did not happen. On the other hand, in many examples both approaches were nearly equivalent, while in one case (`key_4`) lazy performed badly; see below.
– The eager approach handles all examples gracefully. It is significantly faster than the plain approach in half the cases, and significantly slower in only one case, `key_4`.
– The contextual methods produce smaller unfoldings than the plain approach in 6 out of 11 cases. Interestingly, these are not the same as those on which they run faster. For `elevator_4` and `rw_12.sync`, the same number of events is produced more quickly. Here, the read arcs are arranged in such a way that each event still has only one history; the time saving comes from the fact that the contextual approach produces fewer *conditions* and hence a smaller concurrency relation. For `key_4` and `rw_1w3r`, the contextual methods produce smaller unfoldings but take longer to run; see below for an explanation.
– Comparing with PR, the eager approach is consistently more efficient except for `key_4`. This clear tendency is slightly surprising given that the enriched contextual prefix has essentially the same size as the PR-prefix. We experimentally traced the difference to the enlarged presets of certain transitions in the PR-encoding (see Fig. 2), causing combinatorial overhead and increasing the number of conditions in the concurrency relation. Note that the ratio between number of events in contextual and number of events in PR is the average number of histories per event in the contextual approach.

We briefly discuss `key_4`, which causes problems for the contextual approaches. In this net, there is one place p with a read arc to almost every transition in the net, similar to Fig. 4 (a), with long sequences of readers. As discussed in Section 5.3, the eager approach constructs a number of enriched conditions linear in the length of each sequence whereas the lazy approach breaks down. The plain encoding works fast because every event creates a new copy of p, and every condition is concurrent with only one such copy. It remains to be seen whether the eager approach can be adapted to handle this special case in the same way.

8 Conclusions

We made theoretical and practical contributions to the computation of unfoldings of contextual nets. To our knowledge, Cunf is the first tool that efficiently produces these objects. The availability of a tool that produces contextual unfoldings may trigger new interest in applications of c-nets and the algorithmics of asymmetric event structures in general.

It will be interesting to explore the applications in verification. Unfolding-based techniques need two ingredients: an efficient method for generating them, and efficient methods for analyzing the prefixes. We have provided the first ingredient in this quest. We believe that traditional unfolding-based verification techniques [5] (e.g., SAT-based techniques) can be extended to work with contextual unfoldings and that their succinctness may help to speed up these analyses. We find this topic to be an interesting avenue for future research.

Moreover, despite promising results, the present work will probably not be the last word on the algorithmics of contextual unfoldings; we have some ideas on how to further speed up the process. It would also be interesting to investigate a mix between eager and lazy that tries to get the best of the two worlds. For instance, one could start with the eager approach and switch (selectively for some conditions) to lazy as soon the number of compound conditions exceeds a certain bound. This, and other ideas, remain to be tested.

References

1. Baldan, P., Corradini, A., Montanari, U.: An event structure semantics for P/T contextual nets: Asymmetric event structures. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 63–80. Springer, Heidelberg (1998)
2. Baldan, P., Bruni, A., Corradini, A., König, B., Schwoon, S.: On the computation of McMillan's prefix for contextual nets and graph grammars. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 91–106. Springer, Heidelberg (2010)
3. Baldan, P., Corradini, A., König, B., Schwoon, S.: McMillan's complete prefix for contextual nets. In: Jensen, K., van der Aalst, W.M.P., Billington, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency I. LNCS, vol. 5100, pp. 199–220. Springer, Heidelberg (2008)
4. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE Transactions on Software Engineering 22, 161–180 (1996)
5. Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 37–56. Springer, Heidelberg (2001)
6. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2008)
7. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design 20, 285–310 (2002)
8. Heljanko, K.: Deadlock and Reachability Checking with Finite Complete Prefixes. Licentiate's thesis, Helsinki University of Technology (1999)

9. Janicki, R., Koutny, M.: Invariant semantics of nets with inhibitor arcs. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 317–331. Springer, Heidelberg (1991)
10. Khomenko, V.: Punf, http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/
11. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993)
12. Montanari, U., Rossi, F.: Contextual occurrence nets and concurrent constraint programming. In: Ehrig, H., Schneider, H.-J. (eds.) Dagstuhl Seminar 1993. LNCS, vol. 776. Springer, Heidelberg (1994)
13. Ristori, G.: Modelling Systems with Shared Resources via Petri Nets. Ph.D. thesis, Department of Computer Science, University of Pisa (1994)
14. Rodríguez, C.: Cunf, http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/
15. Rodríguez, C., Schwoon, S., Baldan, P.: Efficient contextual unfolding. Tech. Rep. LSV-11-14, LSV, ENS de Cachan (2011)
16. Schwoon, S.: Mole, http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/
17. Vogler, W., Semenov, A., Yakovlev, A.: Unfolding and finite prefix for nets with read arcs. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 501–516. Springer, Heidelberg (1998)
18. Winkowski, J.: Reachability in contextual nets. Fundamenta Informaticae 51(1-2), 235–250 (2002)

Parameterized Complexity Results for 1-safe Petri Nets

M. Praveen and Kamal Lodaya

The Institute of Mathematical Sciences, Chennai 600113, India

Abstract. We associate a graph with a 1-safe Petri net and study the parameterized complexity of various problems with parameters derived from the graph. With treewidth as the parameter, we give W[1]-hardness results for many problems about 1-safe Petri nets. As a corollary, this proves a conjecture of Downey et. al. about the hardness of some graph pebbling problems. We consider the parameter benefit depth (that is known to be helpful in getting better algorithms for general Petri nets) and again give W[1]-hardness results for various problems on 1-safe Petri nets. We also consider the stronger parameter vertex cover number. Combining the well known automata-theoretic method and a powerful fixed parameter tractability (FPT) result about Integer Linear Programming, we give a FPT algorithm for model checking Monadic Second Order (MSO) formulas on 1-safe Petri nets, with parameters vertex cover number and the size of the formula.

1 Introduction

Petri nets are popular for modelling because they offer a succinct representation of loosely coupled communicating systems. Some powerful techniques are available but the complexity of analysis is high. In his lucid survey [8], Esparza summarizes the situation as follows: almost every interesting analysis question on the behaviour of general Petri nets is EXPSPACE-hard, and almost every interesting analysis question on the behaviour of 1-safe Petri nets is PSPACE-hard. By considering special subclasses of nets slightly better results can be obtained. Esparza points out that T-systems (also called marked graphs) and S-systems (essentially sequential transition systems) are the largest subclasses where polynomial time algorithms are available. We therefore look for a *structural parameter* with respect to which some analysis problems remain tractable.

Parameterized complexity. A brief review will not be out of place here. Let Σ be a finite alphabet in which instances $I \in \Sigma^*$ of a problem $\Pi \subseteq \Sigma^*$ are specified, where Π is the set of YES instances. The complexity of a problem is stated in terms of the amount of resources—space, time—needed by any algorithm solving it, measured as a function of the size $|I|$ of the problem instance. In parameterized complexity, introduced by Downey and Fellows [5], the dependence of resources needed is also measured in terms of a parameter $\kappa(I)$ of the input, which is usually less than the input size $|I|$. A parameterized problem is said

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 358–372, 2011.

to be fixed parameter tractable (FPT) if it can be solved by an algorithm with running time $f(\kappa(I))poly(|I|)$ where f is some computable function and $poly$ is a polynomial. (Similarly, a PARAPSPACE algorithm [10] is one that runs in space $f(\kappa(I))poly(|I|)$.)

For example, consider the problem of checking that all strings accepted by a given finite state automaton satisfy a given Monadic Second Order (MSO) sentence. The size of an instance of this problem is the sum of sizes of the automaton and the MSO sentence. If the size of the MSO sentence is considered as a parameter, then this problem if FPT, by Büchi, Elgot, Trakhtenbrot theorem [2].

There is a parameterized complexity class W[1], lowest in a hierarchy of intractable classes called the W-hierarchy [5] (similar to the polynomial time hierarchy). A parameterized problem complete for W[1] is to decide if there is an accepting computation of at most k steps in a given non-deterministic Turing machine, where the parameter is k [5]. It is widely believed that parameterized problems hard for W[1] are not FPT. To prove that a problem is hard for a parameterized complexity class, we have to give a parameterized reduction from a problem already known to be hard to our problem. A parameterized reduction from (Π, κ) to (Π', κ') is an algorithm A that maps problem instances in (resp. outside) Π to problem instances in (resp. outside) Π'. There must be computable functions f and g and a polynomial p such that the algorithm A on input I terminates in time $f(\kappa(I))p(|I|)$ and $\kappa'(A(I)) \leq g(\kappa(I))$, where $A(I)$ is the problem instance output by A.

Results. Demri, Laroussinie and Schnoebelen considered synchronized transition systems, a form of 1-safe Petri nets [4] and showed that the number of synchronizing components (processes) is not a parameter which makes analysis tractable. Likewise, our first results are negative. All parameters mentioned below are defined in Sect. 2.

- With the pathwidth of the flow graph of the 1-safe Petri net as parameter, reachability, coverability, Computational Tree Logic (CTL) and the complement of Linear Temporal Logic (LTL) model checking problems are all W[1]-hard, even when the size of the formula is a constant. In contrast, for the class of sequential transition systems and formula size as parameter, Büchi's theorem is that model checking for MSO logic is FPT.
- As a corollary, we also prove a conjecture of Downey, Fellows and Stege that the SIGNED DIGRAPH PEBBLING problem [6, section 5] is W[1]-hard when parameterized by treewidth.
- With the benefit depth of the 1-safe Petri net as parameter, reachability, coverability, CTL and the complement of LTL model checking problems are W[1]-hard, even when the size of the formula is a constant.

We are luckier with our third parameter.

- With the vertex cover number of the flow graph and formula size as parameters, MSO model checking is FPT.

Perspective. As can be expected from the negative results, the class of 1-safe Petri nets which are amenable to efficient analysis (i.e., those with small vertex cover) is not too large. But even for this class, a reachability graph construction can be of exponential size, so just an appeal to Büchi's theorem is not sufficient to yield our result.

Roughly speaking, our FPT algorithm works well for systems which have a small "core" (vertex cover), a small number of "interface types" with this core, but any number of component processes using these interface types to interact with the core (see Fig. 5). Thus, we can have a large amount of conflict and concurrency but a limited amount of causality. Recall that S-systems and T-systems have no concurrency and no conflict, respectively. Since all we need from the logic is a procedure which produces an automaton from a formula, we are able to use the most powerful, MSO logic. Our proofs combine the well known automata-theoretic method [2,21,12] with a powerful result about feasibility of Integer Linear Programming (ILP) parameterized by the number of variables [14,13,11].

Related work. Drusinsky and Harel studied nondeterminism, alternation and concurrency in finite automata from a complexity point of view [7]. Their results also hold for 1-bounded Petri nets.

The SIGNED DIGRAPH PEBBLING problem considered by Downey, Fellows and Stege [6] can simulate Petri nets. They showed that with treewidth and the length of the firing sequence as parameters, the reachability problem is FPT. They conjectured that with treewidth alone as parameter, the problem is W[1]-hard.

Fellows *et al* showed that various graph layout problems that are hard with treewidth as parameter (or whose complexity parameterized by treewidth is not known) are FPT when parameterized by vertex cover number [9]. They also used tractability of ILP and extended feasibility to optimization.

2 Preliminaries

2.1 Petri Nets

A Petri net is a 4-tuple $\mathcal{N} = (P, T, Pre, Post)$, P a set of places, T a set of transitions, $Pre : P \times T \rightarrow \{0, 1\}$ (arcs going from places to transitions) and $Post : P \times T \rightarrow \{0, 1\}$ (arcs going from transitions to places) the incidence functions. A place p is an input (output) place of a transition t if $Pre(p, t) = 1$ ($Post(p, t) = 1$) respectively. We use ${}^{\bullet}t$ ($t^{\bullet}$) to denote the set of input (output) places of a transition t. In diagrams, places are shown as circles and transitions as thick bars. Arcs are shown as directed edges between places and transitions.

Given a Petri net $\mathcal{N}$, we associate with it an undirected **flow graph** $G(\mathcal{N}) = (P, E)$ where $(p_1, p_2) \in E$ iff for some transition t, $Pre(p_1, t) + Post(p_1, t) \geq 1$ and $Pre(p_2, t) + Post(p_2, t) \geq 1$. If a place p is both an input and an output place of some transition, the vertex corresponding to p has a self loop in $G(\mathcal{N})$.

A marking $M : P \to \mathbb{N}$ can be thought of as a configuration of the Petri net, with each place p having $M(p)$ tokens. We will only deal with **1-safe** Petri nets in this paper, where the range of markings is restricted to $\{0, 1\}$. Given a Petri net $\mathcal{N}$ with a marking M and a transition t such that for every place p, $M(p) \geq Pre(p, t)$, the transition t is said to be enabled at M and can be fired (denoted $M \overset{t}{\Longrightarrow} M'$) giving $M'(p) = M(p) - Pre(p, t) + Post(p, t)$ for every place p. This is generalized to a firing sequence $M \overset{t_1}{\Longrightarrow} M_1 \overset{t_2}{\Longrightarrow} \cdots \overset{t_r}{\Longrightarrow} M_r$, more briefly $M \xRightarrow{t_1 t_2 \cdots t_r} M_r$. A firing sequence ρ enabled at M_0 is said to be maximal if it is infinite, or if $M_0 \overset{\rho}{\Longrightarrow} M$ and no transition is enabled at M.

Definition 1 (Reachability, coverability). *Given a 1-safe Petri net $\mathcal{N}$ with initial marking M_0 and a target marking $M : P \to \{0, 1\}$, the reachability problem is to decide if there is a firing sequence ρ such that $M_0 \overset{\rho}{\Longrightarrow} M$. The coverability problem is to decide if there is a firing sequence ρ and some marking $M' : P \to \{0, 1\}$ such that $M_0 \overset{\rho}{\Longrightarrow} M'$ and $M'(p) \geq M(p)$ for every place p.*

2.2 Logics

Linear Temporal Logic (LTL) is a formalism in which many properties of transition systems can be specified [8, section 4.1]. We use the syntax of [8], in particular the places P are the atomic formulae. The LTL formulas are interpreted on runs, sequences of markings $\pi = M_0 M_1 \cdots$ from a firing sequence of a 1-safe Petri net. The satisfaction of a LTL formula ϕ at some position j in a run is defined inductively, in particular $\pi, j \models p$ iff $M_j(p) = 1$. Much more expressive is the Monadic Second Order (MSO) logic of Büchi [2], interpreted on a maximal run $M_0 M_1 \cdots$, with $\pi, s \models p(x)$ iff $M_{s(x)}(p) = 1$ under an assignment s to the variables. Boolean operations, first-order and monadic second-order quantifiers are available as usual.

Computational Tree Logic (CTL) is another logic that can be used to specify properties of 1-safe Petri nets. The reader is referred to [8, section 4.2] for details.

Definition 2 (Model checking). *Given a 1-safe Petri net $\mathcal{N}$ with initial marking M_0 and a logical formula ϕ, the model checking problem (for that logic) is to decide if for every maximal firing sequence ρ, the corresponding maximal run π satisfies $\pi, 0 \models \phi$.*

Reachability, coverability and LTL model checking for 1-safe Petri nets are all PSPACE-complete [8]. Habermehl gave an automata-theoretic model checking procedure for Linear Time μ-calculus on general Petri nets [12].

2.3 Parameters

The study of parameterized complexity derived an initial motivation from the study of graph parameters. Many NP-complete problems can be solved in polynomial time on trees and are FPT on graphs that have tree-structured decompositions.

Definition 3 (Tree decomposition, treewidth, pathwidth). *A tree decomposition of a graph $G = (V, E)$ is a pair $(\mathcal{T}, (B_\tau)_{\tau \in nodes(\mathcal{T})})$, where $\mathcal{T}$ is a tree and $(B_\tau)_{\tau \in nodes(\mathcal{T})}$ is a family of subsets of V such that:*

- *For all $v \in V$, the set $\{\tau \in nodes(\mathcal{T}) \mid v \in B_\tau\}$ is nonempty and connected in $\mathcal{T}$.*
- *For every edge $(v_1, v_2) \in E$, there is a $\tau \in nodes(\mathcal{T})$ such that $v_1, v_2 \in B_\tau$.*

*The width of such a decomposition is the number $\max\{|B_\tau| \mid \tau \in nodes(\mathcal{T})\} - 1$. The **treewidth** $tw(G)$ of G is the minimum of the widths of all tree decompositions of G. If the tree $\mathcal{T}$ in the definition of tree decomposition is a path, we get a path decomposition. The **pathwidth** $pw(G)$ of G is the minimum of the widths of all path decompositions of G.*

From the definition, it is clear that pathwidth is at least as large as treewidth and any problem that is W[1]-hard with pathwidth as parameter is also W[1]-hard with treewidth as parameter. A fundamental result by Courcelle [3] shows that graphs of small treewidth are easier to handle algorithmically: checking whether a graph satisfies a MSO sentence is FPT if the graph's treewidth and the MSO sentence's length are parameters. In our context, the state space of a concurrent system can be considered a graph. However, due to the state explosion problem, the state space can be very large. Instead, we impose treewidth restriction on a compact representation of the large state space — a 1-safe Petri net. Note also that we are not model checking the state space itself but only the language of words generated by the Petri net.

Definition 4 (Vertex cover number). *A vertex cover $VC \subseteq V$ of a graph $G = (V, E)$ is a subset of vertices such that for every edge in E, at least one of its vertices is in VC. The **vertex cover number** of G is the size of a smallest vertex cover.*

Definition 5 (Benefit depth [18]). *The set of places $ben(p)$ benefited by a place p is the smallest set of places (including p) such that any output place of any output transition of a place in $ben(p)$ is also in $ben(p)$. The **benefit depth** of a Petri net is defined as $\max_{p \in P}\{|ben(p)|\}$.*

Benefit depth can be thought of as a generalization of the out-degree in directed graphs. For a Petri net, we take vertex covers of its flow graph $G(\mathcal{N})$. Any vertex cover of $G(\mathcal{N})$ should include all vertices that have self loops. It was shown in [18,17] that benefit depth and vertex cover number bring down the complexity of coverability and boundedness in general Petri nets from exponential space-complete [19] to PARAPSPACE.

3 Lower Bounds for 1-safe Petri Nets and Pebbling

3.1 1-safe Petri Nets, Treewidth and Pathwidth

Here we prove W[1]-hardness of reachability in 1-safe Petri nets with the pathwidth of the flow graph as parameter, through a parameterized reduction from the parameterized Partitioned Weighted Satisfiability (p-PW-SAT) problem. The primal graph of a propositional CNF formula has one vertex for each propositional variable, and an edge between two variables iff they occur together in a clause. An instance of p-PW-SAT problem is a triple $(\mathcal{F}, part : \Phi \to \{1, \ldots, k\}, tg : \{1, \ldots, k\} \to \mathbb{N})$, where $\mathcal{F}$ is a propositional CNF formula, *part* partitions the set of propositional variables Φ into k parts and we need to check if there is a satisfying assignment that sets exactly $tg(r)$ variables to $\top$ in each part r. Parameters are k and the pathwidth of the primal graph of $\mathcal{F}$. We showed in an earlier paper that p-PW-SAT is W[1]-hard when parameterized by the number of parts k and the pathwidth of the primal graph [16, Lemma 6.1].

Now we will demonstrate a parameterized reduction from p-PW-SAT to reachability in 1-safe Petri nets, with the pathwidth of the flow graph as parameter. Given an instance of p-PW-SAT, let $q_1, \ldots, q_n$ be the variables used. Construct an optimal path decomposition of the primal graph of the CNF formula in the given p-PW-SAT instance (doing this is FPT [1]). For every clause in the CNF formula, the primal graph contains a clique formed by all variables occurring in that clause. There will be at least one bag in the path decomposition of the primal graph that contains all vertices in this clique [5, Lemma 6.49]. Order the bags of the path decomposition from left to right and call the clause whose clique appears first C_1, the clause whose clique appears second as C_2 and so on. If more than one such such clique appear for the first time in the same bag, order the corresponding clauses arbitrarily. Let $C_1, \ldots, C_m$ be the clauses ordered in this way. We will call this the path decomposition ordering of clauses, and use it to prove that the pathwidth of the flow graph of the constructed 1-safe Petri net is low (Lemma 7). For a partition r between 1 and k, we let $n[r]$ be the number of variables in r. Following are the places of our 1-safe Petri net.

1. For every propositional variable q_i used in the given p-PW-SAT instance, places $q_i, x_i, \overline{x}_i$.
2. For every partition r between 1 and k, places $t{\uparrow}^r, f{\uparrow}^r, tu_r^0, \ldots, tu_r^{tg(r)}$ and $fl_r^0, \ldots, fl_r^{n[r]-tg(r)}$.
3. For each clause C_j, a place C_j. Additional places C_{m+1}, s, g.

The construction of the Petri net is illustrated in the following diagrams. The notation $part(i)$ stands for the partition to which q_i belongs. Intuitively, the truth assignment of q_i is determined by firing t_i or f_i in Fig. 1. The token in $x_i/\overline{x}_i$ is used to check satisfaction of clauses later. The token in $t{\uparrow}^{part(i)}/f{\uparrow}^{part(i)}$ is used to count the number of variables set to $\top/\bot$ in each part, with the part of the net in Fig. 2. For each clause C_j between 1 and m, the part of the net shown in Fig. 3 is constructed. In Fig. 3, it is assumed that $C_j = q_1 \vee \overline{q_2} \vee q_3$. Intuitively, a token can be moved from place C_j to C_{j+1} iff the clause C_j is satisfied by the truth assignment determined by the firings of t_i/f_i for each i between 1 and n. The net in Fig. 4 checks that the target has been met in all partitions.

The initial marking of the constructed net consists of 1 token each in the places $q_1, \ldots, q_n, s, tu_1^0, \ldots, tu_k^0, fl_1^0, \ldots, fl_k^0$ and C_1, with 0 tokens in all other places. The final marking to be reached has a token in the places s and g.

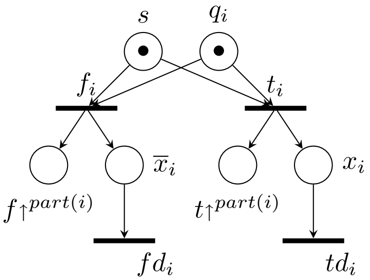


Fig. 1. Part of the net for each variable q_i

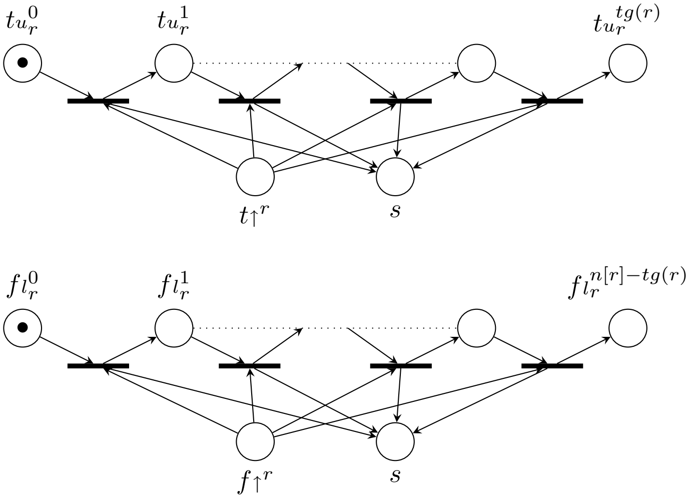


Fig. 2. Part of the net for each part r between 1 and k

Lemma 6 (*[1]). *Given a p-*Pw-Sat *instance, constructing the Petri net described above is* Fpt. *The constructed Petri net is* 1*-safe. The given instance of p-*Pw-Sat *is a* Yes *instance iff in the constructed* 1*-safe net, the required final marking can be reached from the given initial marking.*

It remains to prove that the pathwidth of the flow graph of the constructed 1-safe net is a function of the parameters of the p-Pw-Sat instance.

Lemma 7 (*). *Suppose a given instance of p-*Pw-Sat *has a CNF formula whose primal graph has pathwidth* p_w *and* k *parts. Then, the flow graph of the* 1*-safe net constructed as described above has pathwidth at most* $3p_w + 4k + 7$.

[1] Proofs of lemmas marked with * are omitted due to space constraints. They can be found in the full version available at arxiv, under the same title as this paper.

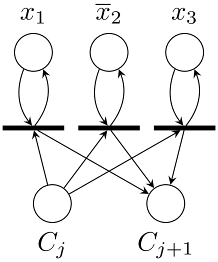


Fig. 3. Part of the net for each clause C_j

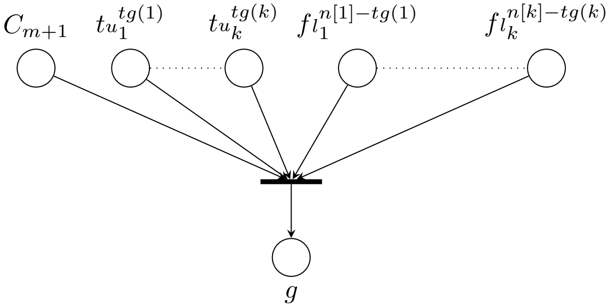


Fig. 4. Part of the net to check that target has been met

In the above reduction, it is enough to check if in the constructed 1-safe net, we can reach a marking that has a token at the place g. This can be expressed as reachability, coverability etc. Hence we get:

Theorem 8. *With the pathwidth (and hence treewidth also) of the flow graph of a* 1*-safe Petri net as parameter, reachability, coverability, CTL model checking and the complement of LTL/MSO model checking (with formulas of constant size) are* W[1]*-hard.*

3.2 Graph Pebbling Problems, Treewidth and Pathwidth

The techniques used in the above lower bound proof can be easily translated to some graph pebbling problems [6]. As conjectured in [6, section 5], we prove that Signed Digraph Pebbling I, parameterized by treewidth is W[1]-hard. An instance of this problem has a bipartite digraph $D = (V, A)$ for which the vertex set V is partitioned $V = Red \cup Blue$, and also the arc set A is partitioned into two partitions $A = A^+ \cup A^-$. The problem is to reach the *finish state* where there are pebbles on all the red vertices, starting from a *start state* where there are no pebbles on any of the red vertices, by a series of moves of the following form:

- If b is a blue vertex such that for all s such that $(s, b) \in A^+$, s is pebbled, and for all s such that $(s, b) \in A^-$, s is not pebbled (in which case we say that b is *enabled*), then the set of vertices s such that $(b, s) \in A^+$ are reset by making them all pebbled, and the set of all vertices s such that $(b, s) \in A^-$ are reset by making them all unpebbled.

Corollary 9 (*). *Parameterized by pathwidth (and hence by treewidth also),* Signed Digraph Pebbling *is* W[1]*-hard.*

The proof is by a parameterized reduction from p-Pw-Sat to reachability in 1-safe Petri nets as in the last sub-section, and another reduction from reachability in 1-safe Petri nets to Signed Digraph Pebbling.

3.3 1-safe Petri Nets and Benefit Depth

Here we show that the parameter benefit depth is not helpful for 1-safe Petri nets, by showing W[1]-hardness using a parameterized reduction from the constraint satisfaction problem (Csp).

Theorem 10 (*). *With benefit depth as the parameter in* 1*-safe Petri nets, reachability, coverability, CTL model checking and the complement of the LTL/MSO model checking problems, even with formulas of constant size, are* W[1]*-hard.*

4 Vertex Cover and Model Checking 1-safe Petri Nets

In this section, we will show that with the vertex cover number of the flow graph of the given 1-safe Petri net and the size of the given LTL/MSO formula as parameters, checking whether the given net is a model of the given formula is Fpt. With vertex cover number as the only parameter, we cannot hope to get this kind of tractability:

Proposition 11 (*). *Model checking LTL (and hence MSO) formulas on* 1*-safe Petri nets whose flow graph has constant vertex cover number is* Co-Np*-hard.*

Since a run of a 1-safe net $\mathcal{N}$ with set of places P is a sequence of subsets of P, we can think of such sequences as strings over the alphabet $\mathscr{P}(P)$ (the power set of P). It is known [2,21] that with any LTL or MSO formula ϕ, we can associate a finite state automaton $\mathcal{A}_\phi$ over the alphabet $\mathscr{P}(P)$ accepting the set of finite strings which are its models, as well as a finite state Büchi automaton $\mathcal{B}_\phi$ accepting the set of infinite string models.

Figure 5 shows the schematic of a simple manufacturing system modelled as a 1-safe Petri net. Starting from p_1, it picks up one unit of a raw material α and goes to p_2, then picks up raw material β, then γ. Transition t_1 does some processing and then the system starts from p_1 again. Suppose we want to make sure that whenever the system picks up a unit of raw material β, it is processed immediately. In other words, whenever the system stops at a marking where no transitions are enabled, there should not be a token in p_3. This can be checked

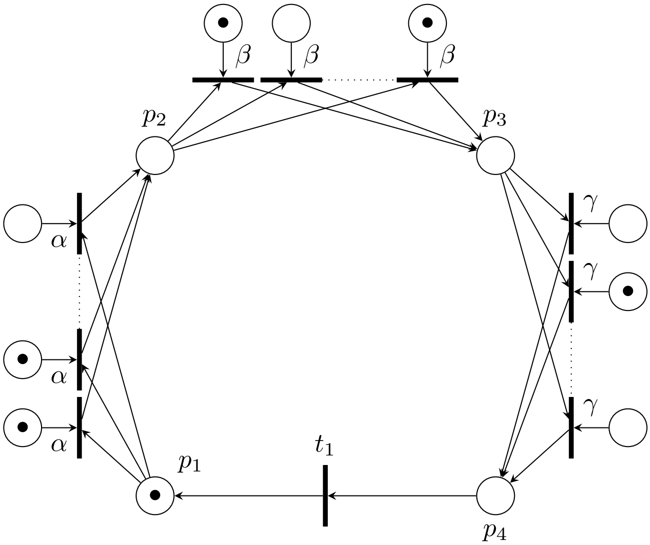


Fig. 5. An example of a system with small vertex cover

by verifying that all finite maximal runs satisfy the formula $\forall x((\forall y \quad y \leq x) \Rightarrow \neg p_3(x))$. The satisfaction of this formula depends only on the number of units of raw materials α, β and γ at the beginning, i.e., the number of tokens at the initial marking. The naive approach of constructing the whole reachability graph results in an exponentially large state space, due to the different orders in which the raw materials of each type can be drawn. If we want to reason about only the central system (which is the vertex cover $\{p_1, p_2, p_3, p_4\}$ in the above system), it turns out that we can ignore the order and express the requirements on the numbers by integer linear constraints.

Suppose VC is a vertex cover for $G(\mathcal{N})$. We use the fact that if $v_1, v_2 \notin VC$ are two vertices not in VC that have the same set of neighbours, v_1 and v_2 have similar properties. This has been used to obtain FPT algorithms for many hard problems (e.g. [9]). The following definitions formalize this.

Definition 12. *Let VC be a vertex cover of $G(\mathcal{N})$. The **(VC-) neighbourhood** of a transition t is the ordered pair $({}^\bullet t \cap VC, t^\bullet \cap VC)$. We denote by l the number of different VC-neighbourhoods.*

Definition 13. *Suppose $\mathcal{N}$ is a Petri net with l neighbourhoods for vertex cover VC, and $p \notin VC$. The **(VC-) interface** $int[p]$ of p is defined as the function $int[p] : \{1, \ldots, l\} \to \mathscr{P}(\{-1, 1\})$, where for every j between 1 and l and every $w \in \{1, -1\}$, there is a transition t_j of VC-neighbourhood j such that $w = -Pre(p, t_j) + Post(p, t_j)$ iff $w \in int[p](j)$.*

In the net in Fig. 5 with $VC = \{p_1, p_2, p_3, p_4\}$, all transitions labelled α have the same VC-neighbourhood and all the corresponding places have the same VC-interface. Since there can be $2k$ arcs between a transition and places in VC if

$|VC| = k$, there can be at most 2^{2k} different VC-neighbourhoods of transitions. There are at most $4^{2^{2k}}$ VC-interfaces. The set of interfaces is denoted by *Int*.

Proposition 14. *Let $\mathcal{N}$ be a 1-safe net with VC being a vertex cover of $G(\mathcal{N})$. Let $p_1, p_2, \ldots, p_i$ be places not in the vertex cover, all with the same interface. Let M be some marking reachable from the initial marking of $\mathcal{N}$. If $M(p_j) = 1$ for some j between 1 and i, then M does not enable any transition that adds tokens to any of the places $p_1, \ldots, p_i$.*

Proof. Suppose there is a transition t enabled at M that adds a token to $p_{j'}$ for some j' between 1 and i. Then there is a transition t' with the same neighbourhood as t (and hence enabled at M too) that can add a token to p_j. Firing t' from M will create 2 tokens at p_j, contradicting the fact that $\mathcal{N}$ is 1-safe. □

If the initial marking has tokens in many places with the same interface, then no transition can add tokens to any of those places until all the tokens in all those places are removed. Once all tokens are removed, one of the places can receive one token after which, no place can receive tokens until this one is removed. All these places have the same interface. Thus, a set of places with the same interface can be thought of as an initial storehouse of tokens, after depleting which it can be thought of as a single place. However, a formula in our logic can reason about individual places, so we still need to keep track of individual places that occur in the formula.

Proposition 15 (*). *Let $\mathcal{N}$ be a 1-safe net and ϕ be an MSO formula. Let $P_\phi \subseteq P$ be the subset of places that occur in ϕ. Let $\pi = M_0 M_1 \cdots$ and $\pi' = M'_0 M'_1 \cdots$ be two finite or infinite runs of $\mathcal{N}$ such that for all positions j of π and for all $p \in P_\phi$, $M_j(p) = M'_j(p)$. For any assignment s, we have $\pi, s \models \phi$ iff $\pi', s \models \phi$.*

Let $\mathcal{N}$ be a 1-safe net such that $G(\mathcal{N})$ has a vertex cover VC of size k. Suppose ϕ is a formula and we have to check if $\mathcal{N}$ satisfies ϕ. For each interface I, let $P_I \subseteq P$ be the places not in VC with interface I. If $P_I \setminus P_\phi \neq \emptyset$ (i.e., if there are places in P_I that are not in ϕ), designate one of the places in $P_I \setminus P_\phi$ as p_I. Define the set of special places $\mathcal{S} = VC \cup P_\phi \cup \{p_I \in P_I \setminus P_\phi \mid I \text{ is an interface and } P_I \setminus P_\phi \neq \emptyset\}$. Note that $|\mathcal{S}| \leq k + |\phi| + 4^{2^{2k}}$. Since this number is a function of the parameters of the input instance, we will treat it as a parameter.

We need a structure that keeps track of changes in places belonging to $\mathcal{S}$, avoiding a construction involving all reachable markings. This can be done by a finite state machine whose states are subsets of $\mathcal{S}$. Transitions of the Petri net that only affect places in $\mathcal{S}$ can be simulated by the finite state machine with its usual transitions. To simulate transitions of the net that affect places outside $\mathcal{S}$, we need to impose some conditions on the number of times transitions of the finite state machine can be used. The following definition formalizes this. For a marking M of $\mathcal{N}$, let $M \lceil \mathcal{S} = \{p \in \mathcal{S} \mid M(p) = 1\}$.

Definition 16. *Given a 1-safe net $\mathcal{N}$ with initial marking M_0 and $\mathcal{S}$ defined from ϕ as above, the* ***edge constrained automaton*** *$\mathcal{A}_\mathcal{N} = (Q_\mathcal{N}, \Sigma, \delta_\mathcal{N}, u, F_\mathcal{N})$ is a structure defined as follows. $Q_\mathcal{N} = \mathscr{P}(\mathcal{S})$ and $\Sigma = Int \cup \{\perp\}$ (recall that*

Int is the set of interfaces in $\mathcal{N}$). The transition relation $\delta \subseteq Q_{\mathcal{N}} \times \Sigma \times Q_{\mathcal{N}}$ is such that for all $P_1, P_2 \subseteq \mathcal{S}$ and $I \in Int \cup \{\bot\}$, $(P_1, I, P_2) \in \delta$ iff there are markings M_1, M_2 and a transition t of $\mathcal{N}$ such that

- *$M_1 \lceil \mathcal{S} = P_1$, $M_2 \lceil \mathcal{S} = P_2$ and $M_1 \xRightarrow{t} M_2$,*
- *t removes a token from a place $p \in P_I \setminus \mathcal{S}$ of interface I if $I \in Int$ and*
- *t does not have any of its input or output places in $P \setminus \mathcal{S}$ if $I = \bot$.*

*The **edge constraint** $u : Int \to \mathbb{N}$ is given by $u(I) = |\{p \in P_I \setminus \mathcal{S} \mid M_0(p) = 1\}|$. A subset $P_1 \subseteq \mathcal{S}$ is in $F_{\mathcal{N}}$ iff for every marking M with $M \lceil \mathcal{S} = P_1$, the only transitions enabled at M remove tokens from some place not in $\mathcal{S}$.*

Intuitively, the edge constraint u defines an upper bound on the number of times those transitions can be used that reduce tokens from places not in $\mathcal{S}$.

Definition 17. *Let $\mathcal{A}_{\mathcal{N}}$ be an edge constrained automaton as in Def. 16 and let $\pi = P_0 P_1 \cdots$ be a finite or infinite word over $\mathscr{P}(\mathcal{S})$. Then π is a valid run of $\mathcal{A}_{\mathcal{N}}$ iff for every position $j \geq 1$ of π, we can associate an element $I_j \in \Sigma$ such that*

- *for every position $j \geq 1$ of π, $(P_{j-1}, I_j, P_j) \in \delta$ and*
- *for every $I \in Int$, $|\{j \geq 1 \mid I_j = I\}| \leq u(I)$.*
- *if π is finite and P_j is the last element of π, then $P_j \in F_{\mathcal{N}}$ and for every interface $I \in Int$ and marking $M_j \lceil \mathcal{S} = P_j$ enabling some transition that removes tokens from some place in $P_I \setminus \mathcal{S}$, $|\{j \geq 1 \mid I_j = I\}| = u(I)$.*

Next we have a run construction lemma.

Lemma 18 (*). *Let $\mathcal{N}$ be a 1-safe net with initial marking M_0, ϕ be a formula and $\mathcal{A}_{\mathcal{N}}$ be as in Def. 16. For every infinite (maximal finite) run $\pi = M_0 M_1 \cdots$ of $\mathcal{N}$, there exists an infinite (finite) run $\pi' = M'_0 M'_1 \cdots$ such that the word $(M'_0 \lceil \mathcal{S})(M'_1 \lceil \mathcal{S}) \cdots$ is a valid run of $\mathcal{A}_{\mathcal{N}}$ and for every position j of π, $M'_j \lceil P_\phi = M_j \lceil P_\phi$. If an infinite (finite) word $\pi = P_0 P_1 \cdots$ over $\mathscr{P}(\mathcal{S})$ is a valid run of $\mathcal{A}_{\mathcal{N}}$ and $P_0 = M_0 \lceil \mathcal{S}$, then there is an infinite (finite maximal) run $M_0 M_1 \cdots$ of $\mathcal{N}$ such that $M_j \lceil \mathcal{S} = P_j$ for all positions j of π.*

Lemma 18 implies that in order to check if $\mathcal{N}$ is a model of the formula ϕ, it is enough to check that all valid runs of $\mathcal{A}_{\mathcal{N}}$ satisfy ϕ. This can be done by checking that no finite valid run of $\mathcal{A}_{\mathcal{N}}$ is accepted by $\mathcal{A}_{\neg\phi}$ and no infinite valid run of $\mathcal{A}_{\mathcal{N}}$ is accepted by $\mathcal{B}_{\neg\phi}$. As usual, this needs a product construction. Automata $\mathcal{A}_{\neg\phi}$ and $\mathcal{B}_{\neg\phi}$ run on the alphabet $\mathscr{P}(P_\phi)$. Let $Q_{\mathcal{A}}$ and $Q_{\mathcal{B}}$ be the set of states of $\mathcal{A}_{\neg\phi}$ and $\mathcal{B}_{\neg\phi}$ respectively. Then, $\mathcal{A}_{\neg\phi} = (Q_{\mathcal{A}}, \mathscr{P}(P_\phi), \delta_{\mathcal{A}}, Q_{0\mathcal{A}}, F_{\mathcal{A}})$ and $\mathcal{B}_{\neg\phi} = (Q_{\mathcal{B}}, \mathscr{P}(P_\phi), \delta_{\mathcal{B}}, Q_{0\mathcal{B}}, F_{\mathcal{B}})$.

Definition 19. *$\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi} = (Q_{\mathcal{N}} \times Q_{\mathcal{A}}, \Sigma, \delta^{\mathcal{N}}_{\mathcal{A}}, \{M_0 \lceil \mathcal{S}\} \times Q_{0\mathcal{A}}, F_{\mathcal{N}} \times F_{\mathcal{A}}, u)$, $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi} = (Q_{\mathcal{N}} \times Q_{\mathcal{B}}, \Sigma, \delta^{\mathcal{N}}_{\mathcal{B}}, \{M_0 \lceil \mathcal{S}\} \times Q_{0\mathcal{B}}, Q_{\mathcal{N}} \times F_{\mathcal{B}}, u)$ where*

$$((q_1, q_2), I, (q'_1, q'_2)) \in \delta^{\mathcal{N}}_{\mathcal{A}} \text{ iff } (q_1, I, q'_1) \in \delta_{\mathcal{N}} \text{ and } (q_2, q_1 \cap P_\phi, q'_2) \in \delta_{\mathcal{A}}$$
$$((q_1, q_2), I, (q'_1, q'_2)) \in \delta^{\mathcal{N}}_{\mathcal{B}} \text{ iff } (q_1, I, q'_1) \in \delta_{\mathcal{N}} \text{ and } (q_2, q_1 \cap P_\phi, q'_2) \in \delta_{\mathcal{B}}$$

An accepting path of $\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi}$ is a sequence $(q_0, q'_0) I_1 (q_1, q'_1) \cdots I_r (q_r, q'_r)$ which is $\delta^{\mathcal{N}}_{\mathcal{A}}$-respecting:

- $(q_0, q'_0), (q_1, q'_1), \ldots, (q_r, q'_r) \in Q_{\mathcal{N}} \times Q_{\mathcal{A}}$,
- *the word* $I_1 \cdots I_r \in \Sigma^*$ *witnesses the validity of the run* $q_0 q_1 \cdots q_r$ *in* $\mathcal{A}_{\mathcal{N}}$ *(as in Def. 17) and*
- *the word* $(q_0 \cap P_\phi) \cdots (q_r \cap P_\phi)$ *is accepted by* $\mathcal{A}_{\neg\phi}$ *through the run* $q'_0 q'_1 \cdots q'_r q'_F$ *for some* $q'_F \in F_{\mathcal{A}}$ *with* $(q'_r, q_r \cap P_\phi, q'_F) \in \delta_{\mathcal{A}}$.

An accepting path of $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi}$ *is defined similarly.*

Proposition 20 (*). *A* 1*-safe net* $\mathcal{N}$ *with initial marking* M_0 *is a model of a formula* ϕ *iff there is no accepting path in* $\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi}$ *and* $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi}$.

To efficiently check the existence of accepting paths in $\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi}$ and $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi}$, it is convenient to look at them as graphs, possibly with self loops and parallel edges. Let the set of states be the set of vertices of the graph and each transition (q, I_j, q') be an I_j-labelled edge leaving q and entering q'. If there is a path μ in the graph from q to q', the number of times an edge e occurs in μ is denoted by $\mu(e)$. If $s \notin \{q, q'\}$ is some node occurring in μ, then the number of edges of μ entering s is equal to the number of edges of μ leaving s. These conditions can be expressed as integer linear constraints.

$$\begin{aligned} \sum_{e \text{ leaves } q} \mu(e) - \sum_{e \text{ enters } q} \mu(e) &= 1 \\ \sum_{e \text{ enters } q'} \mu(e) - \sum_{e \text{ leaves } q'} \mu(e) &= 1 \\ s \notin \{q, q'\} : \sum_{e \text{ enters } s} \mu(e) &= \sum_{e \text{ leaves } s} \mu(e) \end{aligned} \tag{1}$$

Lemma 21 (Theorem 2.1, [20]). *In a directed graph* $G = (V, E)$ *(possibly with self loops and parallel edges), let* $\mu : E \to \mathbb{N}$ *be a function such that the underlying undirected graph induced by edges* e *such that* $\mu(e) > 0$ *is connected. Then, there is a path from* q *to* q' *with each edge* e *occurring* $\mu(e)$ *times iff* μ *satisfies the constraints* (1) *above.*

If the beginning and the end of a path are same (i.e., if $q = q'$), small modifications of (1) and Lemma 21 are required. Finally we can prove our desired theorem.

Theorem 22. *Let* $\mathcal{N}$ *be a* 1*-safe net with initial marking* M_0 *and* ϕ *be a MSO formula. Parameterized by the vertex cover number of* $G(\mathcal{N})$ *and the size of* ϕ, *checking whether* $\mathcal{N}$ *is a model of* ϕ *is* FPT.

Proof. By Prop. 20, it is enough to check that there is no accepting paths in $\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi}$ and $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi}$. To check the existence of accepting paths in $\mathcal{A}_{\mathcal{N}} \times \mathcal{B}_{\neg\phi}$, we have to check if from some initial state in $\{M_0\lceil\mathcal{S}\} \times Q_{0\mathcal{B}}$, we can reach some vertex in a maximal strongly connected component induced by $\perp$-labelled edges, which contains some states from $Q_{\mathcal{N}} \times F_{\mathcal{B}}$. For every such initial state q and a vertex q' in such a strongly connected component, check the feasibility of (1) along with the following constraint for each interface I:

$$\sum_{e \text{ is } I- \text{ labelled}} \mu(e) \leq u(I) \tag{2}$$

To check the existence of accepting paths in $\mathcal{A}_{\mathcal{N}} \times \mathcal{A}_{\neg\phi}$, check the feasibility of (1) and (2) for every state q in $\{M_0 \lceil \mathcal{S}\} \times Q_{0\mathcal{A}}$ and every state (P_1, q'') in $F_{\mathcal{N}} \times Q_{\mathcal{A}}$ with some $q_F \in F_{\mathcal{A}}$ such that $(q'', P_1 \cap P_\phi, q_F) \in \delta_{\mathcal{A}}$. If some marking M with $M\lceil \mathcal{S} = P_1$ enables some transition removing a token from some place with interface I, then for each such interface, add the following constraint:

$$\sum_{e \text{ is } I- \text{ labelled}} \mu(e) = u(I) \tag{3}$$

The variables in the above Ilp instances are $\mu(e)$ for each edge e. The number of variables in each Ilp instance is bounded by some function of the parameters. As Ilp is Fpt when parameterized by the number of variables [13,14,11], the result follows. □

The dependence of the running time of the above algorithm on formula size is non-elementary if the formula is MSO [15]. The dependence reduces to single exponential in case of LTL formulas [21]. The dependence on vertex cover number is dominated by the running time of Ilp, which is singly exponential in the number of its variables. The number of variables in turn depends on the number of VC-interfaces (Def. 13). In the worst case, this can be triply exponential but a given 1-safe Petri net need not have all possible VC-interfaces.

5 Conclusion

The main idea behind the Fpt upper bound for MSO/LTL model checking is the fact that the problem can be reduced to graph reachability and hence to Ilp. It remains to be seen if such techniques or others can be applied for branching time logics such as CTL.

We have some negative results with pathwidth and benefit depth as parameters and a positive result with vertex cover number as parameter. We think it is a challenging problem to identify other parameters associated with 1-safe Petri nets for which standard problems in the concurrency literature are Fpt. Another direction for further work, suggested by a referee, is to check if the upper bound can be extended to other classes of Petri nets such as communication-free nets.

The results of Sect. 3 proves hardness for the lowest level of the W-hierarchy. It remains to be seen if the lower bounds could be made tighter. The parameterized classes ParaNp and Xp include the whole W-hierarchy. Lower bounds or upper bounds corresponding to these classes would be interesting.

Acknowledgements. We thank the anonymous Concur referees for providing detailed comments that helped in improving the presentation.

References

1. Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. J. Alg. 21(2), 358–402 (1996)
2. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Logic, Methodology, Philosophy and Science, pp. 1–11. Stanford Univ. Press, Stanford (1962)
3. Courcelle, B.: The monadic second-order logic of graphs I: Recognizable sets of finite graphs. Information and Computation 85, 12–75 (1990)
4. Demri, S., Laroussinie, F., Schnoebelen, P.: A parametric analysis of the state-explosion problem in model checking. J. Comput. Syst. Sci. 72(4), 547–575 (2006)
5. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
6. Downey, R.G., Fellows, M.R., Stege, U.: Parameterized complexity: A framework for systematically confronting computational intractability. In: Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future. DIMACS, vol. 49, pp. 49–100 (1999)
7. Drusinsky, D., Harel, D.: On the power of bounded concurrency I: Finite automata. J. Assoc. Comput. Mach. 41(3), 517–539 (1994)
8. Esparza, J.: Decidability and complexity of Petri net problems — An introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
9. Fellows, M.R., Lokshtanov, D., Misra, N., Rosamond, F.A., Saurabh, S.: Graph layout problems parameterized by vertex cover. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 294–305. Springer, Heidelberg (2008)
10. Flum, J., Grohe, M.: Describing parameterized complexity classes. Information and Computation 187(2), 291–319 (2003)
11. Frank, A., Tardos, E.: An application of simultaneous diophantine approximation in combinatorial optimization. Combinatorica 7(1), 49–65 (1987)
12. Habermehl, P.: On the complexity of the linear-time μ-calculus for Petri-nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 102–116. Springer, Heidelberg (1997)
13. Kannan, R.: Minkowski's convex body theorem and integer programming. Math. Oper. Res. 12(3), 415–440 (1987)
14. Lenstra, H.W.: Integer programming with a fixed number of variables. Math. Oper. Res. 8, 538–548 (1983)
15. Meyer, A.R.: Weak monadic second order theory of succesor is not elementary-recursive. In: Proc. Logic Colloquium. Lecture Notes in Mathematics, vol. 453, pp. 132–154 (1975)
16. Praveen, M.: Does treewidth help in modal satisfiability? (extended abstract). In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 580–591. Springer, Heidelberg (2010), Full version http://arxiv.org/abs/1006.2461
17. Praveen, M.: Small vertex cover makes petri net coverability and boundedness easier. In: Raman, V., Saurabh, S. (eds.) IPEC 2010. LNCS, vol. 6478, pp. 216–227. Springer, Heidelberg (2010)
18. Praveen, M., Lodaya, K.: Modelchecking counting properties of 1-safe nets with buffers in parapspace. In: FSTTCS. LIPIcs, vol. 4, pp. 347–358 (2009)
19. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoret. Comp. Sci. 6, 223–231 (1978)
20. Reutenauer, C.: The mathematics of Petri nets (1990); translated by Craig, I.
21. Vardi, M.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)

The Decidability of the Reachability Problem for CCS$^{!\star}$

Chaodong He

BASICS, Department of Computer Science
Shanghai Jiao Tong University, Shanghai 200240, China
MOE-MS Key Laboratory for Intelligent Computing and Intelligent Systems

Abstract. CCS$^!$ is a variant of CCS in which infinite behaviors are defined by the replication operator. We show that the reachability problem for CCS$^!$ is decidable by a reduction to the same problem for Petri Nets.

1 Introduction

Process calculi provide languages in which the structure of syntactic terms represents the structure of processes and the operational semantics represents steps of computation or interaction. Among various process calculi, CCS remains a standard representative.

Several variants of CCS have appeared in literature. The relative expressive power of these variants is investigated in [5,6,7,11,10]. It seems that there are two aspects which affect the expressive power significantly. One is the mechanism adopted for extending finite processes in order to express infinite behaviors [5]. The other is the capability of producing and manipulating local channels [14]. According to this fact, five major variants of CCS are given in Fig. 1. In the diagram an arrow ‘ $\longrightarrow$ ’ indicates the sub-language relationship. The five variants of CCS are further divided into three classes. The first class contains CCS$^{\mathrm{Pdef}}$, in which infinite behaviors are specified by *parametric definition* [21,11] (or equivalently *dynamic-scoping recursion* [5]). This mechanism offers a certain degree of name-passing capability such that process copies can be nested at arbitrary depth, which results in the Turing completeness of CCS$^{\mathrm{Pdef}}$ [11,5,23]. The second class contains CCS$^\mu$ and CCS$^!$, in which the infinite behaviors are specified by (static-scoping) recursion and replication, respectively. These two subcalculi have the power of producing new local channels but not have the power of passing names around. They are not Turing complete because they are not expressive enough to define ‘counter’ [10]. The third class contains CCS$^\mu_\bullet$ and CCS$^!_\bullet$ in which the local names are always static. In these two variants, localization operators can only act as the outermost constructors to ensure that no local channels can be produced during the evolution of processes.

Given a variant of CCS, a legitimate question is whether a certain process property is decidable. This paper explores the reachability problem for the CCS

* The work is supported by NSFC (60873034, 61033002).

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 373–388, 2011.

$$\begin{tikzcd}
\mathrm{CCS}^{\mu}_{\bullet} \arrow[r] & \mathrm{CCS}^{\mu} \arrow[r] & \mathrm{CCS}^{\mathrm{Pdef}} \\
\mathrm{CCS}^{!}_{\bullet} \arrow[u] \arrow[r] & \mathrm{CCS}^{!} \arrow[u] &
\end{tikzcd}$$

Fig. 1. CCS Variants

variants. This problem asks whether a given source process can evolve to a given target process within finitely many steps of computation or interaction.

The reachability problem for CCS$^{\mathrm{Pdef}}$ is undecidable due to Turing completeness. The reachability problem for CCS$^{\mu}_{\bullet}$ and CCS$^{!}_{\bullet}$ is decidable, which is obtained from the fact that CCS$^{\mu}_{\bullet}$ and CCS$^{!}_{\bullet}$ can be embedded into Labeled Petri Nets (LPN for short) with simple modification of U.Goltz's encoding [12] by C.He *et al.* [14], and from a prominent discovery for Petri Nets, by Ernst W. Mayr [18], that *the reachability problem for Petri Nets is decidable.*

The contribution of this paper is to show that the reachability problem for CCS$^{!}$ is decidable. The result is proved by a reduction to the reachability problem for Labeled Petri Nets.

At first we notice that the way of deciding reachability problem for CCS$^{!}_{\bullet}$ (or CCS$^{\mu}_{\bullet}$) does not work for CCS$^{!}$. The standard encodings from CCS to LPN [23,12] share the guideline that the sequential processes are represented by places and their parallel occurrences are counted by tokens. These encodings do nothing with local names. This is why the encoding from CCS$^{!}_{\bullet}$ (or CCS$^{\mu}_{\bullet}$) to LPN relies heavily on static local names. Intuitively, in CCS$^{!}$ there are processes, for instance of the form $!(\ldots \parallel (a)\,!P \parallel (b)\,!Q \parallel \ldots)$, in which nested local names form a tree. The standard encodings may cause tokens for different component confused. There are other evidences which suggest that no reasonable encoding from CCS$^{!}$ to LPN exists. In [6], N.Busi *et al.* show that CCS$^{!}$ can model Minsky Machine non-deterministically, which confirms that CCS$^{!}$ is 'nearly' Turing complete, while such a result seems not to hold for LPN, which is likely to be 'far from' Turing complete. A more convincing fact is that, for CCS$^{!}$ strong bisimilarity with a given regular process is undecidable [14], while this problem is decidable for LPN [15]. Even though CCS can indeed be encoded in terms of LPN with infinite places or with inhibitor arcs [8], it is helpless in deciding reachability problem for CCS, for these accessories make the corresponding reachability problem for Petri Nets undecidable.

The key observation yielding the decidability result is the following property of CCS$^{!}$: When a replicated subprocess becomes 'active' (i.e. not guarded by a prefix), this subprocess remains active evermore. Based on this observation, if there exists an evolution path from source to target, the number of active occurrences of a certain replicated subprocess in any intermediate states is 'bounded' by that number in the target. Within these 'bounds', predefined in the target process, a Labeled Petri Net is constructed recursively. These 'bounds' serve as the requisite copies of subnets for representing every replicated subprocess. The

$$\textsf{Choice}\ \frac{}{\sum_{i=1}^{n} \lambda_i.P_i \xrightarrow{\lambda_i} P_i} \qquad \textsf{Composition}\ \frac{P \xrightarrow{\lambda} P'}{P \parallel Q \xrightarrow{\lambda} P' \parallel Q} \quad \frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\overline{l}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$\textsf{Localization}\ \frac{P \xrightarrow{\lambda} P' \quad a \text{ not appear in } \lambda}{(a)P \xrightarrow{\lambda} (a)P'} \qquad \textsf{Replication}\ \frac{P \xrightarrow{\lambda} P'}{!\,P \xrightarrow{\lambda} !\,P \parallel P'}$$

Fig. 2. Semantics of CCS$^!$

constructed net is strong enough to produce the evolution path in which the numbers of active replicated subprocesses are 'bounded'.

The rest of the paper is organized as follows. Section 2 lays down the preliminaries. Section 3 expounds the main idea and formal definitions. Section 4 describes the construction of Labeled Petri Net. Section 5 states the whole algorithm. Section 6 gives concluding remarks. Some technical details and proofs are omitted. See [13] for complete coverage.

2 Preliminaries

2.1 The Calculus

To describe the interactions between systems, we need channel names. The set of the names $\mathcal{N}$ is ranged over by $a, b, c, \ldots$, and the set of the names and the conames $\mathcal{N} \cup \overline{\mathcal{N}}$ is ranged over by $l, \ldots$. The set of the action labels $\mathcal{A} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$ is ranged over by λ.

The set $\mathcal{P}_{\text{CCS}^!}$ of CCS$^!$ processes, ranged over by $P, Q, \ldots$, is generated inductively by the grammar

$$P \quad ::= \quad \mathbf{0} \quad | \quad \sum_{i=1}^{n} \lambda_i.P_i \quad | \quad P \parallel P' \quad | \quad (a)P \quad | \quad !\,P$$

A name a appeared in process $(a)P$ is *local*. A name is *global* if it is not local. We write $\mathsf{gn}(P)$ for the set of global names of P.

The semantics of CCS$^!$ is given by *labeled transition system* $(\mathcal{P}_{\text{CCS}^!}, \mathcal{A}, \longrightarrow)$, where the elements of $\mathcal{P}_{\text{CCS}^!}$ are often referred to as *states*. The relation $\longrightarrow\ \subseteq \mathcal{P}_{\text{CCS}^!} \times \mathcal{A} \times \mathcal{P}_{\text{CCS}^!}$ is the *transition* relation. The membership $(P, \lambda, P') \in\ \longrightarrow$ is always indicated by $P \xrightarrow{\lambda} P'$. The relation $\longrightarrow$ is generated inductively by the rules defined in Fig. 2. The symmetric rules are omitted.

Standard notations and conventions in process calculi will be used throughout the paper. The inactive process $\mathbf{0}$ is omitted in most occasions. For instance $a.b.\mathbf{0}$ is abbreviated to $a.b$. A finite sequence (or set) of names $a_1, \ldots, a_n$ is often abbreviated to $\widetilde{a}$. The guarded choice term $\sum_{i=1}^{n} \lambda_i.P_i$ is usually written as $\lambda_1.P_1 + \cdots + \lambda_n.P_n$. Processes are not distinguished syntactically up to the commutative monoid generated by '+' and '$\parallel$'. We shall write $\prod_{i=1}^{n} P_i$ for $P_1 \parallel \cdots \parallel P_n$. The notation '$\equiv$' is used to indicate syntactic congruence. The set of the *derivatives* of a process P, denoted by $\mathtt{Drv}(P)$, is the set of the processes P' such that $P \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} P'$ for some $n \geq 0$ and $\lambda_1, \ldots, \lambda_n \in \mathcal{A}$.

2.2 The Petri Nets

Let $\mathbb{N}$ be the set of natural numbers. A *Petri Net* is a tuple $N = (S, T, F, \mathbf{m}_{\text{Init}})$ and a *Labeled Petri Net* is a tuple $N = (S, T, F, L, \mathbf{m}_{\text{init}})$, where S and T are finite disjoint sets of *places* and *transitions* respectively, $F : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ is a *flow function* and $L : T \rightarrow \mathcal{A}$ is a *labeling*. $\mathbf{m}_{\text{init}}$ is the *initial marking*, where a *marking* $\mathbf{m}$ is a function $S \rightarrow \mathbb{N}$ assigning the number of *tokens* to each place.

A transition $t \in T$ is *enabled* at a marking $\mathbf{m}$, denoted by $\mathbf{m} \overset{t}{\rightsquigarrow}$, if $\mathbf{m}(\mathsf{s}) \geq F(\mathsf{s}, t)$ for every $\mathsf{s} \in S$. A transition t enabled at $\mathbf{m}$ may *fire* yielding the marking $\mathbf{m}'$, denoted by $\mathbf{m} \overset{t}{\rightsquigarrow} \mathbf{m}'$, where $\mathbf{m}'(\mathsf{s}) = \mathbf{m}(\mathsf{s}) - F(\mathsf{s}, t) + F(t, \mathsf{s})$ for all $\mathsf{s} \in S$. For each $\lambda \in \mathcal{A}$, we write $\mathbf{m} \overset{\lambda}{\rightsquigarrow}$, respectively $\mathbf{m} \overset{\lambda}{\rightsquigarrow} \mathbf{m}'$ to mean that $\mathbf{m} \overset{t}{\rightsquigarrow}$, respectively $\mathbf{m} \overset{t}{\rightsquigarrow} \mathbf{m}'$ for some t with $L(t) = \lambda$. A labeled transition system $(\mathcal{M}, \mathcal{A}, \rightsquigarrow)$ can be generated from a Labeled Petri Net N, where $\mathcal{M}$ is the set of all markings of N.

In the remainder of this paper, Labeled Petri Nets are treated more algebraically. Let $S = \{\mathsf{s}_i\}_{i=1}^{|S|}$ be the set of places of N. A marking $\mathbf{m} = \{m_i\}_{i=1}^{|S|}$ is viewed as a vector with dimension $|S|$, or equivalently a multiset over S. A transition t will be specified by a label λ and two vectors $\mathbf{v} = \{v_i\}_{i=1}^{|S|}$ and $\mathbf{w} = \{w_j\}_{j=1}^{|S|}$. The flow function F for t is defined by $F(\mathsf{s}_i, t) = v_i$ and $F(t, \mathsf{s}_j) = w_j$ for every $i, j \in \{1, \ldots, |S|\}$. We will use *labeled transition rules* of the form

$$\mathsf{s}_{i_1}^{v_{i_1}} \mathsf{s}_{i_2}^{v_{i_2}} \ldots \mathsf{s}_{i_p}^{v_{i_p}} \overset{\lambda}{\rightsquigarrow} \mathsf{s}_{j_1}^{w_{i_1}} \mathsf{s}_{j_2}^{w_{i_2}} \ldots \mathsf{s}_{j_q}^{w_{i_q}}$$

to indicate a transition t with label λ, vectors $\{v_i\}_{i=1}^{|S|}$ and $\{w_j\}_{j=1}^{|S|}$, where v_i and w_j is zero if $i \notin \{i_1, \ldots, i_p\}$ or $j \notin \{j_1, \ldots, j_q\}$. Whenever $\mathbf{m} = \mathbf{r} + \mathbf{v}$, $\mathbf{m}$ can be replaced by $\mathbf{m}' = \mathbf{r} + \mathbf{w}$. The empty multiset is denoted by ϵ. Thus an Labeled Petri Net N is specified by $(S, \mathcal{A}, \rightsquigarrow, \mathbf{m}_{\text{init}})$, where $\rightsquigarrow \in \mathcal{M} \times \mathcal{A} \times \mathcal{M}$ is a set of labeled transition rules.

2.3 Reachability Problem

The formalization of the reachability problem depends on when two processes are regarded syntactically equal. Let $\bumpeq$ be an equivalence relation on $\mathcal{P}_{\text{CCS}}$. We have the following parameterized reachability problem:

Problem:	Reachability(CCS, $\bumpeq$)
Instance:	Two CCS processes P and Q.
Question:	Does there exist Q' such that $Q \bumpeq Q' \in \mathtt{Drv}(P)$?

The relation $\bumpeq$ serves as the syntactical equality. The question here is how shall we choose $\bumpeq$? The syntactic nature requires that $\bumpeq$ must be decidable, and it also should validate that following *harmonic* property:

If $P_1' \bumpeq P_1 \xrightarrow{\lambda} P_2$, then there exists P_2' such that $P_1' \xrightarrow{\lambda} P_2' \bumpeq P_2$.

The harmonic property is exactly the strong bisimulation property [20,22]. We require that the inference of $P_1' \xrightarrow{\lambda} P_2'$ can be effectively constructed.

The strong bisimilarity itself is not a good candidate of $\bumpeq$. Using the construction of Busi *et al.* [6], one can show Reachability(CCS$^!$, $\sim$) undecidable. On the other hand, Reachability(CCS$^!$, $\bumpeq$) could be decided in an obvious way if we not imposing requisite equations on $\bumpeq$. For example if P is not equated to $P \parallel \mathbf{0}$, Reachability(CCS$^!$, $\bumpeq$) can be decided with the intuition that, during the evolution, the number of unguarded composition operators (not appear under guarded choice) cannot decrease, and every infinite evolution path must eventually using the rule for replication, which increases this number strictly.

Definition 1. *The* strong structural congruence*, $\equiv$, is the smallest congruence relation generated by the following laws:*

$$P \parallel Q \equiv Q \parallel P \qquad (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \qquad P \parallel \mathbf{0} \equiv P$$

The weak structural congruence*, $\dot{\equiv}$, is the smallest congruence generated by the laws for $\equiv$ together with the following laws:*

$$(a_1)(a_2)P \dot{\equiv} (a_2)(a_1)P \qquad (a)(P \parallel Q) \dot{\equiv} P \parallel (a)Q \ \ \textit{if } a \notin \mathsf{gn}(P) \qquad (a)\mathbf{0} \dot{\equiv} \mathbf{0}$$

In this paper, we treat $\bumpeq$ to be $\equiv$ or $\dot{\equiv}$. The reachability problem for CCS$^!$ always refers to Reachability(CCS$^!$, $\equiv$) or Reachability(CCS$^!$,$\dot{\equiv}$).

3 Main Idea

3.1 Informal Description

We have mentioned in Section 1 that the reachability problem for CCS$^!_\bullet$ can be decided by a structural encoding to LPN. Each process $P \in \mathcal{P}_{\mathrm{CCS}^!_\bullet}$ can be assumed to be in the form $(\tilde{a})\prod_{i\in I} P_i$ in which $\widetilde{a}$ are all the local names of P, and every P_i, called *concurrent component*, is localization free and is not a composition. The encoding depends on the fact that local names are static, and the number of the possible concurrent components of all derivatives of P is finite. The encoding works for CCS$^\mu_\bullet$ as well [14]. However, this encoding is not sound if local names can appear underneath replicators. In this situation, the local names newly produced may be capable of preventing certain interactions between components, which is unknown before running the process.

The basic idea of our deciding algorithm for the reachability problem of CCS$^!$ is motivated by the following observation. Consider the following two processes

$$\begin{aligned} P &\stackrel{\text{def}}{=} \,!c.!((a.!P_1 + b.!P_2) \parallel !P_3) \\ Q &\stackrel{\text{def}}{=} \,!c.!((a.!P_1 + b.!P_2) \parallel !P_3) \parallel \\ &\quad\ !((a.!P_1 + b.!P_2) \parallel !P_3) \parallel !P_1 \parallel !P_3 \parallel \\ &\quad\ !((a.!P_1 + b.!P_2) \parallel !P_3) \parallel !P_2 \parallel P_2' \parallel !P_3 \end{aligned}$$

where $P_2 \xrightarrow{\lambda} P_2'$. It is easy to check that $Q \equiv \in \mathtt{Drv}(P)$. We find in the definition of Q that the number of active (i.e. not guarded by a prefix) occurrences of the replicated process $!\,P_1$, $!\,P_2$, and $!\,P_3$ are *one*, *one*, and *two*, respectively. The key observation is that, once a replicated subprocess becomes active, this subprocess remains always active. Based on this observation, any intermediate states in any evolution paths from P to Q have at most one active occurrence of $!\,P_1$, one active occurrence of $!\,P_2$, and two active occurrences of $!\,P_3$.

The main difficulty in translating $\text{CCS}^!$ to LPN is that the interplay of localization and replication have the power of creating unbounded number of different components (modulo $\equiv$ or $\dot{\equiv}$). Because each component is usually interpreted as a place, it seems that infinite places are needed. The above observation enlightens us that some computation paths can be excluded, and in the remaining computation paths, at most finite number of different components can emerge. With this insight, a Labeled Petri Net is constructed recursively, in which every component is translated into a certain copy of some sub-net.

3.2 Formal Definitions

In order to formalize the above intuition, we need some auxiliary notations.

Let $P \in \mathcal{P}_{\text{CCS}^!}$, and $P' \in \mathtt{Drv}(P)$. A component P_i of P' may be created during the evolution by applying rule $\mathsf{Replication}$. In this situation, it is helpful for us to know the place P_i comes from. This suggests the following.

Definition 2. *Let $\mathcal{T}$ be the set of tags, ranged over by* $\mathsf{u}, \mathsf{v}, \ldots$. *The processes of* $\text{CCS}^!$ *with tags is generated inductively by the grammar*

$$P \quad ::= \qquad \mathbf{0} \quad | \quad \sum_{i=1}^{n} \lambda_i.P_i \quad | \quad P \parallel P' \quad | \quad (a)P \quad | \quad !P \quad | \quad \langle P \rangle_{\mathsf{v}}$$

The process $\langle P \rangle_{\mathsf{v}}$ is in tagged form. *The semantic rule for tag is*

$$\mathsf{Tag} \quad \frac{P \xrightarrow{\lambda} P'}{\langle P \rangle_{\mathsf{v}} \xrightarrow{\lambda} \langle P' \rangle_{\mathsf{v}}}$$

The (weak) structural congruence is generated to tagged processes by letting $\langle \mathbf{0} \rangle_{\mathsf{v}} \equiv \mathbf{0}$. By congruence we have $\langle (a)\mathbf{0} \rangle_{\mathsf{v}} \dot{\equiv} \mathbf{0}$ and $(a)\langle \mathbf{0} \rangle_{\mathsf{v}} \dot{\equiv} \mathbf{0}$. Note that we do not have equations such as $\langle P \parallel Q \rangle_{\mathsf{v}} \equiv P \parallel \langle Q \rangle_{\mathsf{v}}$ in general.

Definition 3. *A process P of* $\text{CCS}^!$ *is* standard, *if the subprocesses of P in tagged form are exactly the ones just underneath replication operators, and every tag in P is different.*

Neither $!\,\langle a + b.\,!\,c.d \rangle_{\mathsf{v}}$ nor $!\,\langle a \rangle_{\mathsf{v}} \parallel !\,\langle a \rangle_{\mathsf{v}}$ is standard, because in the former $c.d$ is not in tagged form, and in the latter the same tag v appears twice. These processes can be rectified to $!\,\langle a + b.\,!\,\langle c.d \rangle_{\mathsf{v}_2} \rangle_{\mathsf{v}_1}$ and $!\,\langle a \rangle_{\mathsf{v}_1} \parallel !\,\langle a \rangle_{\mathsf{v}_2}$, which is now standard.

The next notation is used to specify standard processes or their derivatives.

Definition 4. *The set $\mathcal{C}$ of* context*, ranged over by* $\mathbf{C}$*, is generated inductively by the grammar*

$$\mathbf{C} \quad ::= \quad \underline{\mathsf{v}} \quad | \quad \mathbf{0} \quad | \quad \sum_{i=1}^{n} \lambda_i.\mathbf{C}_i \quad | \quad \mathbf{C} \parallel \mathbf{C}' \quad | \quad (a)\mathbf{C} \quad | \quad \langle \mathbf{C} \rangle_{\mathsf{v}}$$

We use $\mathbf{C}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ to indicate a context with tags exactly $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$. $\mathbf{C}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ are often abbreviated as $\mathbf{C}$ if no ambiguity arises. We use $\mathbf{C}[P_1, P_2, \ldots, P_n]$ to indicate the process by replacing each $\underline{\mathsf{v}_i}$ with $!\,\langle P_i \rangle_{\mathsf{v}_i}$. We use $\mathbf{C}\{R/\underline{\mathsf{v}_i}\}$ to indicate the process by replacing the hole $\underline{\mathsf{v}_i}$ with the process R.

A standard process P can be represented as:

$$\mathbf{C}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$$

in which every v_i occurs only once and every P_{v_i} is standard for $i = 1, \ldots, n$. We will call $\mathbf{C}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ the *characteristic context* of P. The derivatives of P can also be represented in this way where the same v_i may occur several times. For example, let P be the process $b.!\,\langle c.(a)(!\,\langle a \rangle_{\mathsf{v}_1} \parallel d.!\,\langle \overline{a} \rangle_{\mathsf{v}_2})\rangle_{\mathsf{u}}$. P can be represented as $\mathbf{C}[P_{\mathsf{u}}]$ with $\mathbf{C}[\underline{\mathsf{u}}] = b.\underline{\mathsf{u}}$, and P_{u} is represented as $\mathbf{C}_{\mathsf{u}}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}]$ with $\mathbf{C}_{\mathsf{u}}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}] = c.(a)(\underline{\mathsf{v}_1} \parallel d.\underline{\mathsf{v}_2})$, $P_{\mathsf{v}_1} \equiv a$ and $P_{\mathsf{v}_2} \equiv \overline{a}$. When

$$P \xrightarrow{b}\xrightarrow{c} P' \equiv !\,\langle c.(a)(!\,\langle a \rangle_{\mathsf{v}_1} \parallel d.!\,\langle \overline{a} \rangle_{\mathsf{v}_2})\rangle_{\mathsf{u}} \parallel \langle (a)(!\,\langle a \rangle_{\mathsf{v}_1} \parallel d.!\,\langle \overline{a} \rangle_{\mathsf{v}_2})\rangle_{\mathsf{u}},$$

we can represent P' as $\mathbf{C}'[P_{\mathsf{u}}, P_{\mathsf{v}_1}, P_{\mathsf{v}_2}]$ with $\mathbf{C}'[\underline{\mathsf{u}}, \underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}] = \underline{\mathsf{u}} \parallel \langle (a)(\underline{\mathsf{v}_1} \parallel d.\underline{\mathsf{v}_2})\rangle_{\mathsf{u}}$.

Definition 5. *Let* $\mathbf{C}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ *be a context with tags* $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n$*. We say that* v_i *is* active *in* $\mathbf{C}$*, denoted by* $\mathbf{C} \rhd \mathsf{v}_i$*, if* v_i *is not under guarded choice. The set of active tags in* $\mathbf{C}$ *is denoted by* $\mathtt{Act}(\mathbf{C}')$*.*

Let P be standard and $P' \in \mathtt{Drv}(P)$. When $P' = \mathbf{C}'[P_{\mathsf{v}_1}, \ldots, P_{\mathsf{v}_n}]$ and $\mathbf{C}' \rhd \mathsf{v}_i$, we say that $!\,\langle P_{\mathsf{v}_i} \rangle_{\mathsf{v}_i}$ is active *in P', denoted by $P' \rhd !\,\langle P_{\mathsf{v}_i} \rangle_{\mathsf{v}_i}$.*

The number of active occurrences of v_i in $\mathbf{C}'$ (or $!\,\langle P_{\mathsf{v}_i} \rangle_{\mathsf{v}_i}$ in P') is denoted by $\mathbf{num}(\mathsf{v}_i, \mathbf{C}')$ (or $\mathbf{num}(!\,\langle P_{\mathsf{v}_i} \rangle_{\mathsf{v}_i}, P')$).

For example, $!\,\langle b.c \rangle_{\mathsf{v}}$ is active in $(b)(!\,\langle b.c \rangle_{\mathsf{v}} \parallel d)$, while $!\,\langle b.c \rangle_{\mathsf{v}}$ is not active in $(b)d.(!\,\langle b.c \rangle_{\mathsf{v}} \parallel e)$. Note also that it is meaningless to talk about whether $!\,\langle b.c \rangle_{\mathsf{v}}$ is active in $!\,\langle !\,\langle b.c \rangle_{\mathsf{v}} \rangle_{\mathsf{u}}$, for v does not occur in the characteristic context.

Now we are in a position to formalize the framework of the decision procedure for Reachability(CCS$^!$, $\doteq$) where $\doteq$ can be either $\equiv$ or $\dot{\equiv}$.

Given two processes $P, Q \in \mathcal{P}_{\mathrm{CCS}^!}$. We want to decide whether $Q \doteq\, \in \mathtt{Drv}(P)$. At first, P can be converted to $\widehat{P}$ which is standard by adding different tags to every subprocess of P just under replicator. After that, we need to convert Q to $\widehat{Q}$ and confirm that $Q \doteq\, \in \mathtt{Drv}(P)$ if and only if $\widehat{Q} \doteq\, \in \mathtt{Drv}(\widehat{P})$. This is done by rewriting Q to some Q' via the structural congruence laws and then adding tags on Q' by guessing! By using some rewriting strategy (eliminating redundant $\mathbf{0}$ when possible), we insure that this can be done algorithmically.

For example, consider the process P and Q in Section 3.1. P can be converted to $\widehat{P} \equiv !\,\langle c.!\,\langle (a.!\,\langle P_{\mathsf{u}_1} \rangle_{\mathsf{u}_1} + b.!\,\langle P_{\mathsf{u}_2} \rangle_{\mathsf{u}_2}) \parallel !\,\langle P_{\mathsf{u}_3} \rangle_{\mathsf{u}_3} \rangle_{\mathsf{v}_2} \rangle_{\mathsf{v}_1}$. By guessing the position of tags, Q can be converted to $\widehat{Q} \equiv \mathbf{C}'[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, P_{\mathsf{u}_1}, P_{\mathsf{u}_2}, P_{\mathsf{u}_3}]$ with

$$\mathbf{C}'[\underline{\mathsf{v}_1},\underline{\mathsf{v}_2},\underline{\mathsf{u}_1},\underline{\mathsf{u}_2},\underline{\mathsf{u}_3}] = \underline{\mathsf{v}_1} \parallel \langle \underline{\mathsf{v}_2} \parallel \langle \underline{\mathsf{u}_1} \parallel \underline{\mathsf{u}_3} \rangle_{\mathsf{v}_2} \rangle_{\mathsf{v}_1} \parallel \langle \underline{\mathsf{v}_2} \parallel \langle \underline{\mathsf{u}_2} \parallel \langle P'_{\mathsf{u}_2} \rangle_{\mathsf{u}_2} \parallel \underline{\mathsf{u}_3} \rangle_{\mathsf{v}_2} \rangle_{\mathsf{v}_1}$$

where $P_{\mathsf{u}_2} \xrightarrow{\lambda} P'_{\mathsf{u}_2}$. Here we use the representation by characteristic context. Note also that active occurrences of $\mathsf{u}_1, \mathsf{u}_2, \mathsf{u}_3$ in $\mathbf{C}'$ are *one*, *one*, *two*, respectively.

The key assertion here is that through the transitions from $\widehat{P}$ to $\widehat{Q}$, $\widehat{P}$ can never transit to the following $\mathbf{C}''[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, P_{\mathsf{u}_1}, P_{\mathsf{u}_3}]$ with

$$\mathbf{C}''[\underline{\mathsf{v}_1},\underline{\mathsf{v}_2},\underline{\mathsf{u}_1},\underline{\mathsf{u}_3}] = \underline{\mathsf{v}_1} \parallel \langle \underline{\mathsf{v}_2} \parallel \langle \underline{\mathsf{u}_1} \parallel \underline{\mathsf{u}_3} \rangle_{\mathsf{v}_2} \parallel \langle \underline{\mathsf{u}_1} \parallel \underline{\mathsf{u}_3} \rangle_{\mathsf{v}_2} \rangle_{\mathsf{v}_1}$$

in which active occurrences of u_1 is more than once.

In the following, we always assume that P is standard. The intuition described in Section 3.1 is summarized as the next two lemmas.

Lemma 1 (Monotonicity Lemma). *Let P be standard. If $P' \in \mathtt{Drv}(P)$ and $P'' \in \mathtt{Drv}(P')$, then $\mathbf{num}(!\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}, P') \leq \mathbf{num}(!\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}, P'')$.*

Lemma 2 (Bounding Lemma). *Let P be standard, and $Q \in \mathtt{Drv}(P)$. Suppose that $\mathbf{num}(!\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}, Q) = k$, then for every processes R which is an intermediate process during the evolution from P to Q, $\mathbf{num}(!\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}, R) \leq k$.*

Based on Lemma 1 and Lemma 2, a Labeled Petri Net can be constructed. Within this net, we can solve the reachability problem for CCS$^!$.

Theorem 1. Reachability(CCS$^!$, $\equiv$) *and* Reachability(CCS$^!$,$\dot{\equiv}$) *are both decidable.*

Before ending this section, we define the *component* of a context, which will be used in the formal construction of the Labeled Petri Net in Section 4.

Definition 6. *Let P be in standard form and $\mathbf{C}$ be the characteristic context of P. The* component *of $\mathbf{C}$, denoted by $\mathtt{Comp}(\mathbf{C})$, is defined inductively:*

$$\begin{aligned}
\mathtt{Comp}(\underline{\mathsf{v}}) &\overset{\text{def}}{=} \{\underline{\mathsf{v}}\} \\
\mathtt{Comp}(\sum_{i=1}^{n} \lambda_i.\mathbf{C}_i) &\overset{\text{def}}{=} \{\sum_{i=1}^{n} \lambda_i.\mathbf{C}_i\} \cup \bigcup_{i=1}^{n} \mathtt{Comp}(\mathbf{C}_i) \\
\mathtt{Comp}(\mathbf{C}_1 \parallel \mathbf{C}_2) &\overset{\text{def}}{=} \{\mathbf{C}'_1 \parallel \mathbf{C}'_2 \mid \mathbf{C}'_1 \in \mathtt{Comp}(\mathbf{C}_1), \mathbf{C}'_2 \in \mathtt{Comp}(\mathbf{C}_2)\} \\
\mathtt{Comp}((a)\mathbf{C}) &\overset{\text{def}}{=} \{(a)\mathbf{C}' \mid \mathbf{C}' \in \mathtt{Comp}(\mathbf{C})\} \\
\mathtt{Comp}(\langle \mathbf{C} \rangle_{\mathsf{v}}) &\overset{\text{def}}{=} \{\langle \mathbf{C}' \rangle_{\mathsf{v}} \mid \mathbf{C}' \in \mathtt{Comp}(\mathbf{C})\}
\end{aligned}$$

Note that $\mathtt{Comp}(\mathbf{C})$ is very similar to that of $\mathtt{Sub}(P)$ in [6,7]. In $\mathtt{Sub}(P)$, the localization operators are completely neglected. When localization is taken into account, the defining equation in the case $\mathbf{C}_1 \parallel \mathbf{C}_2$ also need to be modified. Since we will treat the replication operator inductively, there is no need to define $\mathtt{Comp}(!P)$. Intuitively, $\mathtt{Comp}(\mathbf{C})$ is understood as the finite set of possible subprocesses in $\mathtt{Drv}(P)$ which is not produced from replication.

1. If $\mathbf{C}_{\mathsf{v}} \xrightarrow{\lambda} \mathbf{C}'$, we have rule
$$[\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}\ [\mathbf{C}']_{\mathsf{v}} \tag{B1}$$
2. If $\mathbf{C} \xrightarrow{\lambda} \mathbf{C}'$, we have rule
$$[\mathbf{C}]_{\mathsf{v}} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\mathbf{C}']_{\mathsf{v}} \tag{B2}$$
3. If $\mathbf{m}_1 \overset{l}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}_1'$ and $\mathbf{m}_2 \overset{\overline{l}}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}_2'$ are rules defined by (B1) and (B2), we have rule
$$\mathbf{m}_1\ \mathbf{m}_2 \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}_1'\ \mathbf{m}_2' \tag{B3}$$

Fig. 3. Rules for the Base Step

4 The Construction of Petri Nets

This section is devoted to the technical part of the construction of Label Petri Net $N = (S, \mathcal{A}_\diamond, \rightsquigarrow, \mathbf{m}_{\mathrm{init}})$ from the source process P and the target process Q. $\mathcal{A}_\diamond$ is the set $\mathcal{A} \cup \{\diamond\}$ in which $\diamond$ is the auxiliary label appearing in N and not acting as a transition label of P. The construction is inductive. For every subprocess of form $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}}$, a Labeled Petri Net $N_{\mathsf{v}} = (S_{\mathsf{v}}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}}, \mathbf{m}_{\mathsf{v},\mathrm{init}})$ is constructed. Usually, a place of N_{v} corresponds to a context of P_{v}. Thus a partial function $\mathtt{Ctxt} : S_{\mathsf{v}} \rightharpoonup \mathcal{C}$ is maintained during the construction. By means of $\mathtt{Ctxt}$ and the inductive procedure, we can compute $\mathtt{Proc}(N_{\mathsf{v}})$, a subprocess of P_{v}, from a given place or marking of N_{v}. The function $\mathtt{Ctxt}$ will be used to translate Q to a marking of N in Section 5. The formal definition of $\mathtt{Proc}$ is omitted since it can be obtained from $\mathtt{Ctxt}$ and the inductive construction.

The construction of N includes three steps: *base step*, *induction step*, and *final step*.

4.1 Base Step

In the base step, we treat the process in the form $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}}$ with P_{v} replication free. In this special case, P_{v} is the same as $\mathbf{C}_{\mathsf{v}}$, a context containing no tags. The net for $!\,\langle \mathbf{C}_{\mathsf{v}}\rangle_{\mathsf{v}}$ is $N_{\mathsf{v}} = (S_{\mathsf{v}}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}}, \mathbf{m}_{\mathsf{v},\mathrm{init}})$ defined as follows. The place set

$$S_{\mathsf{v}} = [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \cup \{\, [\mathbf{C}]_{\mathsf{v}} \mid \mathbf{C} \in \mathtt{Comp}(\mathbf{C}_{\mathsf{v}})\}.$$

and $\mathtt{Ctxt}([\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}) = !\,\langle \mathbf{C}_{\mathsf{v}}\rangle_{\mathsf{v}}$, and $\mathtt{Ctxt}([\mathbf{C}]_{\mathsf{v}}) = \langle \mathbf{C}\rangle_{\mathsf{v}}$ if $\mathbf{C} \in \mathtt{Comp}(\mathbf{C}_{\mathsf{v}})$. The initial marking

$$\mathbf{m}_{\mathsf{v},\mathrm{init}} = [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}.$$

The labeled transition rules $\rightsquigarrow_{\mathsf{v}}$ is defined by the rules in Fig. 3.

Rule (B1) deals with the case that $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}} \xrightarrow{\lambda} !\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}} \parallel \langle P_{\mathsf{v}}'\rangle_{\mathsf{v}}$ caused by $\langle P_{\mathsf{v}}\rangle_{\mathsf{v}} \xrightarrow{\lambda} \langle P'\rangle_{\mathsf{v}}$. The derivatives of $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}}$ must be of the form $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}} \parallel \langle P_1\rangle_{\mathsf{v}} \parallel \ldots \parallel \langle P_m\rangle_{\mathsf{v}}$, in which every $P_r (1 \leq r \leq m)$ can be represented as some $\mathbf{C} \in \mathtt{Comp}(\mathbf{C}_{\mathsf{v}})$. Rule (B2) deals with the behaviors of $\langle P_r\rangle_{\mathsf{v}}$'s. Rule (B3) deals with the interaction between two $\langle P_r\rangle_{\mathsf{v}}$'s, or between $!\,\langle P_{\mathsf{v}}\rangle_{\mathsf{v}}$ and $\langle P_r\rangle_{\mathsf{v}}$.

4.2 Induction Step

In the induction step, we treat the process in the form $!\langle P_\mathsf{v}\rangle_\mathsf{v}$. The characteristic context of P_v is $\mathbf{C}_\mathsf{v}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$, and $P_\mathsf{v} = \mathbf{C}_\mathsf{v}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$. Notice also that the base step is the special case of the induction step.

By hypothesis, the Labeled Petri Nets $N_{\mathsf{v}_i} = (S_{\mathsf{v}_i}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}_i}, \mathbf{m}_{\mathsf{v}_i,\mathrm{init}})$ has already been constructed for every $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}$. From these nets, we shall define $N_\mathsf{v} = (S_\mathsf{v}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_\mathsf{v}, \mathbf{m}_{\mathsf{v},\mathrm{init}})$, the net for $!\langle P_\mathsf{v}\rangle_\mathsf{v}$.

Let k_i be $\mathbf{num}(!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}, Q)$. According to the Bounding Lemma, if $Q \in \mathtt{Drv}(P)$, $\mathbf{num}(!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}, R) \leq k_i$ for any intermediate processes R. Because of that, we need k_i disjoint copies of N_{v_i}, named $N_{\mathsf{v}_i,j_i} = (S_{\mathsf{v}_i,j_i}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}_i,j_i}, \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i})$ for $j_i = 1, 2, \ldots, k_i$. During the evolution of $!\langle P_\mathsf{v}\rangle_\mathsf{v}$, whenever a certain $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}$ come to be active, one of the copies of N_{v_i} is triggered, and the corresponding index j_i is consumed. In this way, N_v is constructed, which can partially mimic the process $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}$.

Let $\mathbf{C}'_\mathsf{v}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ be a derivative of $\mathbf{C}_\mathsf{v}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ in which a certain $\underline{\mathsf{v}_i}$ becomes active. Then one of the k_i copies of N_{v_i} is designated to this active tag. In this case we need to record which copy is designated to a given tag. Thus we need a function $\mathbf{lk}$ which maps every v_i to a number $\mathbf{lk}(\mathsf{v}_i) \in \{\perp\} \cup \{1, 2, \ldots, k_i\}$. If $\mathbf{lk}(\mathsf{v}_i) = j_i$, then the copy with index j_i is designated to tag $\underline{\mathsf{v}_i}$ in the context. If $\mathbf{lk}(\mathsf{v}_i) = \perp$, it means no copy of N_{v_i} has been designated to the tag $\underline{\mathsf{v}_i}$. We will use $\mathbf{lk}_\perp$ to indicate the function with $\mathbf{lk}_\perp(\mathsf{v}_i) = \perp$ for every v_i.

Now we begin to describe the definition of $N_\mathsf{v} = (S_\mathsf{v}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_\mathsf{v}, \mathbf{m}_{\mathsf{v},\mathrm{init}})$. The place set

$$S_\mathsf{v} = [!\mathbf{C}_\mathsf{v}]_\mathsf{v} \cup \bigcup_{\mathbf{lk}} \{ [\mathbf{C}]_\mathsf{v}^{\mathbf{lk}} \mid \mathbf{C} \in \mathtt{Comp}(\mathbf{C}_\mathsf{v})\} \cup \bigcup_{i=1}^{n} \bigcup_{j_i=1}^{k_i} \{\mathsf{R}_{\mathsf{v}_i}^{j_i}\} \cup \bigcup_{i=1}^{n} \bigcup_{j_i=1}^{k_i} \{S_{\mathsf{v}_i,j_i}\}.$$

and $\mathtt{Ctxt}([!\mathbf{C}_\mathsf{v}]_\mathsf{v}) = !\langle \mathbf{C}_\mathsf{v}\rangle_\mathsf{v}$, $\mathtt{Ctxt}([\mathbf{C}]_\mathsf{v}^{\mathbf{lk}}) = \langle \mathbf{C}\rangle_\mathsf{v}$ if $\mathbf{C} \in \mathtt{Comp}(\mathbf{C}_\mathsf{v})$, $\mathtt{Ctxt}(\mathsf{R}_{\mathsf{v}_i}^{j_i}) = \mathtt{Ctxt}(S_{\mathsf{v}_i,j_i}) = \mathbf{0}$. The places $\mathsf{R}_{\mathsf{v}_i}^{j_i}$'s act as the resources. Whenever a certain copy of N_{v_i}, say N_{v_i,j_i}, is triggered, the corresponding $\mathsf{R}_{\mathsf{v}_i}^{j_i}$'s are consumed. Meanwhile, the superscript $\mathbf{lk}$ is changed to $\mathbf{lk}[\mathsf{v}_i \mapsto j_i]_{i \in I}$ whose value at v_i, originally $\perp$, is changed to j_i. The initial marking

$$\mathbf{m}_{\mathsf{v},\mathrm{init}} = [!\mathbf{C}_\mathsf{v}]_\mathsf{v} \prod_{i=1}^{n} \prod_{j_i=1}^{k_i} \mathsf{R}_{v_i}^{j_i}.$$

The labeled transition rules $\rightsquigarrow_\mathsf{v}$ is defined by the rules in Fig. 4.

Now we explain the rules in Fig. 4.

The initial process $!\langle P_\mathsf{v}\rangle_\mathsf{v}$ is interpreted as the special marking $[!\mathbf{C}_\mathsf{v}]_\mathsf{v}$. The behavior of $!\langle P_\mathsf{v}\rangle_\mathsf{v}$ is specified by the semantic rules Replication. That is, the transition $!\langle P_\mathsf{v}\rangle_\mathsf{v} \xrightarrow{\lambda} !\langle P_\mathsf{v}\rangle_\mathsf{v} \parallel \langle P'_\mathsf{v}\rangle_\mathsf{v}$ caused by $\langle P_\mathsf{v}\rangle_\mathsf{v} \xrightarrow{\lambda} \langle P'\rangle_\mathsf{v}$. Notice that P_v is $\mathbf{C}_\mathsf{v}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$, in which every subprocess $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v_i}}$ has been interpreted as the initial marking of an arbitrary copy of N_{v_i}. Now the transitions of P_v have four possibilities — 1a, 1b, 1c, and 1d. In the case 1a, the transition of $P_\mathsf{v} \xrightarrow{\lambda} P'$ is caused by $\mathbf{C}_\mathsf{v} \xrightarrow{\lambda} \mathbf{C}'$, and P' is $\mathbf{C}'[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$. After this transition, some subprocesses $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}$ may be active in P', and for

1a. If $\mathbf{C}_{\mathsf{v}} \xrightarrow{\lambda} \mathbf{C}'$, $\mathtt{Act}(\mathbf{C}') = \{\mathsf{v}_i\}_{i\in I}$, we have rule

$$[\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}\ [\mathbf{C}']_{\mathsf{v}}^{\mathbf{lk}_\perp[\mathsf{v}_i\mapsto j_i]_{i\in I}} \prod_{i\in I} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I1a)}$$

for every j_i such that $1 \le j_i \le k_i$.

1b. If $\mathbf{C}_{\mathsf{v}}\{\lambda.\mathbf{0}/\mathsf{v}_h\} \xrightarrow{\lambda} \mathbf{C}_{\mathsf{v}}\{\mathbf{0}/\mathsf{v}_h\}$, $\mathtt{Act}(\mathbf{C}_{\mathsf{v}}) = \{\mathsf{v}_i\}_{i\in I}$, and $\mathbf{m}_{\mathsf{v}_h,\mathrm{init}} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, we have rule

$$[\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}\ [\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}^{\mathbf{lk}_\perp[\mathsf{v}_i\mapsto j_i]_{i\in I}}\ \mathbf{m}'_{\mathsf{v}_h,j_h} \prod_{\substack{i\in I\\ i\neq h}} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I1b)}$$

for every j_i such that $1 \le j_i \le k_i$.

1c. If $\mathbf{C}_{\mathsf{v}}\{l.\mathbf{0}/\mathsf{v}_h, \overline{l}.\mathbf{0}/\mathsf{v}_g\} \xrightarrow{\tau} \mathbf{C}_{\mathsf{v}}\{\mathbf{0}/\mathsf{v}_h, \mathbf{0}/\mathsf{v}_g\}$, $\mathtt{Act}(\mathbf{C}_{\mathsf{v}}) = \{\mathsf{v}_i\}_{i\in I}$, $\mathbf{m}_{\mathsf{v}_h,\mathrm{init}} \overset{l}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, and $\mathbf{m}_{\mathsf{v}_g,\mathrm{init}} \overset{\overline{l}}{\rightsquigarrow}_{\mathsf{v}_g} \mathbf{m}'_{\mathsf{v}_g}$, we have rule

$$[\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}\ [\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}^{\mathbf{lk}_\perp[\mathsf{v}_i\mapsto j_i]_{i\in I}}\ \mathbf{m}'_{\mathsf{v}_h,j_h}\ \mathbf{m}'_{\mathsf{v}_g,j_g} \prod_{\substack{i\in I\\ i\neq h,g}} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I1c)}$$

for every j_i such that $1 \le j_i \le k_i$.

1d. If $\mathbf{C}_{\mathsf{v}}\{l.\mathbf{0}/\mathsf{v}_h\} \xrightarrow{\tau} \mathbf{C}'\{\mathbf{0}/\mathsf{v}_h\}$, $\mathtt{Act}(\mathbf{C}') = \{\mathsf{v}_i\}_{i\in I}$, and $\mathbf{m}_{\mathsf{v}_h,\mathrm{init}} \overset{l}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, we have rule

$$[\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} [\,!\,\mathbf{C}_{\mathsf{v}}]_{\mathsf{v}}\ [\mathbf{C}']_{\mathsf{v}}^{\mathbf{lk}_\perp[\mathsf{v}_i\mapsto j_i]_{i\in I}}\ \mathbf{m}'_{\mathsf{v}_h,j_h} \prod_{\substack{i\in I\\ i\neq h}} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I1d)}$$

for every j_i such that $1 \le j_i \le k_i$.

2a. If $\mathbf{C} \xrightarrow{\lambda} \mathbf{C}'$, $\mathtt{Act}(\mathbf{C}') - \mathtt{Act}(\mathbf{C}) = \{\mathsf{v}_i\}_{i\in I}$, we have rule

$$[\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\mathbf{C}']_{\mathsf{v}}^{\mathbf{lk}[\mathsf{v}_i\mapsto j_i]_{i\in I}} \prod_{i\in I} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I2a)}$$

for every j_i such that $1 \le j_i \le k_i$, and for every $\mathbf{lk}$ such that $\mathbf{lk}(\mathsf{v}_i) = \perp$ for $i \in I$.

2b. If $\mathbf{C}\{\lambda.\mathbf{0}/\mathsf{v}_h\} \xrightarrow{\lambda} \mathbf{C}\{\mathbf{0}/\mathsf{v}_h\}$, and $\mathbf{m}_{\mathsf{v}_h} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, we have rule

$$[\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}\ \mathbf{m}_{\mathsf{v}_h,j_h} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}} [\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}\ \mathbf{m}'_{\mathsf{v}_h,j_h} \qquad \text{(I2b)}$$

for every $\mathbf{lk}$ such that $\mathbf{lk}(\mathsf{v}_h) = j_h$.

2c. If $\mathbf{C}\{l.\mathbf{0}/\mathsf{v}_h, \overline{l}.\mathbf{0}/\mathsf{v}_g\} \xrightarrow{\tau} \mathbf{C}\{\mathbf{0}/\mathsf{v}_h, \mathbf{0}/\mathsf{v}_g\}$, $\mathbf{m}_{\mathsf{v}_h} \overset{l}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, and $\mathbf{m}_{\mathsf{v}_g} \overset{\overline{l}}{\rightsquigarrow}_{\mathsf{v}_g} \mathbf{m}'_{\mathsf{v}_g}$, we have rule

$$[\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}\ \mathbf{m}_{\mathsf{v}_h,j_h}\ \mathbf{m}_{\mathsf{v}_g,j_g} \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} [\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}\ \mathbf{m}'_{\mathsf{v}_h,j_h}\ \mathbf{m}'_{\mathsf{v}_g,j_g} \qquad \text{(I2c)}$$

for every $\mathbf{lk}$ such that $\mathbf{lk}(\mathsf{v}_h) = j_h$ and $\mathbf{lk}(\mathsf{v}_h) = j_g$.

2d. If $\mathbf{C}\{l.\mathbf{0}/\mathsf{v}_h\} \xrightarrow{\tau} \mathbf{C}'\{\mathbf{0}/\mathsf{v}_h\}$, $\mathtt{Act}(\mathbf{C}') - \mathtt{Act}(\mathbf{C}) = \{\mathsf{v}_i\}_{i\in I}$, and $\mathbf{m}_{\mathsf{v}_h} \overset{l}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$, we have rule

$$[\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}\ \mathbf{m}_{\mathsf{v}_h,j_h} \prod_{i\in I} \mathsf{R}_{\mathsf{v}_i}^{j_i} \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} [\mathbf{C}']_{\mathsf{v}}^{\mathbf{lk}[\mathsf{v}_i\mapsto j_i]_{i\in I}}\ \mathbf{m}'_{\mathsf{v}_h,j_h} \prod_{i\in I} \mathbf{m}_{\mathsf{v}_i,\mathrm{init},j_i} \qquad \text{(I2d)}$$

for every j_i such that $1 \le j_i \le k_i$,, and for every $\mathbf{lk}$ such that $\mathbf{lk}(\mathsf{v}_h) = j_h$ and $\mathbf{lk}(\mathsf{v}_i) = \perp$ for $i \in I$.

3. If $\mathbf{m}_1 \overset{l}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}'_1$ and $\mathbf{m}_2 \overset{\overline{l}}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}'_2$ are rules defined by (I1a)–(I1d) and (I2a)–(I2d), we have rule
$$\mathbf{m}_1\ \mathbf{m}_2 \overset{\tau}{\rightsquigarrow}_{\mathsf{v}} \mathbf{m}'_1\ \mathbf{m}'_2 \tag{I3}$$

4. We have rules
$$\mathsf{R}^{j_i}_{\mathsf{v}_i} \xrightarrow{\diamond} \epsilon \tag{I4}$$
for every v_i and for every j_i satisfying $1 \leq j_i \leq k_i$.

Fig. 4. Rules for the Induction Step

every active subprocess, one of N_{v_i,j_i} is attached to the interpretation of P', and the corresponding $\mathsf{R}^{j_i}_{\mathsf{v}_i}$'s are consumed. Now the process $\langle P'_{\mathsf{v}} \rangle_{\mathsf{v}}$ is interpreted as $[\mathbf{C}']_{\mathsf{v}}^{\mathbf{lk}_\perp[\mathsf{v}_i \mapsto j_i]_{i\in I}} \prod_{i\in I} \mathbf{m}_{\mathsf{v}_i,\text{init},j_i}$. Thus we have rule (I1a). Pay attention that the attached 'subnets' can not evolve by the labeled transition rules of themselves. However, these rules are used to produce labeled transition rules of N_{v}. In the case 1b, $P_{\mathsf{v}} \xrightarrow{\lambda} P'$ is caused by one of the active subprocess $!\,\langle P_{\mathsf{v}_h} \rangle_{\mathsf{v}_h} \xrightarrow{\lambda} R$. In this case, we have $\mathbf{C}_{\mathsf{v}}\{\,!\,\langle P_{\mathsf{v}_h} \rangle_{\mathsf{v}_h}/\mathsf{v}_h\} \xrightarrow{\lambda} \mathbf{C}_{\mathsf{v}}\{R/\mathsf{v}_h\}$ ($\mathbf{C}_{\mathsf{v}}$ is unchanged for only guarded choices are concerned). By induction, $!\,\langle P_{\mathsf{v}_h} \rangle_{\mathsf{v}_h} \xrightarrow{\lambda} R$ is interpreted by a transition rule $\mathbf{m}_{\mathsf{v}_h,\text{init}} \overset{\lambda}{\rightsquigarrow}_{\mathsf{v}_h} \mathbf{m}'_{\mathsf{v}_h}$ of N_{v_h}. By the same argument of 1a, we have rule (I1b). The case 1c treats the situation that $P_{\mathsf{v}} \xrightarrow{\tau} P'$ is caused by interaction between two active subprocess $!\,\langle P_{\mathsf{v}_h} \rangle_{\mathsf{v}_h}$ and $!\,\langle P_{\mathsf{v}_g} \rangle_{\mathsf{v}_g}$. The case 1d treats the situation that $P_{\mathsf{v}} \xrightarrow{\tau} P'$ is caused by interaction between one subprocess $!\,\langle P_{\mathsf{v}_h} \rangle_{\mathsf{v}_h}$ and the environment $\mathbf{C}_{\mathsf{v}}$.

The derivatives of $!\,\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}$ must be of the form $!\,\langle P_{\mathsf{v}} \rangle_{\mathsf{v}} \parallel \langle P_1 \rangle_{\mathsf{v}} \parallel \ldots \parallel \langle P_m \rangle_{\mathsf{v}}$, in which every $P_r (1 \leq r \leq m)$ can be represented as $\mathbf{C}\{R_i/\mathsf{v}_i\}_{i=1}^n$ where $R_i \in \mathtt{Drv}(\,!\,\langle P_{\mathsf{v}_i} \rangle_{\mathsf{v}_i})$. The cases 2a – 2d in Fig. 4 deal with the behaviors of $\langle P_r \rangle_{\mathsf{v}}$. The process $\langle P_r \rangle_{\mathsf{v}}$ is interpreted as one of the $[\mathbf{C}]_{\mathsf{v}}^{\mathbf{lk}}$ attached by certain markings of N_{v_i}, j_i for every i satisfying $\mathbf{C} \rhd \mathsf{v}_i$. There are also four possibilities for transitions of $\langle P_r \rangle_{\mathsf{v}}$. In the case 2a, $\langle P_r \rangle_{\mathsf{v}} \xrightarrow{\lambda} P'$ is caused by $\mathbf{C} \xrightarrow{\lambda} \mathbf{C}'$. If some new tags, say $\{\mathsf{v}_i\}_{i\in I}$, become active, the copies of N_{v_i}'s are attached in the same way. This leads to rule (I2a). Case 2b deals with the situation that $\langle P_r \rangle_{\mathsf{v}} \xrightarrow{\lambda} P'$ is caused by $R_h \xrightarrow{\lambda} R'_h$. In the case 2c, $\langle P_r \rangle_{\mathsf{v}} \xrightarrow{\tau} P'$ is caused by interaction between R_h and $R_g \xrightarrow{\lambda}$, while in the case 2d, the transition $\langle P_r \rangle_{\mathsf{v}} \xrightarrow{\tau} P'$ is caused by interaction between R_h and $\mathbf{C}$.

Rule (I3) deals with the case that interaction happens between $\langle P_r \rangle_{\mathsf{v}}$'s, or between $!\,\langle P_{\mathsf{v}} \rangle_{\mathsf{v}}$ and $\langle P_r \rangle_{\mathsf{v}}$.

Rule (I4) says that the resource processes $\mathsf{R}^{j_i}_{v_i}$'s can be consumed without side-effect at any moment. The special label $\diamond$ is used here.

The correctness of the construction in base step and induction step is stated in the next two lemmas.

```
$Trans^!(O_\mathsf{u}, N_\mathsf{u})$:
  1. Find distinct $\mathsf{s}_h \in S_\mathsf{u}$ such that $\mathtt{Ctxt}(\mathsf{s}_h) = \mathbf{C}^h_\mathsf{u}$, and $\mathsf{s} \in S$ such that $\mathtt{Ctxt}(\mathsf{s}) = \mathbf{C}_u$.
  2. for each $h$ for each $\underline{\mathsf{u}_i}$ active in $\mathbf{C}^h_\mathsf{u}$, find arbitrary one copy of $N_{\mathsf{u}_i}$, say $N_{\mathsf{u}_i,j_h} =$
     $(S_{\mathsf{u}_i,j_h}, \mathcal{A}_\diamond, \rightsquigarrow_{\mathsf{u}_i,j_h}, \mathbf{m}_{\mathsf{v}_i,\text{init}}, j_h)$ which is constructed for $!\langle P_{\mathsf{u}_i}\rangle_{\mathsf{u}_i}$, and bind $\mathsf{s}_h$ with
     $N_{\mathsf{u}_i,j_h}$. Let $\mathbf{m}^h_{\mathsf{u}_i} = Trans^!(O^j_{\mathsf{u}_i}, N_{\mathsf{u}_i,j_h})$.
  3. return $\mathsf{s}\ (\prod_h \mathsf{s}_h)\ (\prod_{\mathsf{u}_i,h} \mathbf{m}^h_{\mathsf{u}_i})$.

$Trans(Q, N)$:
  1. Find $\mathsf{s} \in S$ in $N$ such that $\mathtt{Ctxt}(\mathsf{s}) = \mathbf{C}_Q$.
  2. for each $\underline{\mathsf{v}_i}$ active in $\mathbf{C}_Q$, find $N_{\mathsf{v}_i} = (S_{\mathsf{v}_i}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}_i}, \mathbf{m}_{\mathsf{v}_i,\text{init}})$. Let $\mathbf{m}_{\mathsf{v}_i} =$
     $Trans(O_{\mathsf{v}_i}, N_{\mathsf{v}_i})$
  3. return $\mathsf{s}\ \prod_{\mathsf{v}_i} \mathbf{m}_{\mathsf{v}_i}$.
```

Fig. 5. The target process as a marking

Lemma 3. *If a marking* $\mathbf{m}$ *of* N_v *is reachable from* $\mathbf{m}_{\mathsf{v},\text{init}}$*, then* $\mathtt{Proc}(\mathbf{m})$ *is reachable from* $!\langle \mathbf{P}_\mathsf{v}\rangle_\mathsf{v}$.

Lemma 4. *If* $P' \equiv\, !\langle P_\mathsf{v}\rangle_\mathsf{v} \parallel \langle P_1\rangle_\mathsf{v} \parallel \langle P_2\rangle_\mathsf{v} \parallel \ldots \parallel \langle P_n\rangle_\mathsf{v}$ *is reachable from* $!\langle P_\mathsf{v}\rangle_\mathsf{v}$*, and* $\mathbf{num}(!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}, P') \leq \mathbf{num}(!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}, Q)$*, then there is a marking* $\mathbf{m}$ *of* N_v *such that* $\mathtt{Proc}(\mathbf{m}) \equiv P'$ *and* $\mathbf{m}$ *is reachable from* $\mathbf{m}_{\mathsf{v},\text{init}}$.

4.3 Final Step

In the final step, we treat the process P in the form $\mathbf{C}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$, in which $\mathbf{C}[\underline{\mathsf{v}_1}, \underline{\mathsf{v}_2}, \ldots, \underline{\mathsf{v}_n}]$ is the characteristic context of P. The final step is a simplified version of the induction step for the absence of outermost replication operator.

By the induction step, the Labeled Petri Net $N_{\mathsf{v}_i} = (S_{\mathsf{v}_i}, \mathcal{A} \cup \{\diamond\}, \rightsquigarrow_{\mathsf{v}_i}, \mathbf{m}_{\mathsf{v}_i,\text{init}})$ has already been constructed for every $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i}$. In the required Labeled Petri Net $N = (S, \mathcal{A}_\diamond, \rightsquigarrow, \mathbf{m}_\text{init})$, the place set

$$S = \{\, [\mathbf{C}'] \mid \mathbf{C}' \in \mathtt{Comp}(\mathbf{C})\} \cup \bigcup_{i=1}^{n} \{S_{\mathsf{v}_i}\}.$$

and $\mathtt{Ctxt}([\mathbf{C}']) = \mathbf{C}$ if $\mathbf{C}' \in \mathtt{Comp}(\mathbf{C})$. The labeled transition rule of N can be obtained by deleting rule (I1a) – (I1d) in the induction step, and do some modifications on (I2a) – (I2d). The readers are referred to [13] for details.

5 Deciding Reachability Problem

In Section 4, a Labeled Petri Net $N = (S, \mathcal{A}_\diamond, \rightsquigarrow, \mathbf{m}_\text{init})$ is constructed based on both the source process P and the target process Q. P is interpreted as the marking $\mathbf{m}_\text{init}$ of N. If $Q \in \mathtt{Drv}(P)$, we need to find in N the marking $\mathbf{m}_Q$ of Q, and confirm that $\mathbf{m}_Q$ is reachable from $\mathbf{m}_\text{init}$ if and only if $Q \in \mathtt{Drv}(P)$. This work is accomplished in a top-down fashion by procedure $Trans(Q, N)$ in Fig. 5.

If $\mathbf{C}[P_{\mathsf{v}_1}, P_{\mathsf{v}_2}, \ldots, P_{\mathsf{v}_n}]$ and $Q \in \mathtt{Drv}(P)$, then, for some $\mathbf{C}_Q \in \mathtt{Drv}(\mathbf{C})$, Q must be in the form of $\mathbf{C}_Q[\underline{\mathsf{v}_1}, \ldots, \underline{\mathsf{v}_n}]\{O_{\mathsf{v}_i}/\underline{\mathsf{v}_i}\}_{i=1}^{n}$, where O_{v_i} is of the form $!\langle P_{\mathsf{v}_i}\rangle_{\mathsf{v}_i} \parallel \prod_h \langle P_{\mathsf{v}_i,h}\rangle_{\mathsf{v}_i}$. When $\mathbf{C}_Q \rhd \underline{\mathsf{v}_i}$, the subprocedure $Trans^!(O_{\mathsf{v}_i}, N_{\mathsf{v}_i})$ is called in order to get the sub-marking of O_{v_i}.

The procedure $Trans^!(O_{\mathsf{u}}, N_{\mathsf{u}})$ aims at the marking for is called, O_{u} must be of the form

$$!\langle \mathbf{C}_{\mathsf{u}}\{P_{\mathsf{u}_i}/\underline{\mathsf{u}_i}\}_{i=1}^{m}\rangle_{\mathsf{u}} \parallel \prod_h \langle \mathbf{C}_{\mathsf{u}}^h\{O_{\mathsf{u}_i}^h/\underline{\mathsf{u}_i}\}_{i=1}^{m}\rangle_{\mathsf{u}}$$

After that, $Trans^!$ may be called recursively with parameters getting smaller and smaller depending on the structure of Q. It is worth noting that, in case Q does not have the desired structure, Q cannot be a derivative of P, and in this situation $Trans(Q, N)$ will terminate with no marking returned.

Lemma 5. *If Q is a process for which $Trans(Q, N)$ returns a marking $\mathbf{m}$ successfully, then, $\mathbf{m}$ can be reached from $\mathbf{m}_{\text{init}}$ if and only if $Q \in \mathtt{Drv}(P)$.*

6 Concluding Remark

We have presented a deciding procedure of the reachability problem for $\text{CCS}^!$. In order to focus on the main argument, the syntax and semantics are simplified: The rule Replication is REPL1 in [5], while REPL2 is ignored; The guarded choice is used instead of the general choice. The decidability result will not change for such kinds of generalization of $\text{CCS}^!$. The reachability problem can also be confined by only considering τ-transitions. This confined problem is also decidable.

The interplay between replication and localization makes $\text{CCS}^!$ very expressive. Some basic properties of $\text{CCS}^!$ are studied in [5,6,7]. The relative expressiveness of variants of CCS is further studied in [11,10]. It is proved in [6,11] that $\text{CCS}^!$ and CCS^{μ} are less expressive than CCS^{Pdef}. The two problems left open in [11] are both answered positively in [10], which confirms the existence of an encoding from CCS^{μ} to $\text{CCS}^!$ that is codivergent branching bisimilar, and the existence of an encoding from CCS^{μ} to itself with only guarded recursion. The expressiveness of $\text{CCS}^!$ is also studied in [3,4]. A unified approach to the study of relative expressiveness is proposed in [9]. It is shown in [1,2] that $\text{CCS}^!$ can express behavioral types for the π-calculus, while several safety properties are still decidable.

Are there more direct ways to decide the reachability problem for $\text{CCS}^!$? In literature some formalisms which is more powerful than LPN are studied, for example PRS [19] and wPRS, whose reachability problem is decidable [19,16]. Afterward, for these formalism the reachability of HM Property is also decidable [17]. These facts suggest that encoding $\text{CCS}^!$ into PRS or wPRS directly is impossible.

Acknowledgements. The author thanks the anonymous referees for their detailed reviews on the previous version of the paper. Their criticisms, questions and suggestions have led to a significant improvement of the paper. He likes to thank professor Yuxi Fu for helpful suggestions, and to thank Hongfei Fu for discussions in the initial stage of this work.

References

1. Acciai, L., Boreale, M.: Spatial and behavioral types in the pi-calculus. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 372–386. Springer, Heidelberg (2008)
2. Acciai, L., Boreale, M.: Deciding safety properties in infinite-state pi-calculus via behavioural types. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 31–42. Springer, Heidelberg (2009)
3. Aranda, J., Di Giusto, C., Nielsen, M., Valencia, F.D.: CCS with replication in the chomsky hierarchy: The expressive power of divergence. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 383–398. Springer, Heidelberg (2007)
4. Aranda, J., Valencia, F.D., Versari, C.: On the expressive power of restriction and priorities in CCS with replication. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 242–256. Springer, Heidelberg (2009)
5. Busi, N., Gabbrielli, M., Zavattaro, G.: Replication vs. recursive definitions in channel based calculi. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 133–144. Springer, Heidelberg (2003)
6. Busi, N., Gabbrielli, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 307–319. Springer, Heidelberg (2004)
7. Busi, N., Gabbrielli, M., Zavattaro, G.: On the expressive power of recursion, replication and iteration in process calculi. Mathematical Structures in Computer Science 19(6), 1191–1222 (2009)
8. Busi, N., Gorrieri, R.: Distributed conflicts in communicating systems. In: Pareschi, R. (ed.) ECOOP 1994. LNCS, vol. 821, pp. 49–65. Springer, Heidelberg (1994)
9. Fu, Y.: Theory of interaction (2011), Submitted and downloadable at http://basics.sjtu.edu.cn/~yuxi/
10. Fu, Y., Lu, H.: On the expressiveness of interaction. Theor. Comput. Sci. 411(11-13), 1387–1451 (2010)
11. Giambiagi, P., Schneider, G., Valencia, F.D.: On the expressiveness of infinite behavior and name scoping in process calculi. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 226–240. Springer, Heidelberg (2004)
12. Goltz, U.: CCS and petri nets. In: Semantics of Systems of Concurrent Processes, pp. 334–357 (1990)
13. He, C.: The decidability of the reachability problem for CCS! (2011), Downloadable at http://basics.sjtu.edu.cn/~chaodong/
14. He, C., Fu, Y., Fu, H.: Decidability of behavioral equivalences in process calculi with name scoping. In: FSEN (2011)
15. Jancar, P., Moller, F.: Checking regular properties of petri nets. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 348–362. Springer, Heidelberg (1995)
16. Křetínský, M., Řehák, V., Strejček, J.: Extended process rewrite systems: Expressiveness and reachability. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 355–370. Springer, Heidelberg (2004)
17. Křetínský, M., Řehák, V., Strejček, J.: Reachability of hennessy-milner properties for weakly extended PRS. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 213–224. Springer, Heidelberg (2005)
18. Mayr, E.W.: An algorithm for the general petri net reachability problem. In: STOC, pp. 238–246 (1981)

19. Mayr, R.: Process rewrite systems. Inf. Comput. 156(1-2), 264–286 (2000)
20. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River (1989)
21. Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press, Cambridge (1999)
22. Park, D.M.R.: Concurrency and automata on infinite sequences. Theoretical Computer Science, 167–183 (1981)
23. Taubner, D.: Finite Representations of CCS and TCSP Programs by Automata and Petri Nets. LNCS, vol. 369. Springer, Heidelberg (1989)

Static Livelock Analysis in CSP*

Joël Ouaknine, Hristina Palikareva, A.W. Roscoe, and James Worrell

Department of Computer Science, Oxford University, UK
{joel,hrip,awr,jbw}@cs.ox.ac.uk

Abstract. In a process algebra with hiding and recursion it is possible to create processes which compute internally without ever communicating with their environment. Such processes are said to diverge or livelock. In this paper we show how it is possible to conservatively classify processes as livelock-free through a static analysis of their syntax. In particular, we present a collection of rules, based on the inductive structure of terms, which guarantee livelock-freedom of the denoted process. This gives rise to an algorithm which conservatively flags processes that can potentially livelock. We illustrate our approach by applying both BDD-based and SAT-based implementations of our algorithm to a range of benchmarks, and show that our technique in general substantially outperforms the model checker FDR whilst exhibiting a low rate of inconclusive results.

1 Introduction

It is standard in process algebra to distinguish between the visible and invisible (or silent) actions of a process. The latter correspond to state changes arising from internal computation, and their occurrence is not detectable or controllable by the environment. A process is said to *diverge* or *livelock* if it reaches a state from which it may forever compute internally through an infinite sequence of invisible actions. This is usually a highly undesirable feature of the process, described in the literature as "even worse than deadlock" [6, page 156]. Livelock invalidates certain analysis methodologies, and is often symptomatic of a bug in the modelling. However the possibility of writing down divergent processes arises from the presence of two crucial constructs, recursion and hiding. The latter converts visible actions into invisible ones and is a key device for abstraction.

We distinguish two ways in which a process may livelock. In the first, a process may be able to communicate an infinite unbroken sequence of some visible event, and this process then occurs inside the scope of an operator which hides that event. Alternatively, a process may livelock owing to the presence of an unguarded recursion. Roughly speaking, the latter means that the process may recurse without first communicating a visible action.

This paper is concerned with the problem of determining whether a process may livelock in the context of the process algebra CSP, although the principles upon which our analysis is based should be transferable to other process algebras

* A full version of this paper, including all proofs, is available as [11].

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 389–403, 2011.

as well. While it is straightforward to show that the problem is in general undecidable[1], we are still able to provide a conservative (i.e., sound but incomplete) method of checking for the possibility of livelock: this method either correctly asserts that a given process is livelock-free, or is inconclusive. The algorithm is based on a static analysis[2] of the given process, principally in terms of the interaction of hiding, renaming, and recursion. This analysis naturally divides into two parts according to the two sources of livelock outlined above.

The basic intuitions underlying our approach are fairly straightforward. In part they mirror the guardedness requirements which ensure that well-behaved CSP process equations have unique, livelock-free fixed points [13, Chap. 8]. However, we extend the treatment of [13] by allowing guarded recursions to include instances of the hiding operator. Incidentally, Milner's notion of guarded recursions in CCS is similarly restricted by the requirement that variables not occur inside parallel compositions [9]. Complications arise mainly because we want to be able to fully incorporate hiding and renaming in our treatment, both of which can have subtle indirect effects on guardedness.

We note that the idea of guarded recursions is standard in process algebra. For instance, in Milner's framework, a variable is 'strongly guarded' in a given term if every free occurrence of the variable in the term occurs within the scope of a prefixing operator [9]. This notion is introduced in order to justify certain proof principles, such as that guaranteeing the uniqueness of fixed points up to bisimilarity. Strong guardedness has also been extended to a calculus with hiding and action refinement [2]. A key difference between our approach and these notions is that we seek to guarantee livelock-freedom, rather than merely the existence of unique fixed points.

In fact, there are few papers which deal with the problem of guaranteeing livelock-freedom in the setting of concurrent process calculi.[3] The existing work on livelock-freedom has mostly been carried out in the context of mobile calculi. [15] presents an approach for guaranteeing livelock-freedom for a certain fragment of the π-calculus. Unlike the combinatorial treatment presented here, this approach makes use of the rich theory of types of the π-calculus, and in particular the technique of logical relations. Another study of divergence-freedom in the π-calculus appears in [20], and uses the notions of graph types.

Note that CSP is predicated upon *synchronous* (i.e., handshake) communication. In terms of livelock analysis, different issues (and additional difficulties) arise in an asynchronous context (assuming unbounded communication buffers); see, e.g., [7, 8].

Of course, one way to check a process for divergence is to search for reachable cycles of silent actions in its state space, which is a labelled transition system built from the operational semantics. Assuming this graph is finite, this

[1] For example, CSP can encode counters, and is therefore Turing-powerful.

[2] Here *static analysis* is used to distinguish our approach from the state-space exploration methods that underlie model checking or refinement checking.

[3] In contrast, there are numerous works treating termination for the λ-calculus or combinatory logic [5, 10, 4].

can be achieved by calculating its strongly connected components. The latter can be carried out in time linear in the size of the graph, which may however be exponential (or worse) in the syntactic size of the term describing the process. By circumventing the state-space exploration, we obtain a static analysis algorithm which in practice tends to substantially outperform state-of-the-art model-checking tools such as FDR—see Sect. 6 for experimental comparisons.

Naturally, there is a trade-off between the speed and accuracy of livelock checking. It is not hard to write down processes which are livelock-free but which our analysis indicates as potentially divergent. However, when modelling systems in practice, it makes sense to try to check for livelock-freedom using a simple and highly economical static analysis before invoking computationally expensive state-space exploration algorithms. Indeed, as Roscoe [13, page 208] points out, the calculations required to determine if a process diverges are significantly more costly than those for deciding other aspects of refinement, and it is advantageous to avoid these calculations if at all possible.

Recent works in which CSP livelock-freedom plays a key role include [3] as well as [17, 16]; see also references within.

2 CSP: Syntax and Conventions

Let Σ be a finite set of *events*, with $\checkmark \notin \Sigma$. We write $\Sigma^{\checkmark}$ to denote $\Sigma \cup \{\checkmark\}$ and $\Sigma^{*\checkmark}$ to denote the set of finite sequences of elements from Σ which may end with $\checkmark$. In the notation below, we have $a \in \Sigma$ and $A \subseteq \Sigma$. R denotes a binary (renaming) relation on Σ. Its lifting to $\Sigma^{\checkmark}$ is understood to relate $\checkmark$ to itself. The variable X is drawn from a fixed infinite set of process variables.

CSP terms are constructed according to the following grammar:

$$\begin{aligned} P ::= {} & \mathit{STOP} \mid a \longrightarrow P \mid \mathit{SKIP} \mid P_1 \sqcap P_2 \mid P_1 \mathrel{\Box} P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid \\ & P_1 \mathbin{;} P_2 \mid P \setminus A \mid P[R] \mid X \mid \mu X \centerdot P \mid \mathit{DIV} \ . \end{aligned}$$

STOP is the deadlocked process. The prefixed process $a \longrightarrow P$ initially offers to engage in the event a, and subsequently behaves like P. *SKIP* represents successful termination, and is willing to communicate $\checkmark$ at any time. $P \mathrel{\Box} Q$ denotes the external choice of P and Q, whereas $P \sqcap Q$ denotes the internal (or nondeterministic) alternative. The distinction is orthogonal to our concerns, and indeed both choice operators behave identically over our denotational model. The parallel composition $P_1 \underset{A}{\parallel} P_2$ requires P_1 and P_2 to synchronise on all events in A, and to behave independently of each other with respect to all other events. $P \mathbin{;} Q$ is the sequential composition of P and Q: it denotes a process which behaves like P until P chooses to terminate (silently), at which point the process seamlessly starts to behave like Q. $P \setminus A$ is a process which behaves like P but with all communications in the set A hidden. The renamed process $P[R]$ derives its behaviours from those of P in that, whenever P can perform an event a, $P[R]$ can engage in any event b such that $a \mathrel{R} b$. To understand the meaning of $\mu X \centerdot P$, consider the equation $X = P$, in terms of the unknown X. While this equation may have several solutions, it always has a unique least[4] such, written $\mu X \centerdot P$. Moreover, as it turns out, if $\mu X \centerdot P$ is livelock-free then the equation $X = P$ has no other solutions. Lastly, the process DIV represents livelock, i.e., a process caught in an infinite loop of silent events.

A CSP term is *closed* if every occurrence of a variable X in it occurs within the scope of a μX operator; we refer to such terms as *processes.*

Let us state a few conventions. When hiding a single event a, we write $P \setminus a$ rather than $P \setminus \{a\}$. The binding scope of the μX operator extends as far to the right as possible. We also often express recursions by means of the equational notation $X = P$, rather than the functional $\mu X \centerdot P$.

Let us also remark that CSP processes are often defined via *vectors* of mutually recursive equations. These can always be converted to our present syntax, thanks to Bekič's theorem [19, Chap. 10]. Accordingly, we shall freely make use of the vectorised notation in this paper.

3 Operational and Denotational Semantics

We present congruent (equivalent) operational and denotational semantics for CSP. For reasons of space, many details and clauses are omitted; a full account can be found in [11]. An extensive treatment of a variety of different CSP models can also be found in [13, 14]. The semantics presented below only distill those ideas from [13, 14] which are relevant in our setting.

The operational semantics is presented as a list of inference rules in SOS form; we only give below rules for prefixing, recursion, parallel composition, and hiding. In what follows, a stands for a visible event, i.e., belongs to $\Sigma^{\checkmark}$. $A \subseteq \Sigma$ and $A^{\checkmark} = A \cup \{\checkmark\}$. γ can be a visible event or a silent one ($\gamma \in \Sigma^{\checkmark} \cup \{\tau\}$). $P \xrightarrow{\gamma} P'$ means that P can perform an immediate and instantaneous γ-transition, and subsequently become P' (communicating γ in the process if γ is a visible event). If P is a term with a single free variable X and Q is a process, $[Q/X]P$ represents the process P with Q substituted for every free occurrence of X.

$$\frac{}{(a \longrightarrow P) \xrightarrow{a} P} \qquad \frac{}{\mu X \centerdot P \xrightarrow{\tau} [(\mu X \centerdot P)/X]P}$$

$$\frac{P_1 \xrightarrow{\gamma} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\gamma} P_1' \underset{A}{\parallel} P_2}\ [\gamma \notin A^{\checkmark}] \qquad \frac{P_2 \xrightarrow{\gamma} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\gamma} P_1 \underset{A}{\parallel} P_2'}\ [\gamma \notin A^{\checkmark}]$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2'}\ [a \in A^{\checkmark}]$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\ [a \in A] \qquad \frac{P \xrightarrow{\gamma} P'}{P \setminus A \xrightarrow{\gamma} P' \setminus A}\ [\gamma \notin A]\ .$$

[4] The relevant partial order is defined in Sect. 3.

These rules allow us to associate to any CSP process a labelled transition system representing its possible executions. We say that a process *diverges* if it has an infinite path whose actions are exclusively τ's. A process is *livelock-free* if it never reaches a point from which it diverges.

The denotational semantics ascribes to any CSP process a pair (T, D), where $T \subseteq \Sigma^{*\checkmark}$ is the set of visible event traces that the process may perform, and $D \subseteq T$ is the set of traces after which it may diverge.[5] Following [14], we write $\mathcal{T}^{\Downarrow}$ for the set of pairs (T, D) satisfying the following axioms (where $\frown$ denotes trace concatenation):

1. $D \subseteq T$.
2. $s^\frown\langle\checkmark\rangle \in D$ implies $s \in D$.
3. $T \subseteq \Sigma^{*\checkmark}$ is non-empty and prefix-closed.
4. $s \in D \cap \Sigma^*$ and $t \in \Sigma^{*\checkmark}$ implies $s^\frown t \in D$.

Axiom 4 says that the set of divergences is postfix-closed. Indeed, since we are only interested in *detecting* divergence, we treat it as catastrophic and do not attempt to record any meaningful information past a point from which a process may diverge; accordingly, our semantic model takes the view that a process may perform *any* sequence of events after divergence. Thus the only reliable behaviours of a process are those in $T - D$.

Given a process P, its denotation $[\![P]\!] = (\mathsf{traces}(P), \mathsf{divergences}(P)) \in \mathcal{T}^{\Downarrow}$ is calculated by induction on the structure of P; in other words, the model $\mathcal{T}^{\Downarrow}$ is compositional. The complete list of clauses can be found in [13, Chap. 8], and moreover the traces and divergences of a process may also be extracted from the operational semantics in straightforward fashion.

Hiding a set of events $A \subseteq \Sigma$ from a process P introduces divergence if P is capable of performing an infinite unbroken sequence of events from A. Although our model only records the finite traces of a process, the finite-branching nature of our operators entails (via König's lemma) that a process may perform an infinite trace $u \in \Sigma^\omega$ if and only if it can perform all finite prefixes of u.

We interpret recursive processes in the standard way by introducing a partial order $\sqsubseteq$ on $\mathcal{T}^{\Downarrow}$. We write $(T_1, D_1) \sqsubseteq (T_2, D_2)$ if $T_2 \subseteq T_1$ and $D_2 \subseteq D_1$. In other words, the order on $\mathcal{T}^{\Downarrow}$ is reverse inclusion on both the trace and the divergence components. The bottom element of $(\mathcal{T}^{\Downarrow}, \sqsubseteq)$ is $(\Sigma^{*\checkmark}, \Sigma^{*\checkmark})$, i.e., the denotation of the immediately divergent process DIV. The least upper bound of a family $\{(T_i, D_i) \mid i \in I\}$ is given by $\bigsqcup_{i \in I}(T_i, D_i) = (\bigcap_{i \in I} T_i, \bigcap_{i \in I} D_i)$.

It is readily verified that each n-ary CSP operator other than recursion can be interpreted as a Scott-continuous function $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. By induction we have that any CSP expression P in variables $X_1, \ldots, X_n$ is interpreted as a Scott-continuous map $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. Recursion is then interpreted using the least fixed point operator $\mathrm{fix} : [\mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}] \to \mathcal{T}^{\Downarrow}$. For instance $[\![\mu X \centerdot X]\!]$ is the least fixed

[5] Standard models of CSP also take account of the liveness properties of a process by modelling its *refusals*, i.e., the sets of events it cannot perform after a given trace. However, this information is orthogonal to our concerns: the divergences of a process are independent of its refusals—see [13, Sect. 8.4].

point of the identity function on $\mathcal{T}^{\Downarrow}$, i.e., the immediately divergent process. Our analysis of livelock-freedom is based around an alternative treatment of fixed points in terms of metric spaces.

Definition 1. *A process P is livelock-free if* $\mathsf{divergences}(P) = \emptyset$.

In what follows, we make repeated use of standard definitions and facts concerning metric spaces. We refer the reader who might be unfamiliar with this subject matter to the accessible text [18].

Let $F(X)$ be a CSP term with a free variable X. F can be seen as a selfmap of $\mathcal{T}^{\Downarrow}$. Assume that there exists some metric on $\mathcal{T}^{\Downarrow}$ which is complete and under which F is a contraction[6]. Then it follows from the Banach fixed point theorem that F has a unique (possibly divergent) fixed point $\mu X \centerdot F(X)$ in $\mathcal{T}^{\Downarrow}$.

There may be several such metrics, or none at all. Fortunately, a class of suitable metrics can be systematically elicited from the sets of guards of a particular recursion. Roughly speaking, the metrics that we consider are all variants of the well-known ‘longest common prefix’ metric on traces[7], which were first studied by Roscoe in his doctoral dissertation [12], and independently by de Bakker and Zucker [1]. The reason we need to consider such variants is that hiding fails to be nonexpansive in the ‘longest common prefix’ metric. For instance, the distance between the traces $\langle a, a, b\rangle$ and $\langle a, a, c\rangle$ is $\frac{1}{4}$. However, after the event a is hidden, the distance becomes 1. The solution, in this particular case, is to change the definition of the length of a trace by only counting non-a events. To formalise these ideas let us introduce a few auxiliary definitions. These are all parametric in a given set of events $U \subseteq \Sigma$.

Given a trace $s \in \Sigma^{*\checkmark}$, the U-length of s, denoted $\mathsf{length}_U(s)$, is defined to be the number of occurrences of events from U occurring in s. Given a set of traces $T \subseteq \Sigma^{*\checkmark}$ and $n \in \mathbb{N}$ the restriction of T to U-length n is defined by $T \restriction_U n \mathrel{\widehat{=}} \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}$. We extend this restriction operator to act on our semantic domain $\mathcal{T}^{\Downarrow}$ by defining $(T, D) \restriction_U n \mathrel{\widehat{=}} (T', D')$, where

1. $D' = D \cup \{s^\frown t \mid s \in T \cap \Sigma^* \text{ and } \mathsf{length}_U(s) = n\}$.
2. $T' = D' \cup \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}$.

Thus $P \restriction_U n$ denotes a process which behaves like P until n events from the set U have occurred, after which it diverges. It is the least process which agrees with P on traces with U-length no greater than n.

We now define a metric d_U on $\mathcal{T}^{\Downarrow}$ by

$$d_U(P, Q) \mathrel{\widehat{=}} \inf\{2^{-n} \mid P \restriction_U n = Q \restriction_U n\} \ ,$$

where the infimum is taken in the interval $[0, 1]$.

Notice that the function $U \mapsto d_U$ is antitone: if $U \subseteq V$ then $d_U \geqslant d_V$. In particular, the greatest of all the d_U is $d_\emptyset$; this is the discrete metric on $\mathcal{T}^{\Downarrow}$.

[6] A selfmap F on a metric space $(\mathcal{T}^{\Downarrow}, d)$ is a *contraction* if there exists a non-negative constant $c < 1$ such that, for any $P, Q \in \mathcal{T}^{\Downarrow}$, $d(F(P), F(Q)) \leqslant c \cdot d(P, Q)$.

[7] In this metric the distance between two traces s and t is the infimum in $[0, 1]$ of the set $\{2^{-k} \mid s \text{ and } t \text{ possess a common prefix of length } k\}$.

Furthermore, the least of all the d_U is d_Σ; this is the standard metric on $\mathcal{T}^\Downarrow$ as defined in [13, Chap. 8].

Proposition 2. *Let $U \subseteq \Sigma$. Then $\mathcal{T}^\Downarrow$ equipped with the metric d_U is a complete ultrametric space and the set of livelock-free processes is a closed subset of $\mathcal{T}^\Downarrow$. Furthermore if $F : \mathcal{T}^\Downarrow \to \mathcal{T}^\Downarrow$ is contractive with respect to d_U then F has a unique fixed point given by $\lim_{n\to\infty} F^n(STOP)$. (Note that this fixed point may be divergent.)*

In the rest of this paper, the only metrics we are concerned with are those associated with some subset of Σ; accordingly, we freely identify metrics and sets when the context is unambiguous.

4 Static Livelock Analysis

We present an algorithm based on a static analysis which conservatively flags processes that may livelock. In other words, any process classified as livelock-free really is livelock-free, although the converse may not hold.

Divergent behaviours originate in three different ways, two of which are non-trivial. The first is through direct use of the process DIV; the second comes from unguarded recursions; and the third is through hiding an event, or set of events, which the process can perform infinitely often to the exclusion of all others.

Roscoe [13, Chap. 8] addresses the second and third points by requiring that all recursions be *guarded*, i.e., always perform some event prior to recursing, and by banning use of the hiding operator. Our idea is to extend Roscoe's requirement that recursions should be guarded by stipulating that one may never hide *all* the guards. In addition, one may not hide a set of events which a process is able to perform infinitely often to the exclusion of all others. This will therefore involve a certain amount of book-keeping.

We first treat the issue of guardedness of the recursions. Our task is complicated by the renaming operator, in that a purported guard may become hidden only after several unwindings of a recursion. The following example illustrates some of the ways in which a recursion may fail to be guarded, and thus diverge.

Example 3. Let $\Sigma = \{a, b, a_0, a_1, \dots, a_n\}$ and let $R = \{(a_i, a_{i+1}) | 0 \leqslant i < n\}$ and $S = \{(a, b), (b, a)\}$ be renaming relations on Σ. Consider the following processes.

1. $\mu X \centerdot X$.
2. $\mu X \centerdot a \longrightarrow (X \setminus a)$.
3. $\mu X \centerdot (a \longrightarrow (X \setminus b)) \sqcap (b \longrightarrow (X \setminus a))$.
4. $\mu X \centerdot (a_0 \longrightarrow (X \setminus a_n)) \sqcap (a_0 \longrightarrow X[R])$.
5. $\mu X \centerdot SKIP \sqcap a \longrightarrow (X \fatsemi (X[S] \setminus b))$.

The first recursion is trivially unguarded. In the second recursion the guard a is hidden after the first recursive call. In the third process the guard in each summand is hidden in the other summand; this process will also diverge once it has performed a single event. In the fourth example we cannot choose a set of guards which is both stable under the renaming operator and does not contain a_n. This process, call it P, makes the following sequence of visible transitions:

$$P \xrightarrow{a_0} P \setminus a_n \xrightarrow{a_0} P[R] \setminus a_n \xrightarrow{a_1} P[R][R] \setminus a_n \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} P[R][R] \ldots [R] \setminus a_n .$$

But the last process diverges, since P can make an infinite sequence of a_0-transitions which get renamed to a_n by successive applications of R and are then hidden at the outermost level.

A cursory glance at the last process might suggest that it is guarded in $\{a\}$. However, similarly to the previous example, hiding and renaming conspire to produce divergent behaviour. In fact the process, call it P, can make an a-transition to $P \mathbin{⨾} (P[S] \setminus b)$, and thence to $(P[S] \setminus b)[S] \setminus b$ via two τ-transitions. But this last process can diverge.

Given a variable X and a CSP term $P = P(X)$, we aim to define inductively a collection $\mathsf{C}_X(P)$ of metrics for which P is contractive as a function of X (bearing in mind that processes may have several free variables). It turns out that it is first necessary to identify those metrics in which P is merely nonexpansive as a function of X, the collection of which we denote $\mathsf{N}_X(P)$. Intuitively, the role of $\mathsf{N}_X(P)$ is to keep track of all hiding and renaming in P. A set $U \subseteq \Sigma$ then induces a metric d_U under which P is contractive in X provided P is nonexpansive in U and $\mu X \bullet P$ always communicates an event from U prior to recursing.

The collections of metrics that we produce are conservative, i.e., sound, but not necessarily complete. As the examples above suggest, their calculation is made somewhat complicated by the possibility of recursing under renaming. For reasons that will soon become apparent, $\mathsf{N}_X(P)$ and $\mathsf{C}_X(P)$ consist of sets of *pairs* of metrics, or in other words are identified with subsets of $\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$. The key property of the function N_X is given by the following:

Proposition 4. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. If $(U, V) \in \mathsf{N}_X(P)$, then for all $T_1, T_2, \theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, $d_U(T_1, T_2) \geq d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta}))$.*

For R a renaming relation on Σ and $U \subseteq \Sigma$, let $R(U) = \{y \mid \exists x \in U \bullet x \; R \; y\}$. $\mathsf{N}_X(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ is then computed through the following inductive clauses:

$$\mathsf{N}_X(P) \mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \ \ \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:}$$

$$\begin{aligned}
\mathsf{N}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{N}_X(P) \\
\mathsf{N}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{N}_X(P_1) \cap \mathsf{N}_X(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, ⨾, \underset{A}{\|}\} \\
\mathsf{N}_X(P \setminus A) &\mathrel{\widehat{=}} \{(U, V \cup V') \mid (U, V) \in \mathsf{N}_X(P) \wedge V \cap A = \emptyset\} \\
\mathsf{N}_X(P[R]) &\mathrel{\widehat{=}} \{(U, R(V) \cup V') \mid (U, V) \in \mathsf{N}_X(P)\} \\
\mathsf{N}_X(X) &\mathrel{\widehat{=}} \{(U, V) \mid U \subseteq V\} \\
\mathsf{N}_X(\mu Y \bullet P) &\mathrel{\widehat{=}} \{(U \cap U', V \cup V') \mid (U, V) \in \mathsf{N}_X(P) \wedge (V, V) \in \mathsf{N}_Y(P)\} \\
&\qquad \text{if } Y \neq X \ .
\end{aligned}$$

The proof of Prop. 4 proceeds by structural induction on P and can be found in the full version of the paper [11].

Before defining $\mathsf{C}_X(P)$, we need an auxiliary construct denoted $\mathsf{G}(P)$. Intuitively, $\mathsf{G}(P) \subseteq \mathcal{P}(\Sigma)$ lists the 'guards' of ✓ for P. Formally:

Proposition 5. *Let $P(X, Y_1, \dots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \dots, Y_n\}$. If $U \in \mathsf{G}(P)$, then, with any processes—and in particular DIV—substituted for the free variables of P, P must communicate an event from U before it can do a ✓.*

The inductive clauses for G are given below. Note that these make use of the collection of *fair sets* $\mathsf{F}(P_i)$ of P_i, which is presented later on. The definition is nonetheless well-founded since F is here only applied to subterms. The salient feature of $\mathsf{F}(P_i) \neq \emptyset$ is that the process P_i is guaranteed to be livelock-free.

$$\begin{aligned}
\mathsf{G}(STOP) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \\
\mathsf{G}(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{G}(P) \cup \{V \mid a \in V\} \\
\mathsf{G}(SKIP) &\mathrel{\widehat{=}} \emptyset \\
\mathsf{G}(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{G}(P_1) \cap \mathsf{G}(P_2) \quad \text{if } \oplus \in \{\Box, \sqcap\} \\
\mathsf{G}(P_1 \mathbin{;} P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if } P_1 \text{ is closed and } \mathsf{F}(P_1) \neq \emptyset \\ \mathsf{G}(P_1) & \text{otherwise} \end{cases} \\
\mathsf{G}(P_1 \mathbin{\underset{A}{\|}} P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if, for } i = 1, 2,\ P_i \text{ is closed and } \mathsf{F}(P_i) \neq \emptyset \\ \mathsf{G}(P_1) \cap \mathsf{G}(P_2) & \text{otherwise} \end{cases} \\
\mathsf{G}(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \{V \cup V' \mid V \in \mathsf{G}(P) \wedge V \cap A = \emptyset\} & \text{if } P \text{ is closed and} \\ & (\emptyset, \Sigma - A) \in \mathsf{F}(P) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{G}(P[R]) &\mathrel{\widehat{=}} \{R(V) \cup V' \mid V \in \mathsf{G}(P)\} \\
\mathsf{G}(X) &\mathrel{\widehat{=}} \emptyset \\
\mathsf{G}(\mu X \centerdot P) &\mathrel{\widehat{=}} \mathsf{G}(P) \ .
\end{aligned}$$

We are now ready to define $C_X(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, whose central property is given by the following proposition.

Proposition 6. *Let $P(X, Y_1, \dots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \dots, Y_n\}$. If $(U, V) \in \mathsf{C}_X(P)$, then for all $T_1, T_2, \theta_1, \dots, \theta_n \in \mathcal{T}^{\Downarrow}$, $\frac{1}{2} d_U(T_1, T_2) \geq d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta}))$.*

$$\begin{aligned}
\mathsf{C}_X(P) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \quad \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:} \\
\mathsf{C}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{C}_X(P) \cup \{(U, V) \in \mathsf{N}_X(P) \mid a \in V\} \\
\mathsf{C}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap \mathsf{C}_X(P_2) \quad \text{if } \oplus \in \{\Box, \sqcap, \underset{A}{\|}\} \\
\mathsf{C}_X(P_1 \mathbin{;} P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap (\mathsf{C}_X(P_2) \cup \{(U, V) \in \mathsf{N}_X(P_2) \mid V \in \mathsf{G}(P_1)\})
\end{aligned}$$

$$\begin{aligned}
\mathsf{C}_X(P \setminus A) &\mathrel{\widehat=} \{(U, V \cup V') \mid (U,V) \in \mathsf{C}_X(P) \wedge V \cap A = \emptyset\} \\
\mathsf{C}_X(P[R]) &\mathrel{\widehat=} \{(U, R(V) \cup V') \mid (U,V) \in \mathsf{C}_X(P)\} \\
\mathsf{C}_X(X) &\mathrel{\widehat=} \emptyset \\
\mathsf{C}_X(\mu Y \centerdot P) &\mathrel{\widehat=} \{(U \cap U', V \cup V') \mid (U,V) \in \mathsf{C}_X(P) \wedge (V,V) \in \mathsf{N}_Y(P)\} \\
&\qquad\qquad \text{if } Y \neq X \ .
\end{aligned}$$

Note that contraction guarantees a unique fixed point, albeit not necessarily a livelock-free one. For instance, $P(X) = (a \longrightarrow X \setminus b) \mathrel{\Box} (\mu Y \centerdot b \longrightarrow Y)$ has a unique fixed point which can diverge after a single event.

In order to prevent livelock, we must ensure that, whenever a process can perform an infinite[8] unbroken sequence of events from a particular set A, then we never hide the whole of A. To this end, we now associate to each CSP term P a collection of (pairs of) *fair sets* $\mathsf{F}(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$: intuitively, this allows us to keep track of the events which the process is guaranteed to perform infinitely often in any infinite execution of P.

Given a set $W \subseteq \Sigma$, we say that a process is W-fair if any of its infinite traces contains infinitely many events from W. We now have:

Proposition 7. *Let $P(X_1, \ldots, X_n) = P(\overline{X})$ be a CSP term whose free variables are contained within the set $\{X_1, \ldots, X_n\}$. If $(U,V) \in \mathsf{F}(P)$, then, for any collection of livelock-free, U-fair processes $\theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, the process $P(\theta_1, \ldots, \theta_n)$ is livelock-free and V-fair.*

$$\begin{aligned}
\mathsf{F}(\mathit{STOP}) &\mathrel{\widehat=} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \\
\mathsf{F}(a \longrightarrow P) &\mathrel{\widehat=} \mathsf{F}(P) \\
\mathsf{F}(\mathit{SKIP}) &\mathrel{\widehat=} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \\
\mathsf{F}(P_1 \oplus P_2) &\mathrel{\widehat=} \mathsf{F}(P_1) \cap \mathsf{F}(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, ;\} \\
\mathsf{F}(P_1 \mathbin{\underset{A}{\|}} P_2) &\mathrel{\widehat=} (\mathsf{F}(P_1) \cap \mathsf{F}(P_2)) \cup \\
&\quad\ \{(U_1 \cap U_2, V_1) \mid (U_1, V_1) \in \mathsf{F}(P_1) \wedge (U_2, A) \in \mathsf{F}(P_2)\} \cup \\
&\quad\ \{(U_1 \cap U_2, V_2) \mid (U_2, V_2) \in \mathsf{F}(P_2) \wedge (U_1, A) \in \mathsf{F}(P_1)\} \\
\mathsf{F}(P \setminus A) &\mathrel{\widehat=} \{(U, V \cup V') \mid (U,V) \in \mathsf{F}(P) \wedge V \cap A = \emptyset\} \\
\mathsf{F}(P[R]) &\mathrel{\widehat=} \{(U, R(V) \cup V') \mid (U,V) \in \mathsf{F}(P)\} \\
\mathsf{F}(X) &\mathrel{\widehat=} \{(U,V) \mid U \subseteq V\} \\
\mathsf{F}(\mu X \centerdot P) &\mathrel{\widehat=} \begin{cases} \{(U \cap U', U \cup V') \mid (U,U) \in \mathsf{C}_X(P) \cap \mathsf{F}(P)\} & \text{if } \mu X \centerdot P \text{ is open} \\ \mathcal{P}(\Sigma) \times \{U \cup V' \mid (U,U) \in \mathsf{C}_X(P) \cap \mathsf{F}(P)\} & \text{otherwise} \ . \end{cases}
\end{aligned}$$

We now obtain one of our main results as an immediate corollary:

Theorem 8. *Let P be a CSP process (i.e., closed term) not containing DIV in its syntax. If $\mathsf{F}(P) \neq \emptyset$, then P is livelock-free.*

[8] Recall our understanding that a process can 'perform' an infinite trace iff it can perform all its finite prefixes.

5 Structurally Finite-State Processes

The techniques developed in Sections 3 and 4 allow us to handle the widest range of CSP processes; among others, it enables one to establish livelock-freedom of numerous infinite-state processes including examples making use of infinite buffers or unbounded counters—see [11] for examples. Such processes are of course beyond the reach of explicit-state model checkers such as FDR. In order to create them in CSP, it is necessary to use devices such as recursing under the parallel operator. In practice, however, the vast majority of processes tend to be finite state.

Let us therefore define a CSP process to be *structurally finite state* if it never syntactically recurses under any of parallel, the left-hand side of a sequential composition, hiding, or renaming.

More precisely, we first define a notion of *sequential* CSP terms: $STOP$, $SKIP$, and X are sequential; if P and Q are sequential, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \,\Box\, Q$, and $\mu X \centerdot P$; and if in addition P is closed, then $P \mathbin{;} Q$, $P \setminus A$, and $P[R]$ are sequential. Observe that sequential processes give rise to labelled transition systems of size linear in the length of their syntax.

Now any closed sequential process is deemed to be structurally finite state; and if P and Q are structurally finite state, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \,\Box\, Q$, $P \underset{A}{\parallel} Q$, $P \mathbin{;} Q$, $P \setminus A$, and $P[R]$. Note that structurally finite-state CSP terms are always closed, i.e., are processes.

Whether a given process is structurally finite state can easily be established by syntactic inspection. For such processes, it turns out that we can substantially both simplify and sharpen our livelock analysis. More precisely, the computation of nonexpansive and contractive data is circumvented by instead directly examining closed sequential components in isolation. Furthermore, the absence of free variables in compound processes makes some of the earlier fairness calculations unnecessary, thereby allowing more elaborate and finer data to be computed efficiently, as we now explain.

Let u be an infinite trace over Σ, and let $F, C \subseteq \Sigma$ be two sets of events. We say that u is *fair in* F if, for each $a \in F$, u contains infinitely many occurrences of a,[9] and we say that u is *co-fair in* C if, for each $b \in C$, u contains only finitely many occurrences of b.

Given a structurally finite-state process P, we compute a collection of *fair/co-fair* pairs of sets $\Phi(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, together with a Boolean-valued *livelock flag* $\delta(P) \in \{\text{true}, \text{false}\}$, giving rise to our second main result:

Theorem 9. *Let P be a structurally finite-state process. Write $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\}$. If $\delta(P) = \text{false}$, then P is livelock-free, and moreover, for every infinite trace u of P, there exists $1 \leq i \leq k$ such that u is both fair in F_i and co-fair in C_i.*

[9] Note that this notion of 'fairness' differs from that used in the previous section.

The calculation of $\Phi(P)$ proceeds inductively as follows. For P a closed sequential process, $\Phi(P)$ is computed directly from the labelled transition system associated with P.[10] Otherwise:

$$\begin{aligned}
\Phi(a \longrightarrow P) &\mathrel{\widehat{=}} \Phi(P) \\
\Phi(P_1 \oplus P_2) &\mathrel{\widehat{=}} \Phi(P_1) \cup \Phi(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, \fatsemi\} \\
\Phi(P_1 \mathbin{\underset{A}{\|}} P_2) &\mathrel{\widehat{=}} \{(F_1 \cup F_2, (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))) \mid \\
&\qquad\qquad (F_i, C_i) \in \Phi(P_i) \text{ for } i = 1, 2\} \cup \\
&\quad\; \{(F, C) \mid (F, C) \in \Phi(P_1) \wedge F \cap A = \emptyset\} \cup \\
&\quad\; \{(F, C) \mid (F, C) \in \Phi(P_2) \wedge F \cap A = \emptyset\} \\
\Phi(P \setminus A) &\mathrel{\widehat{=}} \{(F - A, C \cup A) \mid (F, C) \in \Phi(P)\} \\
\Phi(P[R]) &\mathrel{\widehat{=}} \{(F, C) \mid (F', C') \in \Phi(P) \wedge \\
&\qquad\qquad C \subseteq \{b \in \Sigma \mid R^{-1}(b) \subseteq C'\} \wedge F \subseteq R(F')\} \ .
\end{aligned}$$

The calculation of $\delta(P)$ similarly proceeds inductively, making use of the fair/co-fair data, as follows. If P is a closed sequential process, then $\delta(P)$ is determined directly from the labelled transition system associated with P, according to whether the latter contains a τ-cycle or not (using, e.g., Tarjan's algorithm). Otherwise:

$$\begin{aligned}
\delta(a \longrightarrow P) &\mathrel{\widehat{=}} \delta(P) \\
\delta(P_1 \oplus P_2) &\mathrel{\widehat{=}} \delta(P_1) \vee \delta(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, \underset{A}{\|}, \fatsemi\} \\
\delta(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \text{false} & \text{if } \delta(P) = \text{false and, for each } (F, C) \in \Phi(P),\ F - A \neq \emptyset \\ \text{true} & \text{otherwise} \end{cases} \\
\delta(P[R]) &\mathrel{\widehat{=}} \delta(P) \ .
\end{aligned}$$

Theorems 8 and 9 yield a conservative algorithm for livelock-freedom: given a CSP process P (which we will assume does not contain DIV in its syntax), determine first whether P is structurally finite state. If so, assert that P is livelock-free if $\delta(P) = \text{false}$, and otherwise report an inconclusive result. If P is not structurally finite state, assert that P is livelock-free if $\mathsf{F}(P) \neq \emptyset$, and otherwise report an inconclusive result.

[10] It is worth pointing out how this can be achieved efficiently. Given a set $L \subseteq \Sigma$ of events, delete all $(\Sigma - L)$-labelled transitions from P's labelled transition system. If the resulting graph contains a (not necessarily reachable) strongly connected component which comprises every single event in L, include $(L, \Sigma - L)$ as a fair/co-fair pair for P.

Of course, in actual implementations, it is not necessary to iterate explicitly over all possible subsets of Σ. The computation we described can be carried out symbolically using a Boolean circuit of size polynomial in $|\Sigma|$, using well-known circuit algorithms for computing the transitive closure of relations. Consequently, $\Phi(P)$ can be represented symbolically and compactly either as a BDD or a propositional formula. Further implementation details are provided in Sect. 6.

It is perhaps useful to illustrate how the inherent incompleteness of our procedure can manifest itself in very simple ways. For example, let $P = a \longrightarrow Q$ and $Q = (a \longrightarrow P) \square (b \longrightarrow Q)$, and let $R = (P \underset{\{a,b\}}{\parallel} Q) \setminus b$. Using Bekič's procedure, R is readily seen to be (equivalent to) a structurally finite-state process. Moreover, R is clearly livelock-free, yet $\delta(R) = \text{true}$ and $\mathsf{F}(R) = \emptyset$. Intuitively, establishing livelock-freedom here requires some form of state-space exploration, to see that the 'divergent' state $(Q \underset{\{a,b\}}{\parallel} Q) \setminus b$ of R is in fact unreachable, but that is precisely the sort of reasoning that our static analysis algorithm is not geared to do.

Nonetheless, we have found in practice that our approach succeeded in establishing livelock-freedom for a wide range of existing benchmarks; we report on some of our experiments in Sect. 6.

Finally, it is worth noting that, for structurally finite-state processes, Theorem 9 is stronger than Theorem 8—it correctly classifies a larger class of processes as being livelock-free—and empirically has also been found to yield faster algorithms.

6 Implementation and Experimental Results

Computationally, the crux of our algorithm revolves around the manipulation of sets. We have built both BDD-based and propositional-formula-based implementations, using respectively CUDD 2.4.2 and MiniSat 2.0 for computations. Our resulting tool was christened SLAP, for Static Livelock Analyser of Processes.

We experimented with a wide range of benchmarks, including parameterised, parallelised, and piped versions of Milner's Scheduler, the Alternating Bit Protocol, the Sliding Window Protocol, the Dining Philosophers, Yantchev's Mad Postman Algorithm, as well as a Distributed Database algorithm.[11] In all our examples, internal communications were hidden, so that livelock-freedom can be viewed as a progress or liveness property. All benchmarks were livelock-free, although the reader familiar with the above examples will be aware that manually establishing livelock-freedom for several of these can be a subtle exercise.

In all cases apart from the Distributed Database algorithm, SLAP was indeed correctly able to assert livelock-freedom (save for rare instances of timing out). (Livelock-freedom for the Distributed Database algorithm turns out to be remarkably complex; see [13] for details.) In almost all instances, both BDD-based and SAT-based implementations of SLAP substantially outperformed the state-of-the-art CSP model checker FDR, often completing orders of magnitude faster. On the whole, BDD-based and SAT-based implementations performed comparably, with occasional discrepancies. All experiments were carried out on a 3.07GHz Intel Xeon processor running under Ubuntu with 8 GB of RAM.

[11] Scripts and descriptions for all benchmarks are available from the website associated with [14]; the reader may also wish to consult [11] for further details on our case studies.

Table 1. Times reported are in seconds, with * denoting a 30-minute timeout

Benchmark	**FDR**	**SLAP** (BDD)	**SLAP** (SAT)
Milner-15	0	0.19	0.16
Milner-20	409	0.63	0.34
Milner-21	948	0.73	0.22
Milner-22	*	0.89	0.25
Milner-25	*	1.63	0.55
Milner-30	*	7.34	1.14
ABP-0	0	0.03	0.03
ABP-0-inter-2	0	0.03	0.04
ABP-0-inter-3	23	0.03	0.06
ABP-0-inter-4	*	0.03	0.07
ABP-0-inter-5	*	0.03	0.08
ABP-0-pipe-2	0	0.03	0.08
ABP-0-pipe-3	2	0.04	0.12
ABP-0-pipe-4	175	0.04	0.23
ABP-0-pipe-5	*	0.05	0.34
ABP-4	0	0.11	0.92
ABP-4-inter-2	39	0.12	1.49
ABP-4-inter-3	*	0.13	1.71
ABP-4-inter-7	*	0.15	3.68
ABP-4-pipe-2	12	0.13	2.96
ABP-4-pipe-3	*	0.15	6.34
ABP-4-pipe-7	*	0.25	31.5
SWP-1	0	0.03	0.08
SWP-2	0	0.34	*
SWP-3	0	40.94	*
SWP-1-inter-2	0	0.04	0.12
SWP-1-inter-3	31	0.04	0.16
SWP-1-inter-4	*	0.05	0.19
SWP-1-inter-7	*	0.06	0.33
SWP-2-inter-2	170	0.47	*
SWP-2-inter-3	*	0.64	*
SWP-1-pipe-3	0	0.04	0.47
SWP-1-pipe-4	3	0.05	0.73
SWP-1-pipe-5	246	0.05	1.10
SWP-1-pipe-7	*	0.06	2.89
Philosophers-7	2	1.64	0.20
Philosophers-8	20	2.46	0.31
Philosophers-9	140	3.99	0.46
Philosophers-10	960	7.39	0.61
Mad Postman-2	0	0.06	0.04
Mad Postman-3	6	*	0.23
Mad Postman-4	*	*	1.11
Mad Postman-5	*	*	5.67
Mad Postman-6	*	*	27.3

Times in seconds are given in Table 1, with * indicating a 30-minute timeout. Further details of the experiments are provided in [11].

7 Future Work

A interesting property of our approach is the possibility for our algorithm to produce a *certificate* of livelock-freedom, consisting among others in the various sets supporting the final judgement. Such a certificate could then be checked in polynomial time by an independent tool.

Other directions for future work include improving the efficiency of SLAP by incorporating various abstractions (such as collapsing all events on a given channel, or placing *a priori* bounds on the size of sets), or conversely increasing accuracy at modest computational cost, for example by making use of algebraic laws at the syntactic level, such as bounded unfoldings of parallel compositions.

References

[1] De Bakker, J.W., Zucker, J.I.: Processes and the denotational semantics of concurrency. Information and Control 54, 70–120 (1982)

[2] Bravetti, M., Gorrieri, R.: Deciding and axiomatizing weak ST bisimulation for a process algebra with recursion and action refinement. ACM Transactions on Computational Logic 3(4), 465–520 (2002)

[3] Dimovski, A.: A compositional method for deciding program termination. In: ICT Innovations, vol. 83, pp. 71–80. Springer, Heidelberg (2010)

[4] Gandy, R.O.: An early proof of normalization by A.M. Turing. In: To, H.B. (ed.) Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, vol. 267, pp. 453–455. Academic Press, London (1980)

[5] Girard, J.-Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge Tracts in Theoretical Science, vol. 7. Cambridge University Press, Cambridge (1988)

[6] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, London (1985)

[7] Leue, S., Ştefănescu, A., Wei, W.: A livelock freedom analysis for infinite state asynchronous reactive systems. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 79–94. Springer, Heidelberg (2006)

[8] Leue, S., Ştefănescu, A., Wei, W.: Dependency analysis for control flow cycles in reactive communicating processes. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 176–195. Springer, Heidelberg (2008)

[9] Milner, R.: Communication and Concurrency. Prentice-Hall International, London (1989)

[10] Mitchell, J.C.: Foundations for Programming Languages. MIT Press, Cambridge (1996)

[11] Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Static livelock analysis for CSP: Full version (2011), http://www.cs.ox.ac.uk/people/Joel.Ouaknine/download/slaptr.pdf

[12] Roscoe, A.W.: A Mathematical Theory of Communicating Processes. PhD thesis, Oxford University (1982)

[13] Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall International, London (1997)

[14] Roscoe, A.W.: Understanding Concurrent Systems. Springer, Heidelberg (2011), http://www.cs.ox.ac.uk/ucs/

[15] Sangiorgi, D.: Types, or: Where's the difference between CCS and π? In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 76–97. Springer, Heidelberg (2002)

[16] Schneider, S., Treharne, H., Wehrheim, H.: A CSP approach to control in Event-B. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 260–274. Springer, Heidelberg (2010)

[17] Schneider, S., Treharne, H., Wehrheim, H.: A CSP account of Event-B refinement (2011) (unpublished)

[18] Sutherland, W.A.: Introduction to Metric and Topological Spaces. Oxford University Press, Oxford (1975)

[19] Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)

[20] Yoshida, N., Berger, M., Honda, K.: Strong normalisation in the π-Calculus. In: Proceedings of LICS 2001, pp. 311–322. IEEE Computer Society Press, Los Alamitos (2001)

Dynamic Reactive Modules*

Jasmin Fisher[1], Thomas A. Henzinger[2], Dejan Nickovic[2], Nir Piterman[3], Anmol V. Singh[2], and Moshe Y. Vardi[4]

[1] Microsoft Research, Cambridge, UK
[2] IST Austria, Klosterneuburg, Austria
[3] University of Leicester, UK
[4] Rice University, Houston, TX, USA

Abstract. State-transition systems communicating by shared variables have been the underlying model of choice for applications of model checking. Such formalisms, however, have difficulty with modeling process creation or death and communication reconfigurability. Here, we introduce "dynamic reactive modules" (DRM), a state-transition modeling formalism that supports dynamic reconfiguration and creation/death of processes. The resulting formalism supports two types of variables, data variables and reference variables. Reference variables enable changing the connectivity between processes and referring to instances of processes. We show how this new formalism supports parallel composition and refinement through trace containment. DRM provide a natural language for modeling (and ultimately reasoning about) biological systems and multiple threads communicating through shared variables.

1 Introduction

State-transition systems provide a natural formalism in many areas of computer science. They provide a convenient framework for understanding programming languages (cf. [21]), provide a natural executable modeling framework for reactive and concurrent systems (cf., [11]), provide the most intuitive semantics for the application of model checking (cf. [4]), and even proved to be useful to the development of biological models [7,10,8,9], where the straightforward semantics make these formalisms natural and attractive for cell biologists. State-transition systems capture elegantly the concept of a system with variables that change their values over time. The state-transition approach to modeling concurrent systems can be fairly described as enormously successful, combining executability, explorability, and analyzability. In the state-transition approach communication is typically modeled via shared variables, while in the complementary approach of process calculi communication is modeled via message passing [18].

In recent years, new application domains that stress mobility and dynamic reconfigurability gained importance. In mobile and ad-hoc networks, network elements come and go, changing communication configuration according to their position. The state-transition approach, however, does not model naturally reconfigurable systems. Similarly, it has difficulty with dynamics features of biological systems, such as cell movement, division, and death.

* This work was supported by the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 (RiSE), and the EU NOE Grant ArtistDesign.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 404–418, 2011.

In the process-calculi approach, the π-*calculus* has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication [19,20]. The power of the π-calculus comes from its ability to transmit processes as messages, a mathematically natural and powerful construct. This idea immediately allows the encoding of dynamic aspects and has been widely accepted by the research community (cf. [17]). No analogous widely acceptable extension exists for the state-transition approach, enabling the modeling of mobility and reconfigurability.

In this paper, we propose a state-transition formalism that supports reconfiguration of communication and dynamic creation of new processes. We accomplish this by adapting to the state-transition approach three fundamental language mechanisms of modern programming languages: *encapsulation*, *composition*, and *reference*. Encapsulation is a language mechanism for bundling together related data and methods, while restricting access to some of those. Composition is a language mechanism for composing such bundles of data and methods. Finally, reference is a language mechanism for creating such bundles dynamically. While the mechanisms of encapsulation and composition have been used in state-transition formalisms, for example, in *reactive modules* [2], which are the basis for our work here, it is striking that the concept of a reference is missing in all state-based modeling formalisms, while it is present in every reasonable imperative programming language. We show how this well known and widely used concept in programming offers a powerful modeling concept in the state-transition context.

In the reactive-modules formalism, modules define behavior of a bundled set of variables. Behavior of a module is defined through that of its variables, partitioned to internal, interface, and external variables. The module controls its internal and interface variables and reads the external variables from other modules. To allow executability, an update round is partitioned to subrounds. Variables that are co-updated in the same round are not allowed to depend on one another. Thus, the module mechanism essentially supports encapsulation. Then, composition is supported by the ability to compose modules in parallel, and the ability to make multiple copies of modules.

Modern imperative object-oriented programming languages combine our guiding principles: encapsulation, composition, and reference. A *class* is a schema of encapsulated behavior. It has a well defined interface that cleanly supports composition. An *object* has to be *instantiated*, returning a reference through which it can be accessed. It then executes according to its prescribed behavior. Different instantiations of the same class behave differently according to their individual histories, which are stored in their own variables. References, in addition to enabling us to create multiple instances of the same class, allow us to dynamically change the configuration of instances in memory. Classes and references together allow us to organize the program in multiple levels of abstraction and manage (to some extent) the complexity of software.

Here, we adapt these concepts to the world of state-transition modeling. In this context, the instantiation of an object also assigns "dynamic computation power" to it: every newly instantiated variable includes with it a recipe for behavior as a function of the values of some other variables. Our "objects" are independent processes each controlling a set of variables. We impose encapsulation by assigning ownership to variables. Each process has its own variables, which it and it alone can change; the update may depend on the values of variables that it does not own. Thus, our variables are single-write multiple-read variables. These variables can be accessed either, traditionally, by direct static sharing, or via references, by dynamic sharing, enabling dynamic communication configurations. In addition, we model processes that join, leave, or emerge by a specialized creation command, which is analogous to allocating new memory from the heap. Here, again, references are invaluable, as they allow communication in both directions: for a newly instantiated process, this enables initial knowledge about its environment; for an instantiating process this enables access to some of the newly created variables.

There have been few attempts to handle dynamicity in state-transition formalisms. Dynamic I/O automata [3] are an extension of *I/O automata* [16]. In order to change communication configuration, explicit state-based modeling of the reconfiguration is needed (through changing alphabet signatures from state to state). Alur and Grosu extend reactive modules by creation through the usage of unbounded arrays [1]. Global information regarding arrays and their length is required, as indeed exhibited in the "reconfiguration controller" that controls the entire system. Updates are done via λ-expressions on entire arrays and not locally. This makes it impossible to apply multiple levels of abstraction, one of the main strengths of programming languages. This is akin to viewing the heap as a linear sequence of memory locations and using integers as pointers into the array. This gives a low level implementation of the heap, depriving the programmer of the ability to abstract. Lucid-Synchrone is an extension of *Lustre* that supports creation but restricts to a fixed topology [5]. There have been attempts to add object-orientation to statecharts, an important state-transition formalism. For example, in [13,12], the semantics of Rhapsody, an object-orientation extension of StateMate, is described in terms of the underlying programming languages. Thus, they bypass the need to reason about dynamic creation of processes. Damm et al. [6] give a specialized semantics for UML Statecharts. Their formalism is cumbered by the need to support directly many specialized features of Statecharts and does not offer a general solution.

Our main contribution is a new state-transition formalism, based on widely used object-oriented programming paradigms, that supports communication via shared variables and dynamic reconfiguration and creation. We show that our formalism supports a straightforward trace semantics, where refinement corresponds to trace containment over appropriate projections and composition corresponds to a specialized form of intersection. In addition, we allow partial specifications that translate to nondeterminism, just like in standard state-transition settings, and their refinement, through a replacement operator. We provide a rich modeling formalism that suggests many future directions.

2 High-Level Description of Dynamic Reactive Modules

We generalize *reactive modules* [2] to *dynamic reactive modules* by including reference variables and the ability to create new modules. This is similar to modern object-oriented languages, where a reference variable refers to an instance of a class. A class definition describes the way to update multiple included variables and an instantiation leads to allocation of memory. While in standard objects the data is updated only via explicit method invocation; in dynamic reactive modules variables continually update their values according to update rules. Thus, instantiating a module leads to allocation of new variables that are updated simultaneously in all instantiated modules. In this section,

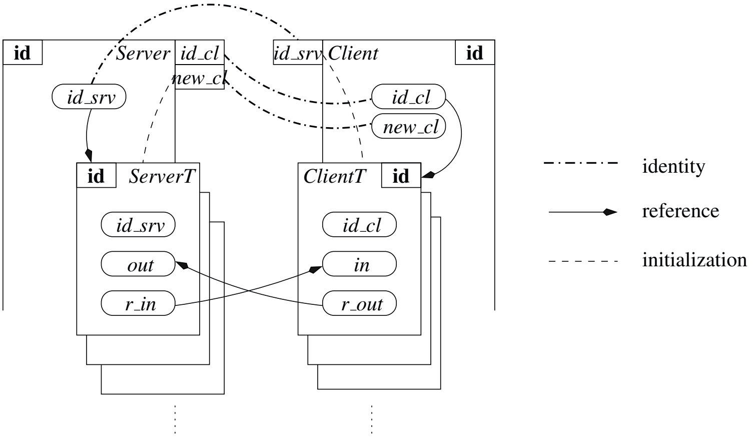


Fig. 1. A server/client system

we further motivate the need for- and introduce dynamic reactive modules through an example of a simple server/client model. Definitions are made formal in Section 4.

Consider the diagram in Figure 1. It includes a *server* and a *client*. The client generates new *client thread*s at arbitrary times. The server detects that a new client thread has been generated and allocates a new *server thread* dedicated to serve the respective client thread's request. The server and the client need to produce a pair of threads and connect them so that the server thread reads the client thread's input *in* and the client thread reads the server thread's output *out*. For that, the server thread will initialize its reference variable *r_in* to refer to *in* and the client thread will initialize its reference variable *r_out* to refer to *out*. Once a pair of server thread and client thread have been connected the server and the client can forget about them and create (and mutually initialize) a new pair. Every newly instantiated module gets a unique identifier and its own reference to itself, through the special **id** variable (akin to *this*). The mutual references between server thread and client thread variables are exchanged between the server and the client through static communication and passed to the corresponding thread as parameter. This exchange of references is done by mutually accessing external variables *id_cl* and *id_srv* that hold references to newly created client thread and server thread, respectively. In addition, the server accesses client's variable *new_cl* that signals when a new client thread is created. We generalize the notion of module to that of a *dynamic module*. We distinguish between (a) a dynamic module *class*, which defines the module, its variables, and how to update them and (b) a dynamic module, which is the actual instantiation. A *dynamic system* defines a collection of dynamic-module classes.

In Figure 2, we include the code for the *ServerClient* dynamic reactive system that models the above example. It consists of four modules depicted in Figure 2, together with the (initial) module *Server* || *Client* that denotes the composition of *Server* and *Client* modules. Every module in the system consists of a declaration, that defines the variables owned by the module, and a body that specifies initialization and update rules for these variables. The module body has a finite set of typed variables that are partitioned into *controlled* and *external* variables and either range over finite domains or

```
system $ServerClient$ =
    $\langle\{Server, ServerT, Client, ClientT, Server \parallel Client\}, Server \parallel Client\rangle$

class $Server$
external $id\_cl : \mathcal{R}, new\_cl : \mathbb{B}$
control $id\_srv : \mathcal{R}$

atom $id\_srv$
  init
    $[]\ true \rightarrow id\_srv' := \mathbf{0}$
  update
    $[]\ new\_cl \rightarrow id\_srv' :=$
          new $ServerT(id\_cl')$

class $ServerT$
param $id\_cl : \mathcal{R}$
control $out : \mathbb{B}, r\_in : \mathcal{R}$

atom $r\_in$
  init
    $[]\ id\_cl' = \mathbf{0} \rightarrow r\_in' := \mathbf{0}$
    $[]\ id\_cl' \neq \mathbf{0} \rightarrow r\_in' :=$
          ref$(id\_cl'.in)$
  update
    $[]\ true \rightarrow$
atom $out$
  initupdate
    $[]\ r\_in' \neq \mathbf{0} \rightarrow out' :=$
          $f($deref$(r\_in'))$

class $Client$
external $id\_srv : \mathcal{R}$
control $id\_cl : \mathcal{R}, new\_cl : \mathbb{B}$

atom $new\_cl$
  initupdate
    $[]\ true \rightarrow new\_cl' := true$
    $[]\ true \rightarrow new\_cl' := false$

atom $id\_cl$
  init
    $[]\ true \rightarrow id\_cl' := \mathbf{0};$
  update
    $[]\ new\_cl \rightarrow id\_cl' :=$
          new $Client(id\_srv')$
class $ClientT$
param $id\_srv : \mathcal{R}$
control $in : \mathbb{B}, r\_out : \mathcal{R}$

atom $r\_out$
  init
    $[]\ id\_srv' = \mathbf{0} \rightarrow r\_out' := \mathbf{0}$
    $[]\ id\_srv' \neq \mathbf{0} \rightarrow r\_out' :=$
          ref$(id\_srv'.out)$
  update
    $[]\ true \rightarrow$
```

Fig. 2. Server/client system modeled with dynamic reactive modules

are reference variables. Additionally, a module has a set of *parameters* and a special variable **id**, which holds the identifier of an instance of the module. Parameter variables are used for initialization of the module according to some information from its environment.

Reference variables establish dynamic communication between module instances. When a module is instantiated, its variable **id** is assigned a unique identifier. For example, $\mathbf{id}.m$ and $\mathbf{id}.n$ use the variable **id** to indirectly access many variables of the same module. We add the two basic functionalities of references. First, the ability to take the address of a variable through ref(x), which returns a reference to x. Second, the ability to dereference a variable and access the value of the variable that it references.

The variable *id_srv* (*id_cl*) holds a reference to a server (client) thread, it is controlled by *Server* (*Client*) and is external to *Client* (*Server*). In addition, *Client* controls *new_cl* (external to *Server*) that signals the instantiation of a new client in the system. *Client* and *Server* communicate statically over these three variables, and mutually exchange references between newly created client and server threads. The communication between the server thread and the client thread has to be dynamic (via reference variables) as the two are instantiated independently. For that, server (client) thread holds reference *r_in* (*r_out*) to the client (server) thread's variable *in* (*out*). The server (client) thread's identifier is passed to the client (server) thread through the parameter *id_srv* (*id_cl*) upon its instantiation. We use the dereference operation to update *out* of the server thread based on the value of *in* of the client, through the expression $f(\mathsf{deref}(r\_in))$.

The module body consists of a set of *atoms* that group rules for setting values to variables owned by the module. Atoms of the module control precisely its controlled variables, and every controlled variable declared in the module is controlled by exactly one atom. We distinguish between the *current* value of a variable, denoted x, and its *next* value x'. Atoms contain initialization and update rules, or commands, that define the value of x' based on current and next values of variables declared in the module. When a module is instantiated, its variables do not have current values. Thus, initial commands may use only next values of variables in the same module, or the values of parameters passed to it. Update commands may refer to both current and next values of variables and can either define an instantiation of a new instance of a module, or a classic update of a variable as a function of current and next values of other variables.

The atom that controls *new_cl* in the *Client* sets the next value *new_cl′* to either *true* or *false*, nondeterministically. An instantiation is a special type of update, using the command new . It can update the reference variables of the instantiating module to refer to the newly instantiated module and uses parameters to pass information to the instantiated module for proper initialization. For example, the update in the atom that controls *id_cl* either takes no action (if *new_cl* is *false*) or instantiates a new *ClientT*. The instantiation updates the reference variable *id_cl′* to refer to the variable **id** of the newly instantiated *ClientT*, which holds the unique identifier of this client thread. When the new client thread is created it receives the value of the identifier of the *ServerT* instance in variable *id_srv′* through the parameter *id_cl*. Passing the identifier to an instance of a module enables access to all variables of that instance. For example, in the initial command in the atom *r_out*, if parameter *id_srv* is null (**0**) it initializes *r_out* to null and otherwise to refer to *out* (using the indirect access *id_srv′.out*). Overall, the co-instantiation of a *ClientT* and *ServerT* modules will initialize the value of *r_out′* of the client thread to refer to the *out* variable of the server thread and the value of *r_in′* of the server thread to refer to the *in* variable of the client thread. To avoid infinite instantaneous creation, we disallow instantiation of new modules in initial commands.

A state of a dynamic reactive system carries the unique identifiers and variable valuations of instantiates modules. In the initial state, the only instantiated module is the initial one. In every subsequent round, the state variables are updated according to the specified commands, which may, in addition, instantiate new modules. Initialization of instantiated modules depends on transferred parameter values.

3 Dynamic Discrete Systems

Dynamic reactive modules is a modelling language. In this section, we introduce a semantic model, which is interesting in its own right, to give a formal semantics to dynamic reactive modules. We extend fair discrete systems (FDS) [14], which are "bare bones" transition systems including a set of variables and prescribed initial states and transition relations by logical formulas. The simplicity of FDS and their resemblance to BDDs, have made them a convenient tool for defining symbolic transition systems. FDSs support composition but not encapsulation and here we extend them with dynamicity. We then use this new model to define the semantics of dynamic reactive modules.

Our template for creating a process is a *simple dynamic discrete system* (SDDS) and the collection of SDDSs is a *dynamic discrete system* (DDS). An SDDS defines a process, its variables, their initializations, and their updates. To create multiple instances of an SDDS, each instantiation has a unique *identifier*. Accordingly, when instantiating an SDDS we allocate all its variables with the same identifier. As mentioned, DDS do not support encapsulation. Thus, the model has a set of variables coming from multiple SDDS and possibly multiple instantiations of the same SDDS. We prefix the variables of the SDDS with the identifier of its instantiation, thus making the variables unique. For that we will use *identified* variables. For example, if the definition includes the variable n, the instantiation with identifier i uses the variable $i.n$.

Let $\mathcal{N}$ be the universal set of variables such that **id** $\in \mathcal{N}$. The variables in $\mathcal{N}$ are going to be used in the definitions of SDDSs. Let $\mathcal{I}$ be the universal set of identifiers. The identifiers in $\mathcal{I}$ are going to be used to identify instances of SDDSs. Apart from the universal set of variables, all sets of variables $N \subset \mathcal{N}$ are *finite*. For example, in Figure 2, the set $\{id\_cl, new\_cl, id\_srv\}$ is the set of variables for the server. When an SDDS is instantiated, all its variables are going to be prefixed with an identifier i. For that, given a set of variables X, let $i.X$ denote $\{i.n \mid n \in X\}$. When a server thread, from Figure 2, is instantiated with identifier i, the set of identified variables for that instance is $\{i.out, i.r\_in\}$. So when there are multiple active instances of server thread, e.g., with identifiers i and j, their variables can be distinguished, e.g., as $i.out$ and $j.out$. Variables range either over some finite domain (for the sake of concreteness we use *Booleans* denoted $\mathbb{B}$) or over the set $\mathcal{R} = \mathcal{I} \cup (\mathcal{I} \times \mathcal{N}) \cup \{\mathbf{0}\}$ of *references*. A reference is either the identifier of an instantiated SDDS ($i \in \mathcal{I}$), an identified variable ($i.n \in \mathcal{I} \times \mathcal{N}$), or null ($\mathbf{0}$). We denote by $\mathsf{type}(x)$ the *type* of a variable x. For a variable $x \in X$, we denote by x' its *primed* copy, and naturally extend this notation for a set X.

Let $\mathfrak{X} = \mathcal{I} \times \mathcal{N}$ be the universal set of *identified variables*. A *state* s is a valuation function $s : \mathfrak{X} \to \mathcal{R} \cup \mathbb{B} \cup \{\bot\}$ such that for every $i \in \mathcal{I}$ we have $s(i.\mathbf{id}) \in \{\bot, i\}$. That is, a state interprets *all* variables as either Booleans, identifiers, identified variables, or $\bot$. The **id** of i is either i or $\bot$. The value $\bot$ is used for two purposes. First, if $s(i.n) = \bot$, then $i.n$ is not allocated in s. Second, $\bot$ is used as a third value in 3-valued propositional logic. This allows to formally represent impossible dereferencing. The type of a variable x in state s is denoted as $\mathsf{type}_s(x)$. A variable x such that $s(x) \in \mathbb{B}$ is said to be Boolean, denoted $\mathsf{type}_s(x) = \mathbb{B}$. A variable x such that $s(x) \in \mathcal{R}$ is said to be a reference, denoted $\mathsf{type}_s(x) = \mathbb{R}$. Let $s_\bot$ denote the state such that $s(x) = \bot$ for every $x \in \mathfrak{X}$. An identified variable $x \in \mathfrak{X}$ is *inactive* in state s if $s(x) = \bot$ and *active* otherwise. If $i.$**id** is inactive in state s then for every variable n we have $i.n$ is inactive in s. An identifier i is *inactive* in state s if $i.$**id** is inactive in s and *active* otherwise.

Reference variables require us to be able to take the reference of a variable and to dereference. Through a reference variable that holds an identifier of an SDDS, we need

$$
\begin{aligned}
(s,t)(x) &= \begin{cases} s(x) & \text{if } x \in \mathfrak{X} \\ t(x) & \text{if } x \in \mathfrak{X}' \end{cases} \\
(s,t)(x.m) &= \begin{cases} \bot & \text{if } (s,t)(x) \notin \mathcal{I} \\ (s,t)((s,t)(x).m) & \text{if } (s,t)(x) \in \mathcal{I} \text{ and } x \in \mathfrak{X} \\ (s,t)((s,t)(x).m') & \text{if } (s,t)(x) \in \mathcal{I} \text{ and } x \in \mathfrak{X}' \end{cases} \\
(s,t)(\mathsf{ref}(i.n)) &= (i,n) \\
(s,t)(\mathsf{ref}(i.n')) &= (i,n) \\
(s,t)(\mathsf{ref}(x.m)) &= \begin{cases} \bot & \text{if } (s,t)(x) \notin \mathcal{I} \\ ((s,t)(x).m) & \text{if } (s,t)(x) \in \mathcal{I} \end{cases} \\
(s,t)(\mathsf{deref}(x)) &= \begin{cases} \bot & \text{if } (s,t)(x) \notin \mathcal{I} \times \mathcal{N} \\ (s,t)((s,t)(x)) & \text{if } (s,t)(x) \in \mathcal{I} \times \mathcal{N} \end{cases} \\
(s,t)(\mathsf{deref}(x.m)) &= \begin{cases} \bot & \text{if } (s,t)(x) \notin \mathcal{I} \\ (s,t)(\mathsf{deref}((s,t)(x).m)) & \text{if } (s,t)(x) \in \mathcal{I} \end{cases} \\
(s,t)(\mathsf{deref}(\tau)) &= \begin{cases} \bot & \text{if } (s,t)(\tau) \notin \mathcal{I} \times \mathcal{N} \\ (s,t)((s,t)(\tau)) & \text{if } (s,t)(\tau) \in \mathcal{I} \times \mathcal{N} \end{cases}
\end{aligned}
$$

Fig. 3. Evaluation of terms on a state pair (s,t)

to be able to access the variables of this SDDS. Given a set of variables $X \subset \mathcal{N}$, we define *indirect accesses* (π), *terms* (τ) and *expressions* (φ) over X as follows.

$$
\begin{aligned}
\pi &::= x \in X \cup X' \mid x.m \text{ for } x \in X \cup X', m \in \mathcal{N} \\
\tau &::= \pi \mid \mathsf{ref}(\pi) \mid \mathsf{deref}(\tau) \\
\varphi &::= \tau \mid \tau = \tau \mid \tau = \mathbf{0} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi
\end{aligned}
\tag{1}
$$

We give values to both x and x' by interpreting indirect accesses, terms, and expressions over *pairs* of states, which stand for current and next values.

In Figure 2, the term $\mathsf{ref}(\mathit{id\_srv.out})$ in *ClientT*'s atom *r_out*, transforms to the term $\mathsf{ref}(\mathit{id\_srv}'.\mathit{out})$ after substitution of the variable *id_srv*$'$ for the parameter *id_srv* during instantiation of the client. The term $\mathsf{ref}(\mathit{id\_srv}'.\mathit{out})$ indirectly accesses the variable *out* of the server thread instance whose identifier is stored in the variable *id_srv*.

An expression that does not use primed variables is *current*. An expression that does not use unprimed variables is *next*. Thus, expressions are logical characterization of possible assignments to variables. As usual, using two copies of a variable x and x' we can use expressions to define the relations between current and next assignments. We assume familiarity Kleene's strongest regular 3-valued propositional logic over the set $\mathbf{3} = \{\mathtt{t}, \bot, \mathtt{f}\}$ [15]. For example, $\mathtt{f} \wedge \bot = \mathtt{f}$, $\bot \vee \mathit{true} = \mathit{true}$, and $\neg\bot = \bot$.

Given two states s and t, we denote by (s,t) the mapping $(s,t) : \mathfrak{X} \cup \mathfrak{X}' \rightarrow \mathcal{R} \cup \mathbb{B} \cup \{\bot\}$ such that for every $x \in \mathfrak{X}$ we have $(s,t)(x) = s(x)$ and $(s,t)(x') = t(x)$. The definition of $\mathsf{type}_{(s,t)}(x)$ is extended as expected.

The value of a term τ in pair (s,t) is defined in Figure 3. For example, consider the value of the indirect access $x.m$. We start by evaluating $(s,t)(x)$. If $(s,t)(x)$ is not an identifier, then clearly we cannot access its m variable and return $\bot$. Otherwise, $(s,t)(x)$ is an identifier i. If x is unprimed, then we access the value of $i.m$ in s. Otherwise, we access the value of $i.m$ in t. Other evaluations of the indirect access are similar. Consider the value of $\mathsf{deref}(x)$ for a variable x. We first evaluate $(s,t)(x)$ and ensure that it indeed holds an identified variable $(i.n)$. Then we check the value of that variable $(s,t)(i.n)$. The value of an expression φ in pair (s,t) denoted $(s,t)(\varphi)$, is defined as follows. For a term τ we have already defined $(s,t)(\tau)$. We define $(s,t)(\tau_1 = \tau_2)$

to be $\mathtt{t}$ if $(s,t)(\tau_1) = (s,t)(\tau_2)$ and $\mathtt{f}$ otherwise. For instance, in Figure 2, the expression $r\_out' = \mathsf{ref}(id\_srv'.out)$ (with substitution of *id_srv* for parameter *id_srv*) is satisfied, if t interprets *r_out* as $i.out$, where $id\_srv = i$. Similarly, $(s,t)(\tau = \mathbf{0})$ is $\mathtt{t}$ iff $(s,t)(\tau) = \mathbf{0}$. The definition of $(s,t)(\varphi)$ for expressions using the Boolean connectives $\wedge$, $\vee$, and $\neg$ is as expected, where every $\alpha \in \mathcal{R}$ is treated like $\bot$. Finally, a pair (s,t) satisfies an expression φ if $(s,t)(\varphi) = \mathtt{t}$. Note that the definitions in Figure 3 takes into account both the current and next versions of variables. Thus, it is defined over a pair of states (s,t). The definition for current expressions and single state is a specialization, where we care only about the state s on the left.

3.1 Definitions

A DDS is $\mathcal{K} = \langle \mathbb{D}, \mathcal{D}_0 \rangle$, where $\mathbb{D}$ is a finite set of SDDS and $\mathcal{D}_0 \in \mathbb{D}$ is an initial SDDS. An SDDS is a tuple $\mathcal{D} = \langle X, Y, \Theta, \rho \rangle$ consisting of the following components.

- $X \subseteq \mathcal{X}$ is the finite set of variables of $\mathcal{D}$ and Y is the finite set of its parameters.
- Θ: The initial condition is a next expression over $X \cup Y$ characterizing all states in which $\mathcal{D}$ can be created. These are the initial states of $\mathcal{D}$ at the time of creation.
- ρ: The transition relation. We extend expressions in Equation (1) to *creation expressions*. Given $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$, for $i \in \{1,2\}$, a *creation* of $\mathcal{D}_2$ by $\mathcal{D}_1$ is either $n_1' = \mathsf{new}\ \mathcal{D}_2(\tau_1, \ldots, \tau_l)$ or $(n_1', \ldots, n_k') = (\mathsf{new}\ \mathcal{D}_2(\tau_1, \ldots, \tau_l)).[m_1, \ldots, m_k]$, where $\{n_1, \ldots, n_k\} \subseteq X_1$, $\{m_1, \ldots, m_k\} \subseteq X_2$, $\{y_1, \ldots, y_l\} = Y_2$, and $\tau_1, \ldots,$ τ_l are terms over X_1. Intuitively, the new expression returns the identifier i of the newly created module. Thus, the first new command stores the identifier of the newly created SDDS in n_1'. The second new command uses the multiple assignment $(n_1', \ldots, n_k') = i.[m_1, \ldots, m_k]$ and updates the n_j variables of $\mathcal{D}_1$ to the newly created variables m_j of $\mathcal{D}_2$. In both cases, the parameters of $\mathcal{D}_2$ are initialized with the values of the expressions τ_j passed by $\mathcal{D}_1$. Let $C(\mathbb{D}, \mathcal{D})$ be the set of all possible creations of SDDS $\mathcal{D}'$ by $\mathcal{D}$ such that $\mathcal{D}' \in \mathbb{D}$. Let φ denote the set of expressions over X, then creation expressions by $\mathcal{D}$ in the context of $\mathbb{D}$ are:
$$\varphi_c \ ::= \ \varphi \mid c \in C(\mathbb{D}, \mathcal{D}) \mid \varphi_c \wedge \varphi_c \mid \varphi_c \vee \varphi_c,$$
The transition relation ρ is a creation expression by $\mathcal{D}$ in the context of $\mathbb{D}$.

We now define the possible traces of an SDDS. This is a sequence of states such that every pair of adjacent states satisfy the transition of the SDDS. However, as creation is involved, the transition relation needs to be augmented with the rules that govern the newly created variables. For that, we add to traces the maps of creations that are performed along them and the update of the transition that governs these new creations.

Given a transition relation ρ, let $\mathsf{subnew}(\rho)$ be the subformulas of ρ that are creations. A *creation-map* m for ρ is a partial one-to-one function $m : \mathsf{subnew}(\rho) \to \mathcal{I}$. A creation map tells us which creations are actually invoked (those for which m is defined) and what is the identifier of the instantiated process.

A pair of states (s,t) satisfies a transition ρ with creation map m and producing transition $\tilde{\rho}$, denoted $(s,t) \models (\rho, m, \tilde{\rho})$ if all the following conditions hold.

1. For every creation $c \in \mathsf{subnew}(\rho)$ of $\mathcal{D}_1$, if $m(c) = i$ then i is *inactive* in s and for every $n \in X_1$ we have $i.n$ is *active* in t. That is, instantiated SDDS are activated.
2. For every $i.n$ such that i is *active* in s we have $i.n$ is *active* in s iff it is *active* in t and $\mathsf{type}_s(i.n) = \mathsf{type}_t(i.n)$. That is, existing instantiations do not change.

3. For every creation $c \in \mathsf{subnew}(\rho)$ of $\mathcal{D}_1$, where c is
$$(i.n_1, \ldots, i.n_k) = (\mathsf{new}\ \mathcal{D}_1(\tau_1, \ldots, \tau_l)).[m_1, \ldots, m_k],$$
if $m(c) = j$ then all the following hold:
   (a) $(s, t) \models \Theta_1[j][\tau_1/y_1, \ldots, \tau_l/y_l]$, where $\Theta_1[j][\tau_1/y_1, \ldots, \tau_l/y_l]$ is obtained from Θ_1 by replacing every mention of n by $j.n$ and every input y_b' by τ_b.
   (b) For every $1 \leq o \leq k$ we have $(s, t)(i.n_o') = (j.m_o)$,

   That is, the pair (s, t) satisfies the initialization of the instantiated SDDS using the inputs sent by the creating SDDS. Furthermore, reference variables of the creating SDDS now reference the newly created variables.
4. $(s, t) \models \overline{\rho}$, where $\overline{\rho}$ is obtained from ρ by replacing the creation sub-formulas $c \in \mathsf{subnew}(\rho)$ such that $m(c) = i$ by $\mathtt{t}$, and $c \in \mathsf{subnew}(\rho)$ such that $m(c)$ is undefined by $\mathtt{f}$.

   That is, the pair (s, t) satisfies the transition relation. We ensure that enough SDDS were instantiated by evaluating those that were not instantiated as $\mathtt{f}$.
5. $\tilde{\rho} = \rho \wedge \bigwedge_{\{c \mid m(c)=i\}} \rho_c[m(c)]$, where $\rho_c[m(c)]$ is the transition relation of the SDDS created by c with every mention of n replaced by $m(c).n$.

   That is, we update the transition relation with the rules that govern the updates of the newly created SDDSs.

We are now ready to define traces of DDS. Traces are going to include the states, transition relations, and creation maps that match them. Consider a finite or infinite sequence $\sigma = s_0, \rho_0, m_0, s_1, \rho_1, m_1 \ldots$, where for every $j \geq 0$ we have s_j is a state, ρ_j is a creation expression, and m_j is a creation map for ρ_j. If σ is infinite we write $|\sigma| = \omega$. If σ is finite it ends in an expression ρ_{n-1} and we write $|\sigma| = n$. A sequence σ is a *creation trace* for an SDDS $\mathcal{D} = \langle X, Y, \Theta, \rho \rangle$ with identifier i at time $0 \leq t < |\sigma|$ and valuations $v_1, \ldots, v_l$ for $\{y_1, \ldots, y_l\} = Y$ if all the following hold.

1. For every $t' < t$ we have $\rho_{t'} = \mathtt{t}$ and $m_{t'}$ is the empty map. Furthermore, $\rho_t = \rho[i]$.
2. If $s_{-1} = s_\perp$ then $(s_{t-1}, s_t) \models \Theta[i][v_1/y_1, \ldots, v_k/y_k]$.
3. For every $0 \leq t' < |\sigma| - 1$ we have $(s_{t'}, s_{t'+1}) \models (\rho_{t'}, m_{t'}, \rho_{t'+1})$.
4. The identifier i is *inactive* in s_{t-1} and for every $n \in X$, $i.n$ is *active* in s_t.

We write in short $(\sigma, i, t, v_1, \ldots, v_k)$ is a CT of $\mathcal{D}$.

Intuitively, the SDDS $\mathcal{D}$ is created at time t by initializing its inputs to v_1, …, v_l. All the variables of $\mathcal{D}$ (and possibly more) identified by i become active in t; And the (mutable according to the creation maps) transition of $\mathcal{D}$ holds on the entire sequence. Prior to the creation of $\mathcal{D}$ the transitions are $\mathtt{t}$ and accordingly creation maps are empty.

A finite CT σ ends in a *deadlock* if it cannot be extended to a longer CT. Intuitively, there can be two reasons for deadlocks. First, a contradiction in the transition, such as requiring that $x = y$ and $y = \neg x$. Obviously, this can be made more interesting by accessing x and y through their references. Second, the option to dereference null, dereference a Boolean variable, or trying to access a wrong name through an identifier.

3.2 Properties

Here we define parallel composition, refinement, and replacement. Parallel composition allows to create models of increasing complexity from smaller parts. It enables static communication through external variables. Refinement says when one DDS is more general than another. Then, replacement is the action of replacing creation of abstract SDDS by SDDS that refine it. Composition corresponds to intersection of traces and refinement to inclusion of traces (both with appropriate adjustments).

We start with parallel composition, which essentially allows to "run" two SDDS side by side. Consider a set of SDDS $\mathbb{D}$ and two SDDS $\mathcal{D}_i \in \mathbb{D}$, where $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$ for $i \in \{1, 2\}$. Then, $\mathcal{D}_{1\|2}$ is the SDDS $\langle X_{1\|2}, Y_{1\|2}, \Theta_{1\|2}, \rho_{1\|2} \rangle$ where $X_{1\|2} = X_1 \cup X_2$, $Y_{1\|2} = Y_1 \cup Y_2$, $\Theta_{1\|2} = \Theta_1 \wedge \Theta_2$, and $\rho_{1\|2} = \rho_1 \wedge \rho_2$.

Consider a CT $\mu = (\sigma, i, t, v_1, \ldots, v_k)$, where $\sigma = s_0, \rho_0, m_1, \ldots$. We say that CTs $(\sigma^1, i, t, v_{j_1}, \ldots, v_{j_{l_1}})$ and $(\sigma^2, i, t, v_{p_1}, \ldots, v_{p_{l_2}})$ *partition* μ if $\{v_{j_1}, \ldots, v_{j_{l_1}}\} \cup \{v_{p_1}, \ldots, v_{p_{l_2}}\} = \{v_1, \ldots, v_k\}$ and for every $t \geq 0$ we have $s_t = s^1_t = s^2_t$, $\rho_t = \rho^1_t \wedge \rho^2_t$, and m_t is the disjoint union of m^1_t and m^2_t.

Theorem 1. *A creation trace μ is a creation trace of $\mathcal{D}_{1\|2}$ iff there exist μ_1 and μ_2 that partition μ such that μ_i is a creation trace of $\mathcal{D}_i$, for $i \in \{1, 2\}$.*

We define refinement as having the same set of traces with specialized creations. Consider a set of SDDS $\mathbb{D}$ and two SDDS $\mathcal{D}_i \in \mathbb{D}$, where $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$ for $i \in \{1, 2\}$. Consider two CTs $\mu_1 = (\sigma_1, i, t, v_{j_1}, \ldots, v_{j_{l_1}})$ and $\mu_2 = (\sigma_2, i, t, v_1, \ldots, v_{l_2})$, where $\sigma_i = s^i_0, \rho^i_0, m^i_0, \ldots$, for $i \in \{1, 2\}$. We say that μ_2 *specializes* μ_1 if for every $t' \geq 0$ we have $\rho^2_{t'} = \rho^1_{t'} \wedge \rho^*_{t'}$ and $m^2_{t'}$ is the disjoint union of $m^1_{t'}$ and $m^*_{t'}$ for some $\rho^*_{t'}$ and $m^*_{t'}$. We say that $\mathcal{D}_2$ *refines* $\mathcal{D}_1$, denoted $\mathcal{D}_2 \preceq \mathcal{D}_1$, if $X_2 \supseteq X_1$, $Y_2 \supseteq Y_1$, and every creation trace μ_2 of $\mathcal{D}_2$ is a specialization of some CT μ_1 of $\mathcal{D}_1$.

Theorem 2. *The refinement relation $\preceq$ is a preorder.*

In order to replace the creation of $\mathcal{D}_2$ by $\mathcal{D}_3$ we have to ensure that $\mathcal{D}_3$ does "more" than $\mathcal{D}_2$. Consider a set of SDDS $\mathbb{D}$. We say that transition relation ρ_2 *refines* transition relation ρ_1 if $\rho_2 = \overline{\rho}_1 \wedge \rho^*$ for some ρ^*, where $\overline{\rho}_1$ is obtained from ρ_1 by replacing every creation $(\ldots, n_k) = (\mathsf{new}\, \mathcal{D}_3(\ldots, \tau_l)).[\ldots, m_k]$ in ρ_1 by creation

$$\begin{aligned}(\ldots, n_k, n_{k+1}, \ldots, n_{k+r}) = \\ &(\mathsf{new}\, \mathcal{D}_4(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+b}).[\ldots, m_k, m_{k+1}, \ldots, m_{k+r}],\end{aligned}$$

where $\mathcal{D}_4$ refines $\mathcal{D}_3$. As for every SDDS $\mathcal{D} \preceq \mathcal{D}$, some creations can remain unchanged.

Theorem 3. *For all* SDDS *$\mathcal{D}_1$ and $\mathcal{D}_2$, if the initial condition Θ_2 refines the initial condition Θ_1 and the transition relation ρ_2 refines the transition relation ρ_1, then $\mathcal{D}_2 \preceq \mathcal{D}_1$.*

Theorem 4. *For all* SDDS *$\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$, we have $\mathcal{D}_{1\|2} \preceq \mathcal{D}_1$ and $\mathcal{D}_{(1\|2)\|3} = \mathcal{D}_{1\|(2\|3)}$.*

Finally, when an SDDS refines another, we can replace creations of the second by creations of the first. Consider SDDS $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$, for $i \in \{1, 2, 3\}$, where $\mathcal{D}_3 \preceq \mathcal{D}_2$. The SDDS $\mathcal{D}_{1[3/2]}$ is given by $\langle X_1, Y_1, \Theta_1, \overline{\rho}_1 \rangle$ where $\overline{\rho}_1$ is obtained from ρ_1 by replacing every creation $(\ldots, n_k) = (\mathsf{new}\, \mathcal{D}_2(\ldots, \tau_l)).[\ldots, m_k]$ by a creation

$$\begin{aligned}(\ldots, n_k, n_{k+1}, \ldots, n_{k+r}) = \\ &(\mathsf{new}\, \mathcal{D}_3(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+b})).[\ldots, m_k, m_{k+1}, \ldots, m_{k+r}].\end{aligned}$$

Theorem 5. *For all* SDDS *$\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ where $\mathcal{D}_3 \preceq \mathcal{D}_2$, we have $\mathcal{D}_{1[3/2]} \preceq \mathcal{D}_1$.*

4 Formal Dynamic Reactive Modules

We give the formal definition of dynamic reactive modules. As mentioned, a module class is the recipe of behavior that may be instantiated multiple times. A dynamic reactive system is a collection of reactive-module classes, where one is identified as initial.

A *dynamic reactive system* $M = (\mathcal{S}, S_0)$ consists of a finite set of *module classes* $\mathcal{S}$ and an *initial* class $S_0 \in \mathcal{S}$. A class $S = (X, Y, \mathcal{A})$ consists of a finite set X of *typed* variables, a finite set Y of *typed parameters* and a finite set $\mathcal{A}$ of *atoms*. The set X is partitioned into two sets: (1) a set *ctr* of *controlled* variables and (3) a set *ext* of *external* variables. The set of atoms $\mathcal{A}$ partitions further the controlled variables, where each atom $A \in \mathcal{A}$ controls the initialization and the updates of a subset $\mathit{ctr}(A) \subseteq \mathit{ctr}$. Note that we allow $\mathcal{A}$ to be *empty*, in which case all the variables in X can have unconstrained behavior. If $\mathcal{A}$ is not empty, every atom $A \in \mathcal{A}$ consists of two finite sets $\mathit{Init}(A)$ and $\mathit{Update}(A)$ of *guarded commands* γ that define rules for initializing and updating variables in $\mathit{ctr}(A)$, respectively. We distinguish between *initial* and *update* guarded commands. A guarded command $\gamma \in A$ is a pair $(p_\gamma, \mathit{Act}_\gamma)$, where p_γ is a *guard*, i.e. a next expression φ over $X \cup Y$ if γ is initial, or an expression over X if γ is update, and Act_γ consists of a finite set of *actions* that can have the following form: (1) $n' := \varphi;$, where $n \in \mathit{ctr}(A)$ and φ is a future expression over $X \cup Y$ if γ is initial, or an expression over X if γ is update, or (2) $(n'_1, \ldots, n'_k) := (\mathsf{new}\ S'(\tau_1, \ldots, \tau_l)).[m_1, \ldots, m_k];$, where $S' \in \mathcal{S}$, for all $i \in [1, k]$, $n_i \in \mathit{ctr}(A)$ and $m_i \in X(S')$, $\mathit{param}(S') = \{y_1, \ldots, y_l\}$ and for all $i \in [1, l]$, we have τ_i is a term over X. A guarded command γ is said to be *creation-free* if Act_γ contains no creation action. We require that for all classes $S \in \mathcal{S}$, all atoms $A \in \mathcal{A}(S)$, the set $\mathit{Init}(A)$ contains only creation-free guarded commands.

Renaming avoids conflicts when statically creating different instances of a class.

Definition 1 (Class Renaming). *For a class S with $X = \{n_1, \ldots, n_k\}$ then $S[m_1 = n_1, \ldots, m_k = n_k]$ is the class that results from S by replacing n_j by m_j for every j.*

The composition operation between two classes results in a single class whose behavior captures the interaction between two classes. Two classes S_1 and S_2 are composable if they do not share controlled variables. Parallel composition encodes the static and hard-coded input/output connections between S_1 and S_2. We naturally extend composition from classes to systems M_1 and M_2.

Definition 2 (Parallel Composition). *Let $S_1 = (X_1, \mathcal{A}_1)$ and $S_2 = (X_2, \mathcal{A}_2)$ be two classes. We say that S_1 and S_2 are* composable *if* $\mathrm{ctr}(S_1) \cap \mathrm{ctr}(S_2) = \emptyset$*. Given two composable classes S_1 and S_2, we denote by $S = S_1 \mid\mid_S S_2$ the* parallel composition *of S_1 and S_2, where $S = (X, \mathcal{A})$, such that $X(S)$ is partitioned into* $\mathrm{ctr}(S) = \mathrm{ctr}(S_1) \cup \mathrm{ctr}(S_2)$, $\mathrm{ext}(S) = (\mathrm{ext}(S_1) \cup \mathrm{ext}(S_2)) \setminus \mathrm{ctr}(S)$ *and* $\mathrm{param}(S) = \mathrm{param}(S_1) \cup \mathrm{param}(S_2)$ *and $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$.*

Let $M_1 = (\mathcal{S}_1, S_1^0)$ and $M_2 = (\mathcal{S}_2, S_2^0)$ be two dynamic reactive systems. We say that M_1 and M_2 are composable *if $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ and S_1^0 and S_2^0 are composable. Given two composable systems M_1 and M_2, we define their* parallel composition*, denoted by $M = M_1 \mid\mid M_2$ as the system $M = (\mathcal{S}, S^0)$, such that $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{S^0\}$ and $S^0 = S_1^0 \mid\mid_S S_2^0$.*

We define the *extending* operator between classes S_1 and S_2 to capture specialization at the syntactic level. Informally, a class S_1 extends S_2 if S_1 and S_2 have the same updates for the joint controlled variables, but S_1 is allowed to constrain more control variables than S_2 and can read more variables (external and input) from its environment.

Definition 3 (Extending Classes). *Let S_1 and S_2 be two classes. We say that S_1* extends *S_2, denoted by $S_1 \sqsubseteq S_2$ if $\mathrm{ctr}(S_2) \subseteq \mathrm{ctr}(S_1)$, $\mathrm{ext}(S_2) \subseteq \mathrm{ext}(S_1)$, $Y(S_2) \subseteq Y(S_1)$ and $\mathcal{A}(S_2) \subseteq \mathcal{A}(S_1)$.*

Definition 4 (Replacement of Classes). *Let S_1 and S_2 be two classes. We say that S_2 is* replaceable *by S_1 if $S_1 \sqsubseteq S_2$.*

Let S_1, S_2 and S_3 be three classes such that $S_3 \sqsubseteq S_2$. We denote by $S_1[S_3/S_2]$ the replacement *of S_2 by S_3 in S_1, that consists in replacing every occurrence of a creation $(n'_1, \ldots, n'_k) := (\textsf{new}\, S_2(\ldots, \tau_l)).[m_1, \ldots, m_k];$ in S_1 by a creation $(n'_1, \ldots, n'_k) := (\textsf{new}\, S_3(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+q})).[m_1, \ldots, m_k];$.*

We extend this operator to systems, and given two systems $M_A = (\mathcal{S}_A, S^0_A)$ and $M_B = (\mathcal{S}_B, S^0_B)$ and two classes S_2 and S_3 such that $S_2 \in \mathcal{S}_A$ and $S_3 \sqsubseteq S_2$, we say that M_B replaces *S_2 by S_3 in M_A, denoted by $M_B = M_A[S_3/S_2]$, if the following conditions hold: (1) $\mathcal{S}_B = \mathcal{S}_A \cup \{S_3\}$; (2) if $S^0_A = S_2$, then $S^0_B = S_3$, and $S^0_B = S^0_A$ otherwise; and (3) every S in $\mathcal{S}(M_A)$ is replaced by $S[S_3/S_2]$ in M_B.*

We define the semantics of a reactive dynamic system M in terms of an associated dynamic discrete system $[\![M]\!]$. We now formalize the translation from M to $[\![M]\!]$.

Definition 5 (Semantics: from DRM to DDS). *Let $M = (\mathcal{S}, S^0)$ be a dynamic reactive system. Then, its associated DDS is $[\![M]\!] = \langle \mathbb{D}, \mathcal{D}^0 \rangle$, where $\mathbb{D} = \bigcup_{S \in \mathcal{S}} \mu(S)$ and $\mathcal{D}^0 = \mu(S^0)$ and for a given S, $\mu(S) = \langle X_S, Y_S, \theta_S, \rho_S \rangle$, such that*

- *$X_S = X(S(M))$ and $Y_S = Y(S(M))$*
- *θ_S is the expression $\bigwedge_{A \in \mathcal{A}(S(M))} \bigvee_{\gamma \in \mathrm{Init}(A)} (p_\gamma \rightarrow (\bigwedge_{\alpha \in \mathrm{Act}_\gamma} e_\alpha))$, where e_α is the expression $n' = \varphi$ obtained from the assignment action $\alpha = (n' := \varphi)$*
- *ρ_S is the expression $\bigwedge_{A \in \mathcal{A}(S(M))} \bigvee_{\gamma \in \mathrm{Update}(A)} (p_\gamma \rightarrow (\bigwedge_{\alpha \in \mathrm{Act}_\gamma} e_\alpha))$, where e_α is either the expression $n' = e$ if $\alpha = (n' = e)$ or the creation $(n'_1, \ldots, n'_k) = (\textsf{new}\, \mathcal{D}_i(\tau_1, \ldots, \tau_l)).[m_1, \ldots, m_k]$ if α is the creation action $(n'_1, \ldots, n'_k) := (\textsf{new}\, S_i(\tau_1, \ldots, \tau_l).[m_1, \ldots, m_k]$.*

The following theorem establishes some derived properties from the operations on modules and the properties shown in Section 3.2

Theorem 6. *Given three classes S_1, S_2, and S_3, we have*
1. *if $S_2 \sqsubseteq S_1$, then $\mu(S_2) \preceq \mu(S_1)$;*
2. *if S_1 and S_2 are composable classes, then $\mu(S_1 \mid\mid_S S_2) \preceq \mu(S_1)$;*
3. *if $S_2 \sqsubseteq S_3$, then $\mu(S_1[S_3 \backslash S_2]) \preceq S_1$.*

Biological Example: As another example, shown in Figure 4, we model a simple system *CellModule* of cells (*Cell* class) arranged in a row. Cells can divide at arbitrary times. This is modeled by creation of a new *Cell* instance by an existing one. A newly created cell always appears to the right of its parent. The parent then updates its right neighbor by updating its variable *right* to refer to its daughter cell. Similarly, the daughter cell updates its left neighbor by updating its variable *left* to refer to the parent cell's

$$\textbf{system}\ \mathit{CellModule} = \langle\{\mathit{Cell}\}, \mathit{Cell}\rangle$$

```
class Cell
control create, left, prev_l, right : $\mathcal{R}$
param pid : $\mathcal{R}$
atom create
  init
    [] $true \rightarrow create' := \mathbf{0}$
  update
    [] $true \rightarrow create' :=$ new $Cell(\mathbf{id})$
    [] $true \rightarrow$
atom prev_l
  init
    [] $true \rightarrow prev\_l' := \mathbf{0}$
  update
    [] $true \rightarrow prev\_l' := left;$
atom right
  init
    [] $pid' = \mathbf{0} \rightarrow right' := \mathbf{0}$
    [] $pid' \neq \mathbf{0} \rightarrow right' := pid'.right$
  update
    [] $create \neq create' \rightarrow right' := create'$
atom left
  init
    [] $pid' \neq \mathbf{0} \rightarrow left' := pid'$
    [] $pid' = \mathbf{0} \rightarrow left' := \mathbf{0}$
  update
    [] $(left.create) \neq (left\_p'.create) \rightarrow$
                                    $left' := left\_p'.create$
```

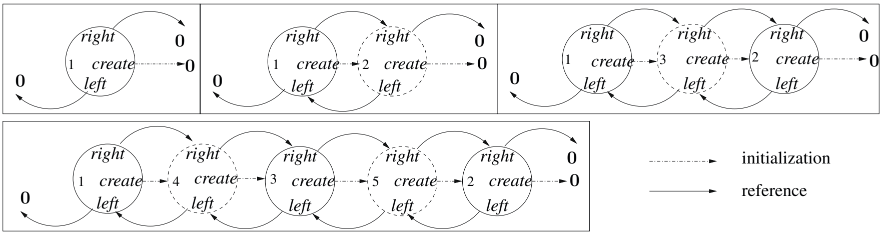


Fig. 4. Dynamic reactive system *CellModule* and dividing cells

id (passed in the parameter *pid*). The cell to the right of the parent cell in the current round, updates its left neighbor in the next round by updating variable *left* to refer to the **id** of the new child cell that appeared on its left. The system runs creating new cells non-deterministically, updating the cell-cell communication pattern with each creation.

5 Conclusions and Future Work

We introduced here dynamic reactive modules, a formalism for modeling dynamic state-transition systems communicating via shared variables. Our formalism supports the three basic features of programming languages: composition, encapsulation, and dynamicity. Previous formalisms supported only the first two and by adding references and creation we achieve dynamicity. The resulting formalism supports instantiation of new “active” variables and reconfiguration of communication.

The resulting formalism is quite powerful and it is clear that many questions, such as deadlock freedom, reachability, and model checking, are going to be undecidable. As dynamicity has been generally missing from state-transition formalisms communicating via shared variables, we hope that this formalism will motivate further research into its modeling capacity and the availability of analysis techniques for it. We state a few obvious such directions. We are interested in techniques from software model checking that could be adapted for its analysis. Similarly, pointer analysis, techniques for understanding the structure of the heap, and static analysis in general, could be applied in this context as well. Another interesting direction is the identification of fragments for which such questions are “well behaved”. One very important type of well behavedness is deadlock avoidance. We are searching for simple rules for deadlock avoidance through by combining (a) avoiding cyclic dependencies between variables and (b) reference safety through typing and access protection.

References

1. Alur, R., Grosu, R.: Dynamic Reactive Modules. Tech. Rep. 2004/6, Stony Brook (2004)
2. Alur, R., Henzinger, T.A.: Reactive modules. FMSD 15(1), 7–48 (1999)
3. Attie, P.C., Lynch, N.A.: Dynamic input/output automata, a formal model for dynamic systems. In: PODC, pp. 314–316 (2001)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
5. Colaço, J.-L., Girault, A., Hamon, G., Pouzet, M.: Towards a higher-order synchronous dataflow language. In: EmSoft, pp. 230–239. ACM, New York (2004)
6. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time UML semantics for concurrency and communication in safety-critical applications. SCP 55(1-3), 81–115 (2005)
7. Efroni, S., Harel, D., Cohen, I.R.: Toward rigorous comprehension of biological complexity: Modeling, execution, and visualization of thymic T-cell maturation. Genome. Res. 13(11), 2485–2497 (2003)
8. Efroni, S., Harel, D., Cohen, I.R.: Emergent dynamics of thymocyte development and lineage determination. PLoS Comput. Biol. 3(1), e13 (2007)
9. Fisher, J., Piterman, N., Hajnal, A., Henzinger, T.A.: Predictive modeling of signaling crosstalk during C. elegans vulval development. PLoS Comput. Biol. 3(5), e92 (2007)
10. Fisher, J., Piterman, N., Hubbard, E.J.A., Stern, M.J., Harel, D.: Computational insights into Caenorhabditis elegans vulval development. Proc. Natl. Acad. Sci. 102(6), 1951–1956 (2005)
11. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8(3), 231–274 (1987)
12. Harel, D., Kugler, H.: The rhapsody semantics of statecharts (or, on the executable core of the UML). In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004)
13. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. 5(4), 293–333 (1996)
14. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Inf. Comput. 163(1), 203–243 (2000)
15. Kleene, S.C.: Introduction to Mathematics. North-Holland, Amsterdam (1987)
16. Lynch, N., Tuttle, M.: An introduction to input/output automata. In: Distributed Systems Engineering (1988)
17. Mandel, L., Pouzet, M.: ReactiveML: a reactive extension to ML. In: PPDP, pp. 82–93. ACM, New York (2005)
18. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
19. Milner, R.: The polyadic pi-calculus (abstract). In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630. Springer, Heidelberg (1992)
20. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i & ii. Inf. Comput. 100(1), 1–77 (1992)
21. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University (1981)

Weak Kripke Structures and LTL

Lars Kuhtz[1] and Bernd Finkbeiner[2]

[1] Microsoft Redmond
[2] Saarland University

Abstract. We revisit the complexity of the model checking problem for formulas of linear-time temporal logic (LTL). We show that the classic PSPACE-hardness result is actually limited to a subclass of the Kripke frames, which is characterized by a simple structural condition: the model checking problem is only PSPACE-hard if there exists a strongly connected component with two distinct cycles. If no such component exists, the problem is in coNP. If, additionally, the model checking problem can be decomposed into a polynomial number of finite path checking problems, for example if the frame is a tree or a directed graph with constant depth, or the frame has an SCC graph of constant depth, then the complexity reduces further to NC.

1 Introduction

Model checking, the automatic verification of a finite-state structure against a formula of a temporal logic, is one of the key advances in systems theory over the past decades. Many artifacts in modern computers, including hardware, protocols, and operating system components, can be described as finite-state structures. Model checking algorithms have also found numerous applications beyond computer-aided verification, including XML data bases, planning, and computational biology.

The complexity of the model checking problem has been the subject of intensive investigation. The fundamental result, by Sistla and Clarke in 1985, is that the model checking problem for linear-time temporal logic (LTL) is PSPACE-complete [21]. A first refinement of this result, due to Lichtenstein and Pnueli, separates the complexity in the length of the formula from the complexity in the size of the Kripke structure. It turns out that the problem really is PSPACE-complete only in the size of formula and linear in the size of the Kripke structure. Much of the subsequent work has therefore focused on detailing the complexity with respect to different classes of formulas [21,14,8,4,3].

However, the linear complexity in the size of the Kripke structure does not mean that the impact of the Kripke structure should be neglected. Consider, for example, Kripke structures that consist of a single state. The model checking problem for such Kripke structures corresponds to the problem of evaluating Boolean formulas, which is NC^1-complete [6]. The PSPACE-hardness result relies on the possibility to encode the computations of a Turing machine as paths in the Kripke structure. What happens if the frame of the Kripke structure does

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 419–433, 2011.

not allow for such an encoding? Examples of such restricted frames occur in many different domains.

- **Paths:** The problem of checking whether a given finite path satisfies an LTL formula plays a key role in monitoring and runtime verification (cf. [9]), where individual paths are checked either online, during the execution of the system, or offline, for example based on an error report. Similarly, path checking occurs in testing [1] and in several static verification techniques, notably in Monte-Carlo-based probabilistic verification, where large numbers of randomly generated sample paths are analyzed [22].
- **Trees:** Model checking trees occurs in assertion checking, querying, debugging, and searching in all kinds of parse trees, class hierarchies, thread or process trees, abstract data types, file systems, and XML documents and XML databases (cf. [5]).
- **Restricted loops:** Control and scheduling programs often enter, after some initialization, an infinitely executed static loop [2]. The frame of the Kripke structure thus is a lasso path. If the system proceeds in stages, where each stage consists of the nondeterministically many iterations of a static loop, then the Kripke frame has several loops, but no nested loops.

In this paper, we abstract from the possible LTL formulas and the possible labelings of the Kripke structure, and instead focus entirely on the structure of the frame. A key role in our analysis is played by the *path checking* problem [19], i.e., the model checking problem where the frame is restricted to a single path. We recently showed that, for LTL formulas, path checking is in NC [13]. We generalize this result to Kripke structures for which the model checking problem can be deterministically reduced to a polynomial number of parallel path checking problems: Kripke structures that are trees or directed graphs with constant depth, or that have an SCC graph of constant depth, all have model checking problems in NC.

Our main result is that the separation between Kripke structures with a PSPACE hard model checking problem and Kripke structures with a model checking problem in coNP is a strict dichotomy. The borderline between PSPACE and coNP can in fact be characterized by a simple structural condition: We call a Kripke structure *weak* if there are no two distinct cycles within a single strongly connected component. The model checking problem for weak Kripke structures can be decomposed into the path checking problems for an exponential number of relevant paths. If a Kripke structure is weak then the model checking problem is therefore in coNP; otherwise, the model checking problem is PSPACE hard.

Related Work. The subject of this paper, the model checking problem for restricted classes of Kripke frames, is an extension of the path checking problem, which was introduced as an open problem by Demri and Schnoebelen in [8]. In addition to our recent work [13] on LTL path checking, Markey and Schnoebelen investigate the path checking problem for various extensions and restrictions of LTL [19] and also show that the complexity of the (finite) path checking problem for the μ-calculus is P-hard [20]. Markey and Raskin [18] study the complexity of the model checking problem for restricted sets of paths for extensions of LTL to continuous time.

Another area of investigation that is closely related is the study of the state explosion problem. The state explosion problem occurs in compositional model checking when the Kripke structure is represented as some kind of product Kripke structure. Demri, Laroussinie, and Schnoebelen study the complexity of model checking parameterized by the number of factors of the product Kripke structure [7].

In classical modal logic, systems are defined via frame conditions. Starting with Ladners seminal results in [16] there is a line of research about the complexity of problems for modal logics systems under certain frame conditions (cf. [11,10] for recent results and overview on past work).

2 Computation Paths and Kripke Structures

Linear-time temporal logic reasons about linearly ordered sequences of states, which we call computation paths. In the following we define the semantic framework for the logic: propositions, states, computation paths, Kripke frames, and Kripke structures. Kripke structures symbolically represent sets of computations paths in a compact way. Kripke frames represent the topology of transition relation of a Kripke structure.

Given a set of *atomic propositions* AP. A *state* $s \in 2^{\mathrm{AP}}$ is an evaluation of the atomic propositions in AP. For $p \in \mathrm{AP}$ we say that p holds in s if and only if $p \in s$. We write $s(p)$ to denote the value of p in s with $s(p) = 1$, if p holds in s, and $s(p) = 0$ otherwise. An ordered sequence $\rho = \rho_0, \rho_1, \ldots$ of states is called a *computation path* over AP. The *length of* ρ is denoted by $|\rho|$. If ρ is infinite, we set $|\rho| = \infty$; $i < \infty$ for all $i \in \mathbb{N}$. For a computation path ρ and $0 \leq i < |\rho|$ we write ρ_i for the state at position i; $\rho_{i,j}$, where $0 \leq i \leq j < |\rho|$, denotes the computation path $\rho_i, \rho_{i+1}, \ldots, \rho_j$ of length $|\rho_{i,j}| = j - i + 1$; $\rho_{i,\ldots}$ denotes the suffix of ρ at position i. The empty sequence is denoted ϵ with $|\epsilon| = 0$. We denote concatenation of computation paths as a product and write either $\sigma\rho$ or $\sigma \cdot \rho$ for the concatenation of the computation paths σ and ρ, where σ is finite. For a finite computation path σ we set $\sigma^n = \prod_0^{n-1} \sigma$, $\sigma^* = \left\{\prod_0^{n-1} \sigma \mid n \in \mathbb{N}\right\}$, and $\sigma^\omega = \prod_0^\infty \sigma$. In the context of automata we will treat computation paths over AP as *words* over the *alphabet* $\Sigma = 2^{\mathrm{AP}}$, where a *letter* is a state. The set of all finite words over Σ is denoted as Σ^*. The set of infinite words is denoted as Σ^ω. A *language* over Σ is a subset of $\Sigma^* \cup \Sigma^\omega$. A computation path (or a word) $\rho = \rho_0, \rho_1, \ldots, \rho_n, n \in \mathbb{N}$ canonically defines a (graph theoretic) path. In the following we view ρ as a path whenever adequate.

A *Kripke structure* $\mathcal{K}$ is a four-tuple $\langle K, k_i, R, \lambda\rangle$ where K is a set of vertices, $k_i \subseteq K$ are the initial vertices, $R \subseteq K \times K$ is a transition relation, and $\lambda\colon K \to 2^{\mathrm{AP}}$ is a labeling function on the vertices of $\mathcal{K}$. The directed graph $\langle K, R\rangle$ is called the *frame* of the Kripke structure $\mathcal{K}$. By abuse of notation we sometimes identify a state $k \in K$ with its labeling $\lambda(k)$, where we assume that $\lambda^{-1}(k)$ is determined from the context.

The *SCC graph* of a Kripke frame $\mathcal{F}$ is the graph that is obtained by collapsing every strongly connected component of $\mathcal{F}$ into a single vertex.

The *language of a Kripke structure* $\mathcal{K} = \langle K, k_i, R, \lambda\rangle$, denoted as $\mathrm{lang}(\mathcal{K})$, is the set of (finite and infinite) computation paths

$$\{\lambda(s_0), \lambda(s_1), \cdots \mid s_0 \in k_i, \langle s_i, s_{i+1}\rangle \in R\}$$

for $i \in \mathbb{N}$ with $0 \leq i$ or $0 \leq i < n$ for some $n \in \mathbb{N}$.

A Kripke structure (respectively a Kripke frame) is called *weak*, if there are no two distinct cycles within a single strongly connected component; in other words: all cycles are pairwise disjoint. This implies that the cycles of a weak Kripke strucutre (Kripke frame, respectively) are partially ordered with respect to reachability.

3 Linear-Time Temporal Logic – LTL

We consider linear-time temporal logic (LTL) with the usual finite-path semantics, which includes a weak and a strong version of the Next operator [17]. Let AP be a set of atomic propositions. The *LTL formulas* are defined inductively as follows: every atomic proposition $p \in \mathrm{AP}$ is a formula. If ϕ and ψ are formulas, then so are

$$\neg\phi, \quad \phi \wedge \psi, \quad \phi \vee \psi, \quad \mathrm{X}^{\exists}\,\phi, \quad \mathrm{X}^{\forall}\,\phi, \quad \phi\,\mathrm{U}\,\psi, \quad \text{and} \quad \phi\,\mathrm{R}\,\psi\ .$$

Let $p \in \mathrm{AP}$. We use *true* to abbreviate $p \vee \neg p$ and *false* as an abbreviation for $p \wedge \neg p$. For a formula ϕ we write $\mathrm{G}\,\phi$ to abbreviate $false\,\mathrm{R}\,\phi$ and $\mathrm{F}\,\phi$ as an abbreviation for $true\,\mathrm{U}\,\phi$. The *size of a formula* ϕ is denoted by $|\phi|$.

LTL formulas are evaluated over computation paths over the set of states 2^{AP}. Given an LTL formula ϕ, a nonempty computation path ρ *satisfies* ϕ at position i ($0 \leq i < |\rho|$), denoted by $(\rho, i) \models \phi$, if one of the following holds:

- $\phi \in \mathrm{AP}$ and $\phi \in \rho_i$,
- $\phi = \neg\psi$ and $(\rho, i) \not\models \psi$,
- $\phi = \phi_l \wedge \phi_r$ and $(\rho, i) \models \phi_l$ and $(\rho, i) \models \phi_r$,
- $\phi = \phi_l \vee \phi_r$ and $(\rho, i) \models \phi_l$ or $(\rho, i) \models \phi_r$,
- $\phi = \mathrm{X}^{\exists}\,\psi$ and $i + 1 < |\rho|$ and $(\rho, i + 1) \models \psi$,
- $\phi = \mathrm{X}^{\forall}\psi$ and $i + 1 = |\rho|$ or $(\rho, i + 1) \models \psi$,
- $\phi = \phi_l\,\mathrm{U}\,\phi_r$ and $\exists i \leq j < |\rho|$ s.t. $(\rho, j) \models \phi_r$ and $\forall i \leq k < j$, $(\rho, k) \models \phi_l$, or
- $\phi = \phi_l\,\mathrm{R}\,\phi_r$ and $\forall i \leq j < |\rho|.(\rho, j) \models \phi_r$ or $\exists i \leq k < j$ s.t. $(\rho, k) \models \phi_l$.

For $|\rho| = \infty$ and for any $i \in \mathbb{N}$ it holds that $(\rho, i) \models \mathrm{X}^{\exists}\,\psi$ if and only if $(\rho, i) \models \mathrm{X}^{\forall}\psi$. An LTL formula ϕ is *satisfied* by a nonempty path ρ (denoted by $\rho \models \phi$) if and only if $(\rho, 0) \models \phi$. A Kripke structure K satisfies the formula ϕ (denoted by $K \models \phi$) if and only if for all $\rho \in \mathrm{lang}(K)$ it holds that $\rho \models \phi$. Two LTL formulas ϕ and ψ are said to be equivalent ($\phi \equiv \psi$) if and only if for all paths ρ it holds that $\rho \models \phi$ if and only $\rho \models \psi$.

An LTL formula ϕ is said to be in *positive normal form* if in ϕ only atomic propositions appear in the scope of the symbol $\neg$. The following *dualities* ensure that each LTL formula ϕ can be rewritten into an equivalent formula ϕ' in positive normal form with $|\phi'| = O(|\phi|)$.

$$\begin{aligned} \neg\neg\phi &\equiv \phi \ ; \\ \neg \mathrm{X}^{\forall}\phi &\equiv \mathrm{X}^{\exists}\,\neg\phi \ ; \\ \neg(\phi_l \wedge \phi_r) &\equiv (\neg\phi_l) \vee (\neg\phi_r) \ ; \\ \neg(\phi_l \,\mathrm{U}\, \phi_r) &\equiv (\neg\phi_l)\,\mathrm{R}(\neg\phi_r) \ . \end{aligned}$$

Given a class of Kripke structures $\mathscr{K}$. The *model checking problem* of LTL over $\mathscr{K}$ (MC[LTL, $\mathscr{K}$]) is defined by

$$\mathrm{MC[LTL},\, \mathscr{K}] = \left\{ K \overset{?}{\models} \phi \mid K \in \mathscr{K}, \phi \in LTL \right\} \ .$$

In the following we often use classes of Kripke structures that are defined through a class of Kripke frames. E.g. *path* denotes the class of all Kripke structures with a frame that is a (finite) path.

Model Checking a Path. Automata-based techniques have proved very successful in establishing upper bounds on complexity of LTL model checking problems. However, there seems to be a barrier on proving sub-polynomial bounds via automata constructions. In [13], we gave a construction that uses monotone Boolean circuits in place of the usual automata-based constructions in order to prove that the LTL path checking problem is in NC.

Theorem 1 (Kuhtz and Finkbeiner (2009)).
MC[LTL, path] is in $\mathsf{AC}^1(\mathsf{logDCFL})$. □

Generalized Stutter Equivalence. It is a well known property of the star free regular languages, which is precisely captured by LTL, that those languages can only count up to a threshold but are unable to do modulo counting. In [15] Kučera and Strejček introduce the notion of *generalized stutter equivalence* which reflects the disability to count of LTL on a syntactic level.

Definition 1 (Generalized Stutter Equivalence). *Given a computation path ρ, a subsequence $\rho_{i,j}$ of ρ is* (m,n)-redundant *if $\rho_{(j+1),(j+1)+m\cdot(j-i)-m+1+n}$ is a prefix of $\rho_{i,j}^{\omega}$.*

We say that two computation paths ρ and σ are (m,n)-stutter equivalent *if ρ is obtained from σ by removing non-overlapping (m,n)-redundant subsequences, or vice versa.*

Kučera and Strejček prove the following generalized stuttering theorem for LTL. Intuitively the theorem states that an LTL formula can not distinguish between words that differ through repeated occurrence (stuttering) of a sub-word beyond a certain threshold that depends on the nesting depth of the different temporal operators.

Theorem 2 (Kučera and Strejček (2005)). *Given an LTL formula ϕ with maximal nesting depth of* U *and* R *modalities of m and with maximal nesting depth of* $\mathrm{X}^{\exists}$ *and* $\mathrm{X}^{\forall}$ *modalities of n, the set of $\{\rho \mid \rho \models \phi\}$ is closed under (m,n)-stutter equivalence.* □

Using this theorem, we will establish upper bounds for model checking of weak Kripke structures.

Model Checking LTL with a single variable. In their seminal paper [21] Sistla and Clarke prove general LTL model checking to be PSPACE-complete. The result is obtained via a reduction of the satisfiability problem to the co-model checking problem. In the same paper, LTL satisfiability is proved PSPACE-complete by encoding the computations of a PSPACE Turing machine into an LTL formula. In [8], Demri and Schnoebelen strengthen this result by showing that satisfiability is hard even for the fragment of LTL with a single atomic proposition.

Theorem 3 (Demri and Schnoebelen (2002)). *Satisfiability of LTL with a single atomic proposition is* PSPACE*-complete.* □

Restricting our attention only to formulas with a single variable, we can represent the class of all possible models in a Kripke structure with just two vertices and two different labels. When proving PSPACE-hardness of model checking of non-weak Kripke frames, we will use the fact that this simple structure can be embedded into any non-weak Kripke frame.

4 LTL Model Checking Problems in NC

We start with some theorems that are corollaries from Theorem 1. In general, any class of Kripke structures for which the model checking problem can be deterministically reduced to a polynomial number of parallel path checking problems can be model checked in NC. In particular, trees can be decomposed into a linear number of paths:

Theorem 4. *MC[LTL, tree] is in* $\mathsf{AC}^1(\mathsf{logDCFL})$. □

DAGs of constantly bounded depth can be unfolded into trees with only polynomial blowup:

Theorem 5. *MC[LTL, DAG of depth $O(1)$] is in* $\mathsf{AC}^1(\mathsf{logDCFL})$. □

Markey and Schnoebelen present in [19] a reduction from the problem of checking ultimately periodic paths to the finite path checking problem. We provide a more general reduction. We start with an observation about weak Kripke structures:

Lemma 1. *Let $\mathcal{K}$ be a weak Kripke structure. Any (finite or infinite) computation path $\rho \in \mathrm{lang}(\mathcal{K})$ is of the form $\left(\prod_{i=0}^{n-1} u_i \cdot v_i^{\alpha_i}\right)$ with*

- $n \leq |\mathcal{K}|$,
- $\alpha_i \in \mathbb{N}$ *for* $i < n-1$ *and* $\alpha_{n-1} \in \mathbb{N} \cup \{\infty\}$, *and*
- u_i, v_i *are finite paths in* $\mathcal{K}$

for $0 \leq i < n$.

Proof. The statement of the lemma follows from the fact that for a weak Kripke structure all cycles are disjoint and the SCC graph is a directed acyclic graph. □

The lemma implies that we can represent a computation path ρ in a weak Kripke structure $\mathcal{K}$ as a path R in the SCC graph of $\mathcal{K}$ together with the coefficient α_i for each cycle v_i that occurs in ρ. We denote this representation of ρ by $\mathsf{R}_{\alpha_0,\ldots,\alpha_{n-1}}$.

Lemma 2. *Given an LTL formula* ϕ *and a weak Kripke structure* $\mathcal{K}$. *If there is a computation path* $\rho = \mathsf{R}_{\alpha_0,\ldots,\alpha_{n-1}} \in \text{lang}(\mathcal{K})$ *with* $\rho \models \phi$ *then there is a computation path* $\rho' = \mathsf{R}_{\beta_0,\ldots,\beta_{n-1}}$ *with* $\beta_i \leq |\phi| + 1$ *such that* $\rho' \models \phi$. *In particular it holds that* $|\rho'| = O(|\phi| \cdot |\mathcal{K}|)$.

Proof. Represent the computation path ρ according to Lemma 1 and apply the generalized stuttering theorem (Theorem 2) from Kučera and Strejček. □

Lemma 2 subsumes the reduction from [19]. By reducing the problem of checking an ultimately periodic path to the finite path checking problem we get the following theorem:

Theorem 6. *MC[LTL, ultimately periodic path] is in* $\mathsf{AC}^1(\mathsf{logDCFL})$.

Proof. The computation path ρ can be represented as R_∞. Due to Lemma 2 it is sufficient to enumerate and check all computation paths R_α for $\alpha \leq |\phi| + 1$. By Theorem 1 each check can be done in $\mathsf{AC}^1(\mathsf{logDCFL})$. Since all checks are independent we can do them all in parallel.

Remark: A more careful interpretation of Theorem 2 would reveal that a single check for $\alpha = |\phi| + 1$ is actually sufficient. □

We can use Lemma 2 to generalize the result to Kripke structures with SCC graphs of constantly bounded depth.

Theorem 7. *MC[LTL, weak, SCC graph of depth* $O(1)$*] is in* $\mathsf{AC}^1(\mathsf{logDCFL})$.

Proof. Unfold the Kripke structure into a tree of polynomial size and constant depth. Each computation path in the unfolded structure can be represented as $\mathsf{R}_{\alpha_0,\ldots\alpha_n}$ where $n \in \mathbb{N}$ is a constant. Due to Theorem 2, it sufficient to enumerate (in L) and check all computation paths for $\alpha_i \leq |\phi|+1$ $(0 \leq i \leq n)$. In total there is a polynomial number of computation paths to be checked. Using Theorem 1, all checks can be done in parallel in $\mathsf{AC}^1(\mathsf{logDCFL})$. □

We can generalize the previous result a bit further: let G be the SCC graph of a Kripke structure $\mathcal{K}$. For a vertex $v \in V(G)$ let

$$\zeta(v) = \alpha(v) + \sum_{\langle w,v\rangle \in E(\mathcal{K})} \beta(v) \cdot \zeta(w)$$

where

$$\alpha(v) = \begin{cases} 1 & \text{if } v \text{ is initial in } \mathcal{K} \text{ and} \\ 0 & \text{otherwise,} \end{cases}$$

$$\beta(v) = \begin{cases} 2 & \text{if } v \text{ represents a cycle of } \mathcal{K} \text{ and} \\ 1 & \text{otherwise, and} \end{cases}$$

the empty sum equals zero. Intuitively, the function $\zeta(v)$ counts the number of paths that lead from an initial state to v, where each cycle occurs either zero or one times in a path. Let $\zeta(\mathcal{K}) = \max_{v \in V(G)} \zeta(v)$.

Theorem 8. *For any class $\mathscr{C}$ of weak Kripke structures such that ζ is polynomial in the size of the structure it holds that MC[LTL, $\mathscr{C}$] is in* $\mathsf{AC}^1(\mathsf{logDCFL})$. □

5 coNP-Complete LTL Model Checking Problems

In favor of a more concise presentation, we exclusively consider LTL over infinite paths throughout the remainder of this paper.

Theorem 9. *LTL model checking of weak Kripke structures is* coNP*-complete.*

Proof. We prove the upper bound by guessing a possibly infinite path and then using Lemma 2 to reduce the problem to the finite path checking problem. Let $\mathcal{K}$ be a weak Kripke structure. In order to decide if $\mathcal{K} \not\models \phi$ guess a path R in the SCC graph of $\mathcal{K}$ such that there is a path $\rho = \mathsf{R}_{\alpha_0,\ldots,\alpha_{n-1}} \in \text{lang}(\mathcal{K})$ with $\rho \models \neg\phi$. Use Lemma 2 to reduce this to checking $\rho' \models \neg\phi$ for a finite path ρ' of polynomial length. Do this check by use of Theorem 1 in $\mathsf{AC}^1(\mathsf{logDCFL}) \subseteq \mathsf{P}$. Hence the model checking problem for weak Kripke structures is in coNP.

The proof of the lower bound reduces the satisfiability problem of propositional logic to the co-model checking problem of weak Kripke structures. The reduction is very similar to the reduction used by [21] to show that the co-model checking problem for the fragment of LTL that has F as the only modality is NP-hard.

Given a propositional logic formula f over the set of variables $\{v_0, \ldots, v_n\}$, $n \in \mathbb{N}$. We obtain the LTL formula ϕ from f by substituting for all $0 \le i \le n$ each occurrence of the variable v_i by the LTL formula $\mathrm{X}^{\exists^{2i+1}} p$, where $p \in \mathrm{AP}$. The formula f is satisfiable if and only if $\neg\phi$ does not hold on the Kripke structure $\mathcal{K}$ shown in Figure 1. □

The above proof actually provides a slightly stronger result than stated in Theorem 9:

Corollary 1. *The problem of model checking LTL on planar acyclic graphs is* coNP*-hard.* □

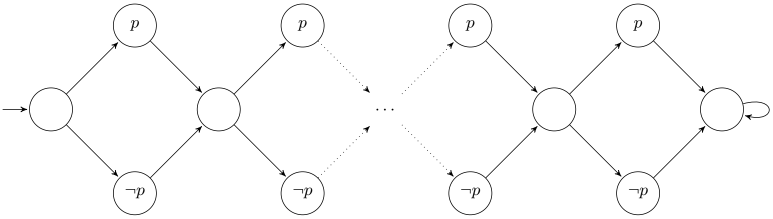


Fig. 1. Kripke structure used to reduce propositional SAT to LTL model checking

The hardness result from Theorem 9 can be sharpened to more restricted classes with a coNP lower bound on the model checking problem. In order to prepare the proof of the next theorem we show here the construction for Kripke structures with an SCC graph that is a path. In fact self loops, i.e. state-stuttering, is sufficient. The idea is similar to the lower bound from the previous theorem, but the diamond shaped substructures are replaced by self loops. For a propositional formula f with variables $v_0, \dots, v_{n-1}$ we build a Kripke structure $\mathcal{K}$ that is a sequence of n self loops as shown in Figure 2 where each vertex is labeled with a unique state (represented as a propositional formula) p_i. The LTL formula ϕ is obtained by substituting in f each variable v_i with the LTL formula $\mathrm{F}(p_i \wedge \mathrm{X}^{\exists}\, p_i)$. It is easy to check that f is satisfiable if and only if $\mathcal{K} \not\models \neg\phi$.

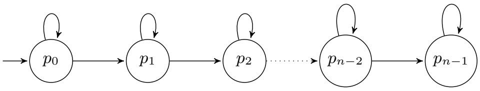


Fig. 2. Kripke structure used to reduce SAT to LTL model checking of Kripke structures for which the SCC graph is a path

The next theorem is a refined (though more technical) version of Theorem 9. Recall that the function ζ from Section 4 counts the number of paths in a weak Kripke structure where each cycle occurs at most once.

Theorem 10. *For any class $\mathscr{C}$ of weak Kripke structures for which ζ is exponential in the size of the structure it holds that MC[LTL, $\mathscr{C}$] is complete for* coNP.

The upper bound remains the same as in Theorem 9; the lower bound is refined through a stronger constraint on the classes of structures.

Proof. The proof for the lower bound combines the proof for the lower bound for planar DAGs and the lower bound for paths Kripke structures with an SCC graph that is a path. Again, we reduce SAT to the co-model checking problem on $\mathscr{C}$. Let f a propositional formula with variables $v_0, \dots, v_{n-1}$. There is a Kripke structure $\mathcal{K}$ in $\mathscr{C}$ such that the SCC graph of $\mathcal{K}$ contains sequence of vertices

$v_0, \ldots, v_{n-1}$, where v_i is reachable from v_{i-1}. Due to the exponential growth of ζ there are $O(n)$ many i with $\zeta(v_i) = O(2 \cdot v_{i-1})$. Moreover, we know that either there are two distinct paths from v_{i-1} to v_i with two distinguishing vertices v_i^0 and v_i^1, or v_i represents a cycle. In the former case we label v_i^0 with a unique label p_i (represented by a propositional formula) and substitute v_i in f with the LTL formula $\mathrm{F}\, p_i$. In the latter case we label v_i with a unique label p_i (represented as a propositional formula) and substitute any occurrence of v_i with the LTL formula $\mathrm{F}(p_i \wedge \mathrm{X}^{\exists}(\mathit{true}\, \mathrm{U}\, p_i))$. We call the resulting LTL formula ϕ. Again, it is easy to check that $\mathcal{K} \not\models \neg\phi$ if and only if f is satisfiable. □

The classification of Kripke structures between NC and coNP-hardness is not a complete dichotomy. There is a gap concerning the structures with $n^{O(1)} \ll \zeta(k) \ll O(2^n)$. This is illustrated via the following theorem.

Theorem 11. *For any class $\mathscr{C}$ of weak Kripke structures with $\zeta = O(n^{\log^{O(1)} n})$, where n is the size of a Kripke structure, it holds that MC[LTL, $\mathscr{C}$] is in* polyL.

Proof. We can enumerate all computation paths of polynomial length that are relevant according to Lemma 2 in polyL. Each computation path can be checked in $\mathsf{AC}^1(\mathsf{logDCFL}) \subseteq$ polyL. □

6 PSPACE-Complete LTL Model Checking Problems

We now investigate the borderline between the frames whose model checking problems are in coNP and those whose model checking problems are PSPACE-hard. It turns out that LTL model checking is PSPACE-complete for *any* non-weak Kripke frame. In contrast to the previous PSPACE hardness results, we get a lower bound that does not depend on the asymptotic behavior of a class of Kripke structures, but holds for each non-weak Kripke frame. Moreover, together with Theorem 9 we obtain a dichotomic classification.

Theorem 12. *The LTL model checking problem is* PSPACE*-complete for any non-weak Kripke frame.*

Proof. Given a non-weak Kripke frame $\mathcal{F}$. We reduce the validity problem for LTL to the co-model checking problem on a structure $\mathcal{K}$ with frame $\mathcal{F}$.

We start by choosing an adequate labeling for $\mathcal{K}$. Let $s, t, u \in 2^{\mathrm{AP}}$ be pairwise disjoint states represented as Boolean formulas. Because $\mathcal{F}$ is non-weak we know that there are two distinct cycles that share a common vertex x. Label x with state s. There is a vertex y that is present in only one of the two cycles. Label y with state t. Label all remaining vertices of $\mathcal{K}$ with u. Figure 3 provides a schematic view of $\mathcal{K}$.

Given some formula ζ with only a single variable. By Theorem 3, deciding validity for an LTL formula with only a single atomic proposition is PSPACE-hard. ζ is valid if and only if $\neg\zeta$ is not satisfiable. To decide if $\phi = \neg\zeta$ is satisfiable is the co-model checking problem on a universal Kripke structure. We will reduce

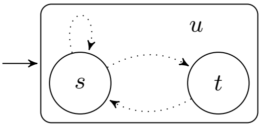


Fig. 3. Non-weak Kripke structure with the labeling used in the proof of Theorem 12

the latter for ϕ to the co-model checking problem on the Kripke structure $\mathcal{K}$ with the labeling given above for a formula ϕ^* that can be constructed from ϕ in L. Since PSPACE is closed under complement, the model checking problem for $\mathcal{K}$ is thus PSPACE hard.

We assume that the unique atomic proposition that occurs in ϕ is p. A Kripke structure with two states, namely p and $\neg p$, that represents the universal language $\{p, \neg p\}^\omega$ is shown in Figure 4.

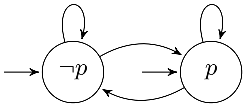


Fig. 4. The Kripke structure that represents the universal language $\{p, \neg p\}^\omega$

In the following we call a suffix of a computation path an a-suffix if the first state of the suffix is a. We call a cycle in a Kripke structure an a-cycle if it starts in an a-state. We identify the cycle with the corresponding state sequence.

The construction of ϕ^* is as follows: First, transform ϕ into positive normal form in L. Next, we define inductively a formula ϕ'. For the cases that ϕ is either an atomic proposition or a negated atomic proposition let

- $p' = s \wedge \mathrm{X}^\exists\,(u \,\mathrm{U}\, s)$ and
- $(\neg p') = s \wedge \mathrm{X}^\exists\,\big(u \,\mathrm{U}\,\big(t \wedge \mathrm{X}^\exists\,(u \,\mathrm{U}\, s)\big)\big)$.

The idea is that the formula p' holds exclusively on an s-cycle that does not include the t state, whereas $(\neg p)'$ holds on any s-cycle that visits the t state. This way, each s-cycle encodes a state of the original Kripke structure. The formula will translate each single step in the original Kripke structure into an s-cycle in $\mathcal{K}$. The remaining cases for ϕ' are defined inductively as follows:

- $\psi' \wedge \chi'$ for $\phi = \psi \wedge \chi$,
- $\psi' \vee \chi'$ for $\phi = \psi \vee \chi$,
- $s \wedge \mathrm{X}^\exists\,(\neg s \,\mathrm{U}\, \psi')$ for $\phi = \mathrm{X}^\exists\,\psi$,
- $s \wedge ((s \rightarrow \psi') \,\mathrm{U}\, \chi')$ for $\phi = \psi \,\mathrm{U}\, \chi$, and
- $s \wedge (\psi' \,\mathrm{R}\, (s \rightarrow \chi'))$ for $\phi = \psi \,\mathrm{R}\, \chi$.

For any formula ϕ we want that a computation path that models ϕ' is “meaningful” in the sense that it can be mapped to a computation path having only p and

$\neg p$ states. Therefore we defined $'$ such that a formula ϕ' holds on a computation path ρ only if ρ starts in the s state.

Finally, we set $\phi^* = (\mathrm{G}\,\mathrm{F}\, s) \wedge (\neg s \,\mathrm{U}\, \phi')$. By requiring that any computation path ρ with $\rho \models \phi^*$ returns to s infinitely often, we prevent that ρ "gets stalled" in some "meaningless" loop. The formula allows the computation to reach s initially, and then it forces the remaining computation path to satisfy ϕ' which encodes on $\mathcal{K}$ the original meaning of ϕ.

We claim that for each computation path ρ it holds that $\rho \models \phi$ if and only if there is a computation path $\rho^* \in \mathrm{lang}(\mathcal{K})$ such that $\rho^* \models \phi^*$. Let c be a s-cycle in $\mathcal{K}$ that does not visit t and let d an s-cycle in $\mathcal{K}$ that visits t. For proving the "only if" part of the claim assume that $\rho \models \phi$. Let ρ' be defined as follows:

$$\rho' = \begin{cases} c \cdot \rho'_{1,\dots} & \text{if } \rho_0(p) \text{, and} \\ d \cdot \rho'_{1,\dots} & \text{otherwise.} \end{cases}$$

It holds that $\rho' \in \mathrm{lang}(\mathcal{K})$, every suffix of ρ' contains an s-state, and ρ' starts with s. Further $'$ induces a surjective mapping from the suffixes of ρ to s-suffixes of ρ'. This mapping is monotonic with respect to the order of start position of the suffixes.

Let e be the (possibly empty) sequence of states on a shortest path in $\mathcal{K}$ that leads from an initial state to s. Let $\rho^* = e \cdot \rho'$. Every suffix of ρ^* contains an s state and therefore it holds that $\rho^* \models \mathrm{G}\,\mathrm{F}\, s$. Since all states in e contradict s from the definition of ρ^* it follows directly that $\rho^* \models (\neg s) \,\mathrm{U}\, \phi'$ if $\rho' \models \phi'$. We prove this by induction over ϕ:

- For $\phi = p$ it holds that if ρ starts with p then ρ' starts with c followed by an s state. Hence $\rho' \models s \wedge \mathrm{X}^{\exists}\, (u \,\mathrm{U}\, s)$.
- For $\phi = \neg p$ it holds that if ρ starts with $\neg p$ then ρ' starts with d followed by an s state. Hence $\rho' \models s \wedge \mathrm{X}^{\exists} \big(u \,\mathrm{U} \big(t \wedge \mathrm{X}^{\exists}\, (u \,\mathrm{U}\, s)\big)\big)$.
- For $\phi \in \{\psi \wedge \chi, \psi \vee \chi\}$ the claim follows directly from the induction hypothesis and the semantics of LTL.
- For $\phi = \mathrm{X}^{\exists}\, \psi$ it holds that $\rho_{1,\dots} \models \psi$. By induction hypothesis $(\rho_{1,\dots})' \models \psi'$ and by definition of ρ' it holds that $\rho' \models s \wedge \mathrm{X}^{\exists}\, (\neg s \,\mathrm{U}\, \psi')$.
- For $\phi = \psi \,\mathrm{U}\, \chi$ there is a position i such that $\rho_{i,\dots} \models \chi$ and $\rho_{j,\dots} \models \phi$ for all $j < i$. By induction hypothesis there is an l such that $\rho'_{l,\dots} \models \chi'$. Since $'$ is surjective and monotonic on the s-suffixes of ρ' from the induction hypothesis it follows that $\rho'_{j,\dots} \models \psi'$ for all s-suffixes with $j < l$. Further recall that ψ' holds only on computation paths that start with s. Hence, for all non-s-suffix $\rho'_{j,\dots}$ with $j < l$ the formula $(s \rightarrow \psi')$ holds trivially. Thus, for all $j < l$ it holds that $\rho'_{j,\dots} \models (s \rightarrow \psi')$ and we get $\rho' \models s \wedge (s \rightarrow \psi') \,\mathrm{U}\, \chi'$.
- The case for $\phi = \psi \,\mathrm{R}\, \chi$ is analogous to the previous case.

We now prove the "if" part of the claim. Given a computation path in $\sigma \in \mathrm{lang}(\mathcal{K})$ such that $\sigma \models \phi^*$. There is a position i_0 such that for all $j < i_0$ it holds that $\sigma_{j,\dots} \models \neg s$ and $\sigma_{i_0,\dots} \models \phi'$. Let $\sigma^0 = \sigma_{i_0,\dots}$. We show that $\sigma^0 \models \phi'$ implies that there is a computation path σ' with $\sigma' \models \phi$. We know that σ^0 contains only

s, t, and u states. Moreover, we know from the definition of ϕ^* that any suffix of σ^0 contains an s state. The computation path σ' is defined as follows:

$$\sigma' = \begin{cases} p \cdot \sigma'_{s_0,\dots} & \text{if } \sigma^0_{0,s_0} \text{ contains no } t \text{ state, and} \\ \neg p \cdot \sigma'_{s_0,\dots} & \text{otherwise,} \end{cases}$$

where s_0 is the position of the first s-state in $\sigma_{1,\dots}$. Note that $'$ induces a monotonic and surjective mapping from the s-suffixes of σ^0 to the suffixes of σ. We show by induction over ϕ that $\sigma' \models \phi$:

- For $\phi = p$ we have $\sigma^0 \models s \wedge \mathrm{X}^{\exists}(u \,\mathrm{U}\, s)$. This implies that σ^0_{0,s_0} does not contain any t state and therefore $\sigma'_0(p)$.
- For $\phi = \neg p$ we have $\sigma^0 \models s \wedge \mathrm{X}^{\exists}\big(u \,\mathrm{U}\, \big(t \wedge \mathrm{X}^{\exists}(u \,\mathrm{U}\, s)\big)\big)$. Therefore σ^0_{0,s_0} contains a t state and hence $\sigma'_0(\neg p)$.
- For $\phi \in \{\psi \wedge \chi, \psi \vee \chi\}$ the claim follows directly from the induction hypothesis and the semantics of LTL.
- For $\phi = \mathrm{X}^{\exists}\,\psi$ we have $\sigma^0 \models s \wedge \mathrm{X}^{\exists}(\neg s \,\mathrm{U}\, \psi')$. This implies that $\sigma^0_{s_0,\dots} \models \psi'$. From the induction hypothesis it follows that $\sigma'_{1,\dots} \models \psi$ and thus $\sigma' \models \mathrm{X}^{\exists}\,\psi$.
- For $\phi = \psi \,\mathrm{U}\, \chi$ we have that $\sigma^0 \models s \wedge ((s \rightarrow \psi') \,\mathrm{U}\, \chi')$. Therefore there is an $i \in \mathbb{N}$ such that $\sigma^0_{i,\dots} \models \chi'$ and for all $j < i$ it holds that $\sigma^0_{j,\dots} \models (s \rightarrow \psi')$. By induction hypothesis there is an $l \in \mathbb{N}$ such that $\sigma'_{l,\dots} \models \chi$. For all s-suffixes $\sigma^0_{j,\dots}$ with $j < i$ it holds that $\sigma^0 j, \dots \models \psi'$. Recall that $'$ induces a monotonic and surjective function from the s-suffixes of σ^0 to the suffixes of σ'. Together with the induction hypothesis we deduce that for all $j < l$ it holds that $\sigma'_{j,\dots} \models \psi$. We conclude that $\sigma' \models \psi \,\mathrm{U}\, \chi$.
- The case for $\phi = \psi \,\mathrm{R}\, \chi$ is analogous to the previous case. Note, however, that there are infinitely many s-states in σ^0. □

7 Conclusions

We have developed a classification of Kripke structures with respect to the complexity of the model checking problem for LTL. We showed that the model checking problem for a Kripke frame is PSPACE-complete if and only if the frame is not weak. The problem is coNP-complete for the class of all weak Kripke structures. The problem is in NC for any class of Kripke structures for which the model checking problem can be reduced to a polynomial number of path checking problems. Examples of such classes include finite paths, ultimately periodic path, finite trees, directed graphs of constant depths, and classes of Kripke structures with an SCC graph of constant depth.

Open questions and future work. There are several open questions that deserve further study. Albeit small, there is a gap between $\mathsf{AC}^1(\mathsf{logDCFL})$ and the best known lower bound, NC^1 for LTL path checking. There is some hope to further reduce the upper bound towards NC^1, the currently known lower bound, because the construction in [13] relies on the algorithms for evaluating monotone planar Boolean circuits with all constant gates on the outer face. The circuits that appear in the construction actually exhibit much more structure. Another way to improve the upper bounds of the path and tree checking algorithms would be to prove a better upper bound for the problem of evaluating one-input-face monotone planar Boolean circuits.

Another area that requires more attention is the model checking problem on restricted frames for branching-time temporal logic. In the first author's thesis [12], it is shown that the model checking problem for CTL on finite trees is in NC (more precisely, in $\mathsf{AC}^2(\mathsf{logDCFL})$). The gap between NC as an upper bound and L as a lower bound for tree structures and P for general structures is comparably small. Still, beyond finite trees we do know very little about other classes for which the model checking problem for CTL is in NC. What are the properties of Kripke structures that allow for efficiently parallel model checking for CTL? What is the complexity of tree checking for CTL+past and CTL with a sibling axis? What is the complexity of CTL* tree checking?

Finally, an important question concerns the translation of the improved complexity bounds for restricted sets of frames into efficient practical model checking algorithms. The proofs in this paper and the underlying proofs for the path checking problem [13] are based on a variety of different computational models: Boolean circuits, space-restricted Turing machines, time-restricted Turing machines, Turing machines with push-down store, and parallel random access memory machines (PRAM). Are there practical parallel implementations for parallel path and tree checking?

Complexity classes that are characterized by efficient parallel algorithms and complexity classes that are characterized by space-efficient algorithms are tightly coupled through simulation theorems. Considering that in modern hardware architectures cache-efficiency and I/O-efficiency matter more than the simple number of computation steps, the following question seems even more important than the previous one: Can we derive practical space-efficient implementations from parallel path and tree checking algorithms? With the fast growing number of available cores in modern computing devices, on the one hand, and the tight resource restrictions on mobile devices, on the other hand, good trade-offs between cache-efficiency, I/O-efficiency, and CPU-usage become increasingly important.

References

1. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theoretical Computer Science 336(2-3), 209–234 (2005)
2. Baker, T.P.: The cyclic executive model and ada. The Journal of Real-Time Systems 1, 120–129 (1989)
3. Bauland, M., Mundhenk, M., Schneider, T., Schnoor, H., Schnoor, I., Vollmer, H.: The tractability of model-checking for ltl: The good, the bad, and the ugly fragments. Electr. Notes Theor. Comput. Sci. 231, 277–292 (2009)
4. Bauland, M., Schneider, T., Schnoor, H., Schnoor, I., Vollmer, H.: The complexity of generalized satisfiability for linear temporal logic. Logical Methods in Computer Science 5(1) (2009)

5. Benedikt, M., Libkin, L., Neven, F.: Logical definability and query languages over ranked and unranked trees. ACM Trans. Comput. Log. 8(2) (2007)
6. Buss, S.R.: The boolean formula value problem is in ALOGTIME. In: STOC, pp. 123–131. ACM, New York (1987)
7. Demri, S., Laroussinie, F., Schnoebelen, P.: A parametric analysis of the state-explosion problem in model checking. J. Comput. Syst. Sci. 72(4), 547–575 (2006)
8. Demri, S., Schnoebelen, P.: The complexity of propositional linear temporal logics in simple cases. Inf. Comput. 174(1), 84–103 (2002)
9. Havelund, K., Roşu, G.: Monitoring programs using rewriting. In: ASE, pp. 135–143. IEEE Computer Society, Los Alamitos (2001)
10. Hemaspaandra, E.: The complexity of poor man's logic. J. Log. Comput. 11(4), 609–622 (2001)
11. Hemaspaandra, E., Schnoor, H.: On the complexity of elementary modal logics. In: Albers, S., Weil, P. (eds.) STACS. LIPIcs, vol. 1, pp. 349–360. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)
12. Kuhtz, L.: Model Checking Finite Paths and Trees. PhD thesis, Universität des Saarlandes (2010)
13. Kuhtz, L., Finkbeiner, B.: LTL path checking is efficiently parallelizable. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 235–246. Springer, Heidelberg (2009)
14. Kupferman, O., Vardi, M.Y.: Relating linear and branching model checking. In: PROCOMET, pp. 304–326. Chapman & Hall, New York (1998)
15. Kučera, A., Strejček, J.: The stuttering principle revisited. Acta Inf. 41(7-8), 415–434 (2005)
16. Ladner, R.E.: The computational complexity of provability in systems of modal propositional logic. SIAM J. Comput. 6(3), 467–480 (1977)
17. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Proceedings of the Conference on Logic of Programs, pp. 196–218. Springer, London (1985)
18. Markey, N., Raskin, J.-F.: Model checking restricted sets of timed paths. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 432–447. Springer, Heidelberg (2004)
19. Markey, N., Schnoebelen, P.: Model checking a path (preliminary report). In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 251–265. Springer, Heidelberg (2003)
20. Markey, N., Schnoebelen, P.: Mu-calculus path checking. Inf. Process. Lett. 97(6), 225–230 (2006)
21. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM 32(3), 733–749 (1985)
22. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 223. Springer, Heidelberg (2002)

Efficient CTL Model-Checking for Pushdown Systems*

Fu Song and Tayssir Touili

LIAFA, CNRS and Univ. Paris Diderot, France.
{song,touili}@liafa.jussieu.fr

Abstract. Pushdown systems (PDS) are well adapted to model sequential programs with (possibly recursive) procedure calls. Therefore, it is important to have efficient model checking algorithms for PDSs. We consider in this paper CTL model checking for PDSs. We consider the "standard" CTL model checking problem where whether a configuration of a PDS satisfies an atomic proposition or not depends only on the control state of the configuration. We consider also CTL model checking with regular valuations, where the set of configurations in which an atomic proposition holds is a regular language. We reduce these problems to the emptiness problem in Alternating Büchi Pushdown Systems, and we give an algorithm to solve this emptiness problem. Our algorithms are more efficient than the other existing algorithms for CTL model checking for PDSs in the literature. We implemented our techniques in a tool, and we applied it to different case studies. Our results are encouraging. In particular, we were able to find bugs in linux source code.

1 Introduction

PushDown Systems (PDS for short) are an adequate formalism to model sequential, possibly recursive, programs [10,13]. It is then important to have verification algorithms for pushdown systems. This problem has been intensively studied by the verification community. Several model-checking algorithms have been proposed for both linear-time logics [1,13,9,14,17], and branching-time logics [1,2,6,24,18,19,14,17].

In this paper, we study the CTL model-checking problem for PDSs. First, we consider the "standard" model-checking problem as was considered in the literature. In this setting, whether a configuration satisfies an atomic proposition or not depends only on the control state of the configuration, not on its stack content. This problem is known to be EXPTIME-complete [25]. CTL corresponds to a fragment of the alternation-free μ-calculus and of CTL*. Existing algorithms for model-checking these logics for PDSs could then be applied for CTL model-checking. However, these algorithms either allow only to decide whether a given configuration satisfies the formula i.e., they cannot compute all the set of PDS configurations where the formula holds [5,6,24,18], or have a high complexity [19,2,1,12,11,14,17]. Moreover, these algorithms have not been implemented due to their high complexity. Thus, there does not exist a tool for CTL model-checking of PDSs.

In this work, we propose a new efficient algorithm for CTL-model checking for PDSs. Our algorithm allows to compute the set of PDS configurations that satisfy a

* Work partially funded by ANR grant ANR-08-SEGI-006.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 434–449, 2011.

given CTL formula. Our procedure is more efficient than the existing model-checking algorithms for μ-calculus and CTL* that are able to compute the set of configurations where a given property holds [19,2,1,12,11,14,17]. Our technique reduces CTL model-checking to the problem of computing the set of configurations from which an Alternating Büchi Pushdown System (ABPDS for short) has an accepting run. We show that this set can be effectively represented using an alternating finite automaton.

Then, we consider CTL model checking with regular valuations. In this setting, the set of configurations where an atomic proposition holds is given by a finite state automaton. Indeed, since a configuration of a PDS has a control state and a stack content, it is natural that the validity of an atomic proposition in a configuration depends on both the control state *and the stack*. For example, in one of the case studies we considered, we needed to check that whenever a function *call_hpsb_send_phy_config* is invoked, there is a path where *call_hpsb_send_packet* is called before *call_hpsb_send_phy_config* returns. We need propositions about the stack to express this property. "Standard" CTL is not sufficient. We provide an efficient algorithm that solves CTL model checking with regular valuations for PDSs. Our procedure reduces the model-checking problem to the problem of computing the set of configurations from which an ABPDS has an accepting run.

We implemented our techniques in a tool for CTL model-checking for pushdown systems. Our tool deals with both "standard" model-checking, and model-checking with regular valuations. As far as we know, this is the *first* tool for CTL model-checking for PDSs. Indeed, existing model-checking tools for PDSs like Moped [21] consider only reachability and LTL model-checking, they don't consider CTL. We run several experiments on our tool. We obtained encouraging results. In particular, we were able to find bugs in source files of the linux system, in a watchdog driver of linux, and in an IEEE 1394 driver of linux. We needed regular valuations to express the properties of some of these examples.

Outline. The rest of the paper is structured as follows. Section 2 gives the basic definitions used in the paper. In section 3, we present an algorithm for computing an alternating automaton recognizing all the configurations from which an ABPDS has an accepting run. Sections 4 and 5 describe the reductions from "standard" CTL model-checking for PDSs and CTL model-checking for PDSs with regular valuations, to the emptiness problem in ABPDS. The experiments are provided in Section 6. Section 7 describes the related work.

2 Preliminaries

2.1 The Temporal Logic CTL

We consider the standard branching-time temporal logic CTL. For technical reasons, we use the operator $\tilde{U}$ as a dual of the until operator for which the stop condition is not required to occur; and we suppose w.l.o.g. that formulas are given in positive normal form, i.e., negations are applied only to atomic propositions. Indeed, each CTL formula can be written in positive normal form by pushing the negations inside.

Definition 1. *Let* $\mathrm{AP} = \{a, b, c, ...\}$ *be a finite set of atomic propositions. The set of CTL formulas is given by (where* $a \in \mathrm{AP}$*):*

$$\varphi ::= a \mid \neg a \mid \varphi\wedge\varphi \mid \varphi\vee\varphi \mid AX\varphi \mid EX\varphi \mid A[\varphi U\varphi] \mid E[\varphi U\varphi] \mid A[\varphi\tilde{U}\varphi] \mid E[\varphi\tilde{U}\varphi].$$

The closure $cl(\varphi)$ of a CTL formula φ is the set of all the subformulas of φ, including φ. Let $AP^+(\varphi) = \{a \in AP \mid a \in cl(\varphi)\}$ and $AP^-(\varphi) = \{a \in AP \mid \neg a \in cl(\varphi)\}$. The size $|\varphi|$ of φ is the number of elements in $cl(\varphi)$. Let $T = (S, \longrightarrow, c_0)$ be a transition system where S is a set of states, $\longrightarrow \subseteq S \times S$ is a set of transitions, and c_0 is the initial state. Let $s, s' \in S$. s' is a successor of s iff $s \longrightarrow s'$. A path is a sequence of states $s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \longrightarrow s_{i+1}$. Let $\lambda : AP \rightarrow 2^S$ be a labelling function that assigns to each atomic proposition a set of states in S. The validity of a formula φ in a state s w.r.t. the labelling function λ, denoted $s \models_\lambda \varphi$, is defined inductively in **Figure 1**. $T \models_\lambda \varphi$ iff $c_0 \models_\lambda \varphi$. Note that a path π satisfies $\psi_1 \tilde{U} \psi_2$ iff either ψ_2 holds everywhere in π, or the first occurrence in the path where ψ_2 does not hold must be preceeded by a position where ψ_1 holds.

$s \models_\lambda a \iff s \in \lambda(a).$
$s \models_\lambda \neg a \iff s \notin \lambda(a).$
$s \models_\lambda \psi_1 \wedge \psi_2 \iff s \models_\lambda \psi_1$ and $s \models_\lambda \psi_2$.
$s \models_\lambda \psi_1 \vee \psi_2 \iff s \models_\lambda \psi_1$ or $s \models_\lambda \psi_2$.
$s \models_\lambda AX\, \psi \iff s' \models_\lambda \psi$ for every successor s' of s.
$s \models_\lambda EX\, \psi \iff$ There exists a successor s' of s s.t. $s' \models_\lambda \psi$.
$s \models_\lambda A[\psi_1 U \psi_2] \iff$ For every path of $T, \pi = s_0, s_1, ...,$ with $s_0 = s, \exists i \geq 0$ s.t. $s_i \models_\lambda \psi_2$ and $\forall 0 \leq j < i, s_j \models_\lambda \psi_1$.
$s \models_\lambda E[\psi_1 U \psi_2] \iff$ There exists a path of $T, \pi = s_0, s_1, ...,$ with $s_0 = s,$ s.t. $\exists i \geq 0, s_i \models_\lambda \psi_2$ and $\forall 0 \leq j < i, s_j \models_\lambda \psi_1$.
$s \models_\lambda A[\psi_1 \tilde{U} \psi_2] \iff$ For every path of $T, \pi = s_0, s_1, ...,$ with $s_0 = s, \forall i \geq 0$ s.t. $s_i \not\models_\lambda \psi_2$, $\exists 0 \leq j < i,$ s.t. $s_j \models_\lambda \psi_1$.
$s \models_\lambda E[\psi_1 \tilde{U} \psi_2] \iff$ There exists a path of $T, \pi = s_0, s_1, ...,$ with $s_0 = s,$ s.t. $\forall i \geq 0$ s.t. $s_i \not\models_\lambda \psi_2$, $\exists 0 \leq j < i$ s.t. $s_j \models_\lambda \psi_1$.

Fig. 1. Semantics of CTL

2.2 PushDown Systems

Definition 2. *A PushDown System* (PDS for short) *is a tuple $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules and $\sharp \in \Gamma$ is a bottom stack symbol.*

A configuration of $\mathcal{P}$ is an element $\langle p, \omega \rangle$ of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. For technical reasons, we consider the bottom stack symbol $\sharp$, and we assume w.l.o.g. that it is never popped from the stack, i.e., there is no transition rule of the form $\langle p, \sharp \rangle \hookrightarrow \langle q, \omega \rangle \in \Delta$. The successor relation $\rightsquigarrow_\mathcal{P} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_\mathcal{P} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$.

Let c be a given initial configuration of $\mathcal{P}$. Starting from c, $\mathcal{P}$ induces the transition system $T^c_\mathcal{P} = (P \times \Gamma^*, \rightsquigarrow_\mathcal{P}, c)$. Let AP be a set of atomic propositions, φ be a CTL formula on AP, and $\lambda : AP \rightarrow 2^{P \times \Gamma^*}$ be a labelling function. We say that $(\mathcal{P}, c) \models_\lambda \varphi$ iff $T^c_\mathcal{P} \models_\lambda \varphi$.

2.3 Alternating Büchi PushDown Systems

Definition 3. *An Alternating Büchi PushDown System* (ABPDS for short) *is a tuple $\mathcal{BP} = (P, \Gamma, \Delta, F)$, where P is a finite set of control locations, Γ is the stack alphabet,*

$F \subseteq P$ is a finite set of accepting control locations and Δ is a function that assigns to each element of $P \times \Gamma$ a positive boolean formula *over $P \times \Gamma^*$.*

A configuration of an ABPDS is a pair $\langle p, \omega \rangle$, where $p \in P$ is a control location and $\omega \in \Gamma^*$ is the stack content. We assume w.l.o.g. that the boolean formulas are in disjunctive normal form. This allows to consider Δ as a subset of $(P \times \Gamma) \times 2^{P \times \Gamma^*}$. Thus, rules of Δ of the form[1] $\langle p, \gamma \rangle \hookrightarrow \bigvee_{j=1}^{n} \bigwedge_{i=1}^{m_j} \langle p_i^j, \omega_i^j \rangle$ can be denoted by the union of n rules of the form $\langle p, \gamma \rangle \hookrightarrow \{\langle p_1^j, \omega_1^j \rangle, ..., \langle p_{m_j}^j, \omega_{m_j}^j \rangle\}$, where $1 \leq j \leq n$. Let $t = \langle p, \gamma \rangle \hookrightarrow \{\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle\}$ be a rule of Δ. For every $\omega \in \Gamma^*$, the configuration $\langle p, \gamma\omega \rangle$ (resp. $\{\langle p_1, \omega_1\omega \rangle, ..., \langle p_n, \omega_n\omega \rangle\}$) is an immediate predecessor (resp. successor) of $\{\langle p_1, \omega_1\omega \rangle, ..., \langle p_n, \omega_n\omega \rangle\}$ (resp. $\langle p, \gamma\omega \rangle$).

A run ρ of $\mathcal{BP}$ from an initial configuration $\langle p_0, \omega_0 \rangle$ is a tree in which the root is labeled by $\langle p_0, \omega_0 \rangle$, and the other nodes are labeled by elements of $P \times \Gamma^*$. If a node of ρ is labeled by $\langle p, \omega \rangle$ and has n children labeled by $\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle$, respectively, then necessarily, $\langle p, \omega \rangle$ has $\{\langle p_1, \omega_1 \rangle, ..., \langle p_n, \omega_n \rangle\}$ as an immediate successor in $\mathcal{BP}$. A path $c_0c_1...$ of a run ρ is an *infinite* sequence of configurations such that c_0 is the root of ρ and for every $i \geq 0$, c_{i+1} is one of the children of the node c_i in ρ. The path is accepting from the initial configuration c_0 if and only if it visits infinitely often configurations with control locations in F. A run ρ is accepting if and only if all its paths are accepting. Note that an accepting run has only *infinite* paths; it does not involve finite paths. A configuration c is accepted (or recognized) by $\mathcal{BP}$ iff $\mathcal{BP}$ has an accepting run starting from c. The language of $\mathcal{BP}$, $\mathcal{L}(\mathcal{BP})$ is the set of configurations accepted by $\mathcal{BP}$.

The reachability relation $\Longrightarrow_{\mathcal{BP}} \subseteq (P \times \Gamma^*) \times 2^{P \times \Gamma^*}$ is the reflexive and transitive closure of the immediate successor relation. Formally $\Longrightarrow_{\mathcal{BP}}$ is defined as follows: (1) $c \Longrightarrow_{\mathcal{BP}} \{c\}$ for every $c \in P \times \Gamma^*$, (2) $c \Longrightarrow_{\mathcal{BP}} C$ if C is an immediate successor of c, (3) if $c \Longrightarrow_{\mathcal{BP}} \{c_1, ..., c_n\}$ and $c_i \Longrightarrow_{\mathcal{BP}} C_i$ for every $1 \leq i \leq n$, then $c \Longrightarrow_{\mathcal{BP}} \bigcup_{i=1}^{n} C_i$.

The functions $Pre_{\mathcal{BP}}$, $Pre_{\mathcal{BP}}^*$ and $Pre_{\mathcal{BP}}^+ : 2^{P \times \Gamma^*} \longrightarrow 2^{P \times \Gamma^*}$ are defined as follows: $Pre_{\mathcal{BP}}(C) = \{c \in P \times \Gamma^* \mid \exists C' \subseteq C$ s.t. C' is an immediate successor of $c\}$, (2) $Pre_{\mathcal{BP}}^*(C) = \{c \in P \times \Gamma^* | \exists C' \subseteq C \ s.t. \ c \Longrightarrow_{\mathcal{BP}} C'\}$, (3) $Pre_{\mathcal{BP}}^+(C) = Pre_{\mathcal{BP}} \circ Pre_{\mathcal{BP}}^*(C)$.

To represent (infinite) sets of configurations of ABPDSs, we use Alternating Multi-Automata:

Definition 4. *[1] Let $\mathcal{BP} = (P, \Gamma, \Delta, F)$ be an ABPDS. An Alternating Multi-Automaton* (AMA for short) *is a tuple $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$, where Q is a finite set of states that contains P, Γ is the input alphabet, $\delta \subseteq (Q \times \Gamma) \times 2^Q$ is a finite set of transition rules, $I \subseteq P$ is a finite set of initial states, $Q_f \subseteq Q$ is a finite set of final states.*

A Multi-Automaton (MA for short) *is an AMA such that $\delta \subseteq (Q \times \Gamma) \times Q$.*

We define the reflexive and transitive transition relation $\longrightarrow_\delta \subseteq (Q \times \Gamma^*) \times 2^Q$ as follows: (1) $q \xrightarrow{\epsilon}_\delta \{q\}$ for every $q \in Q$, where ϵ is the empty word, (2) $q \xrightarrow{\gamma}_\delta Q'$, if $q \xrightarrow{\gamma} Q' \in \delta$, (3) if $q \xrightarrow{\omega}_\delta \{q_1, ..., q_n\}$ and $q_i \xrightarrow{\gamma}_\delta Q_i$ for every $1 \leq i \leq n$, then $q \xrightarrow{\omega\gamma}_\delta \bigcup_{i=1}^{n} Q_i$. The automaton $\mathcal{A}$ recognizes a configuration $\langle p, \omega \rangle$ iff there exists $Q' \subseteq Q_f$ such that $p \xrightarrow{\omega}_\delta Q'$ and $p \in I$. The language of $\mathcal{A}$, $L(\mathcal{A})$, is the set of configurations recognized by $\mathcal{A}$. A set of configurations is regular if it can be recognized by an AMA. It is easy

[1] This rule represents $\Delta(p, \gamma) = \bigvee_{j=1}^{n} \bigwedge_{i=1}^{m_j} (p_i^j, \omega_i^j)$.

to show that AMAs are closed under boolean operations and that they are equivalent to MAs. Given an AMA, one can compute an equivalent MA by performing a kind of powerset construction as done for the determinisation procedure. Similarly, MAs can also be used to recognize (infinite) regular sets of configurations for PDSs.

Proposition 1. *Let $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ be an AMA. Deciding whether a configuration $\langle p, \omega \rangle$ is accepted by $\mathcal{A}$ can be done in $O(|Q| \cdot |\delta| \cdot |\omega|)$ time.*

3 Computing the Language of an ABPDS

Our goal in this section is to compute the set of accepting configurations of an Alternating Büchi PushDown System $\mathcal{BP} = (P, \Gamma, \Delta, F)$. We show that it is regular and that it can effectively be represented by an AMA. Determining whether $\mathcal{BP}$ has an accepting run is a non-trivial problem because a run of $\mathcal{BP}$ is an *infinite* tree with an infinite number of paths labelled by PDS configurations, which are control states and stack contents. All the paths of an accepting run are infinite and should all go through final control locations infinitely often. The difficulty comes from the fact that we cannot reason about the different paths of an ABPDS independently, we need to reason about *runs labeled with PDS configurations*. We proceed as follows: First, we characterize the set of configurations from which $\mathcal{BP}$ has an accepting run. Then, based on this characterization, we compute an AMA representing this set.

3.1 Characterizing $\mathcal{L}(\mathcal{BP})$

We give in this section a characterization of $\mathcal{L}(\mathcal{BP})$, i.e., the set of configurations from which $\mathcal{BP}$ has an accepting run. Let $(X_i)_{i \geq 0}$ be the sequence defined as follows: $X_0 = P \times \Gamma^*$ and $X_{i+1} = Pre^+(X_i \cap F \times \Gamma^*)$ for every $i \geq 0$. Let $Y_{\mathcal{BP}} = \bigcap_{i \geq 0} X_i$. We show that $\mathcal{L}(\mathcal{BP}) = Y_{\mathcal{BP}}$:

Theorem 1. *$\mathcal{BP}$ has an accepting run from a configuration $\langle p, \omega \rangle$ iff $\langle p, \omega \rangle \in Y_{\mathcal{BP}}$.*

To prove this result, we first show that:

Lemma 1. *$\mathcal{BP}$ has a run ρ from a configuration $\langle p, \omega \rangle$ such that each path of ρ visits configurations with control locations in F at least k times iff $\langle p, \omega \rangle \in X_k$.*

Intuitively, let c be a configuration in X_1. Since $X_1 = Pre^+(X_0 \cap F \times \Gamma^*)$, c has a successor C that is a subset of $F \times \Gamma^*$. Thus, $\mathcal{BP}$ has a run starting from c whose paths visit configurations with control locations in F at least once. Since $X_2 = Pre^+(X_1 \cap F \times \Gamma^*)$, it follows that from every configuration in X_2, $\mathcal{BP}$ has a run whose paths visit configurations in $X_1 \cap F \times \Gamma^*$ at least once, and thus, they visit configurations with control locations in F at least twice. We get by induction that for every $k \geq 1$, from every configuration c in X_k, $\mathcal{BP}$ has a run whose paths visit configurations with control locations in F at least k times. Since $Y_{\mathcal{BP}}$ is the set of configurations from which $\mathcal{BP}$ has a run that visits control locations in F infinitely often, Theorem 1 follows.

3.2 Computing $\mathcal{L}(\mathcal{BP})$

Our goal is to compute $Y_{\mathcal{BP}} = \bigcap_{i\geq 0} X_i$, where $X_0 = P \times \Gamma^*$ and for every $i \geq 0$, $X_{i+1} = Pre^+(X_i \cap F \times \Gamma^*)$. We provide a saturation procedure that computes the set $Y_{\mathcal{BP}}$. Our procedure is inspired from the algorithm given in [7] to compute the winning region of a Büchi game on a pushdown graph.

We show that $Y_{\mathcal{BP}}$ can be represented by an AMA $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ whose set of states Q is a subset of $P \times \mathbb{N} \cup \{q_f\}$, where q_f is a special state denoting the final state ($Q_f = \{q_f\}$). ¿From now on, for every $p \in P$ and $i \in \mathbb{N}$, we write p^i to denote (p, i).

Intuitively, to compute $Y_{\mathcal{BP}}$, we will compute iteratively the different X_i's by applying the saturation procedure of [1]. The iterative procedure computes different automata. The automaton computed during the iteration i uses states of the form p^i having i as index. To force termination, we use an acceleration criterion. For this, we need to define two projection functions π^{-1} and π^i defined as follows: For every $S \subseteq P \times \mathbb{N} \cup \{q_f\}$,

$$\pi^{-1}(S) = \begin{cases} \{q^i \mid q^{i+1} \in S\} \cup \{q_f\} \text{ if } q_f \in S \text{ or } \exists q^1 \in S, \\ \{q^i \mid q^{i+1} \in S\} \qquad\qquad \text{else.} \end{cases}$$

$$\pi^i(S) = \{q^i \mid \exists 1 \leq j \leq i \; s.t. \; q^j \in S\} \cup \{q_f \mid q_f \in S\}.$$

The AMA $\mathcal{A}$ is computed iteratively using **Algorithm 1**:

```
Algorithm 1: Computation of $Y_{\mathcal{BP}}$
 Input:   An ABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$.
 Output:  An AMA $\mathcal{A} = (Q, \Gamma, \delta, I, Q_f)$ that recognizes $Y_{\mathcal{BP}}$.
 1.Initially: Let $i = 0, \delta = \{(q_f, \gamma, \{q_f\})$ for every $\gamma \in \Gamma\}$, and for every control state $p \in P, p^0 = q_f$.
 2.   Repeat (we call this loop $loop_1$)
 3.      $i := i + 1$;
 4.      Add in $\delta$ a new transition rule $p^i \xrightarrow{\epsilon} p^{i-1}$, for every $p \in F$;
 5.      Repeat (we call this loop $loop_2$)
 6.         For every $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, \omega_1\rangle, ..., \langle p_n, \omega_n\rangle\}$ in $\Delta$
 7.         and every case where $p_k^i \xrightarrow{\omega_k}_\delta Q_k$, for every $1 \leq k \leq n$;
 8.            Add a new rule $p^i \xrightarrow{\gamma} \bigcup_{k=1}^n Q_k$ in $\delta$;
 9.      Until No new transition rule can be added.
 10.     Remove from $\delta$ the transition rules $p^i \xrightarrow{\epsilon} p^{i-1}$, for every $p \in F$;
 11.     Replace in $\delta$ every transition rule $p^i \xrightarrow{\gamma} R$ by $p^i \xrightarrow{\gamma} \pi^i(R)$, for every $p \in P, \gamma \in \Gamma, R \subseteq Q$;
 12.  Until $i > 1$ and for every $p \in P, \gamma \in \Gamma, R \subseteq P \times \{i\} \cup \{q_f\}$; $p^i \xrightarrow{\gamma} R \in \delta \Longleftrightarrow p^{i-1} \xrightarrow{\gamma} \pi^{-1}(R) \in \delta$
```

Let us explain the intuition behind the different lines of this algorithm. Let A_i be the automaton obtained at step i (a step starts at Line 3). For every $p \in P$, the state p^i is meant to represent state p at step i, i.e., A_i recognizes a configuration $\langle p, \omega\rangle$ iff $p^i \xrightarrow{\omega}_\delta q_f$. Let A_0 be the automaton obtained after the initialization step (Line 1). It is clear that A_0 recognizes $X_0 = P \times \Gamma^*$. Suppose now that the algorithm is at the beginning of the i-th iteration ($loop_1$). Line 4 adds the ϵ-transition $p^i \xrightarrow{\epsilon} p^{i-1}$ for every control state $p \in F$. After this step, we obtain $L(A_{i-1}) \cap F \times \Gamma^*$. $loop_2$ at lines $5-9$ is the saturation procedure of [1]. It computes the Pre^* of $L(A_{i-1}) \cap F \times \Gamma^*$. Line 10 removes the ϵ-transition added by Line 4. After this step, the automaton recognizes $Pre^+(L(A_{i-1}) \cap F \times \Gamma^*)$, i.e., X_i. Let us call **Algorithm B** the above algorithm without Line 11. It follows from the explanation above that if **Algorithm B** terminates, it will produce $Y_{\mathcal{BP}}$. However, this procedure will never terminate if the sequence (X_i) is strictly decreasing. Consider for example the ABPDS $\mathcal{BP} = (\{q\}, \{\gamma\}, \Delta, \{q\})$, where $\Delta = \{\langle q, \gamma\rangle \hookrightarrow \langle q, \epsilon\rangle\}$. Then, for every $i \geq 0$, $X_i = \{\langle q, \gamma^i\omega\rangle \mid \omega \in \gamma^*\}$. It is clear that **Algorithm B** will never terminate on this example.

The substitution at Line 11 is the acceleration used to force the termination of the algorithm, tested at Line 12. We can show that thanks to Line 11 and to the test of Line 12, our algorithm always terminates and produces $Y_{\mathcal{BP}}$:

Theorem 2. *Algorithm 1 always terminates and produces $Y_{\mathcal{BP}}$.*

Proof (Sketch): Termination. Let us first prove the termination of our procedure. Note that due to the substitution of Line 11, at the end of step i, states with index $j < i$ are not useful and can be removed. We can then suppose that at the end of step i, the automaton A_i uses only states of index i (in addition to state q_f). Thus, the termination tested at Line 12 holds when at step i, the transitions of A_i are "the same" than those of A_{i-1}.

We can show that at each step i, $loop_2$ (corresponding to the saturation procedure) adds less transitions than at step $i-1$, meaning that A_i has less transitions than A_{i-1}. Intuitively, this is due to the fact that at step i, we obtain after the saturation procedure $Pre^+(L(A_{i-1}) \cap F \times \Gamma^*)$. Since Pre^+ is monotonic, and since we start at step 0 with an automaton A_0 that recognizes all the configurations $P \times \Gamma^*$, we get that for $i > 0$, $L(A_i) \subseteq L(A_{i-1})$. More precisely, we can show by induction on i that:

Proposition 2. *In **Algorithm** 1, for every $\gamma \in \Gamma$, $p \in P, S \subseteq Q$; at each step $i \geq 2$, if $p^i \xrightarrow{\gamma} S \in \delta$, then $p^{i-1} \xrightarrow{\gamma} \pi^{-1}(\pi^i(S)) \in \delta$.*

Thus, the substitution of Line 11 guarantees that at each step, the number of transitions of the automaton A_i is less than the number of transitions of A_{i-1}. Since the number of transitions that can be added at each step is finite, and since the termination criterion of Line 12 holds if the transitions of A_i are "the same" than those of A_{i-1}, the termination of our algorithm is guaranteed.

Correctness. Let us now prove that our algorithm is correct, i.e., it produces $Y_{\mathcal{BP}}$. As mentionned previously, without Line 11, the algorithm above would have computed the different X_i's. Since $Y_{\mathcal{BP}} = \bigcap_{i\geq 0} X_i$, we need to show that Line 11 does not introduce new configurations that are not in $Y_{\mathcal{BP}}$, nor remove ones that should be in $Y_{\mathcal{BP}}$.

Suppose we are at step i, and let $p \in P$, $\gamma \in \Gamma$, and $R \subseteq Q$ be such that Line 11 adds the transition $p^i \xrightarrow{\gamma} \pi^i(R)$ and removes the transition $p^i \xrightarrow{\gamma} R$. This substitution adds a new transition iff R contains at least one state of the form q^{i-1} (otherwise, $\pi^i(R) = R$ and Line 11 does not introduce any change for this transition). Let then $S \subseteq Q$ be such $R = S \cup \{q^{i-1}\}$. Let us first show that this substitution does not introduce new configurations. Let $u \in \Gamma^*$ such that $p^i \xrightarrow{\gamma}_\delta \pi^i(R) \xrightarrow{u}_\delta q_f$ is a new accepting run of the automaton. Then, due to Proposition 2, we can show that there exists already (before the substitution) a run $p^i \xrightarrow{\gamma}_\delta R \xrightarrow{u}_\delta q_f$ in the automaton that accepts the configuration $\langle p, \gamma u\rangle$.

Let us now show that the substitution above does not remove configurations that are in $Y_{\mathcal{BP}}$. Let $\langle p, \omega \rangle$ be a configuration removed by the substitution above, i.e., $\langle p, \omega \rangle$ is no more recognized by A_i due to the fact that $p^i \xrightarrow{\gamma} R$ is removed. We show that $\langle p, \omega \rangle$ cannot be in $Y_{\mathcal{BP}}$. Let $v \in \Gamma^*$ such that $\omega = \gamma v$ and $\rho = p^i \xrightarrow{\gamma}_\delta q^{i-1} \cup S \xrightarrow{v}_\delta \{q_f\}$ is a run accepting $\langle p, \omega \rangle$ whereas there is no run of the form $q^i \xrightarrow{v}_\delta \{q_f\}$. Suppose for simplicity that ρ is the only run recognizing $\langle p, \omega \rangle$, the same reasoning can also be applied if this is not the case. Since $p^i \xrightarrow{\gamma} q^{i-1} \cup S$, we can show that there exist states $q_1, \ldots, q_n$, and words $\omega_1, \ldots, \omega_n$ such that $\langle p, \gamma \rangle \Longrightarrow_{\mathcal{BP}} \{\langle q, \epsilon \rangle, \langle q_1, \omega_1 \rangle, \cdots \langle q_n, \omega_n \rangle\}$. Then, due to the fact that $\langle p, \omega \rangle$ is removed from the automaton and that ρ is the only path accepting $\langle p, \omega \rangle$, we can show that all the possible runs from the configuration $\langle p, \omega \rangle$ go through the configuration $\langle q, v \rangle$. Since $\langle q, v \rangle \notin Y_{\mathcal{BP}}$ (because there is no run of the form $q^i \xrightarrow{v}_\delta \{q_f\}$), $\mathcal{BP}$ has no accepting run from the configuration $\langle q, v \rangle$. It follows that $\mathcal{BP}$ cannot have an accepting run from $\langle p, \omega \rangle$. □

Complexity: Given an AMA A with n states, [23] provides a procedure that can implement the saturation procedure $loop_2$ to compute the Pre^* of A in time $O(n \cdot |\Delta| \cdot 2^{2n})$. Since at each step i, **Algorithm 1** needs to consider only states of the form p^i and p^{i-1} (in addition to q_f), the number of states at each step i should be $2|P| + 1$. Thus, $loop_2$ can be done in $O(|P| \cdot |\Delta| \cdot 2^{4|P|})$. Furthermore, Line 11 and the termination condition are done in time $O(|\Gamma| \cdot |P| \cdot 2^{2|P|})$ and $O(|\Gamma| \cdot |P| \cdot 2^{|P|})$, respectively. We know that the number of transition rules of A_i is less than those of A_{i-1}. Since the number of transition rules of the AMA is at most $|\Gamma| \cdot |P| \cdot 2^{|P|+1}$, $loop_1$ can be done at most $|\Gamma| \cdot |P| \cdot 2^{|P|+1}$ times. Putting all these estimations together, the algorithm runs in $O(|P|^2 \cdot |\Delta| \cdot |\Gamma| \cdot 2^{5|P|})$ time.

Thus, since $\mathcal{L}(\mathcal{BP}) = Y_{\mathcal{BP}}$, we get :

Theorem 3. *Given an ABPDS $\mathcal{BP} = (P, \Gamma, \Delta, F)$, we can effectively compute an AMA $\mathcal{A}$ with $O(|P|)$ states and $O(|P| \cdot |\Gamma| \cdot 2^{|P|})$ transition rules that recognizes $\mathcal{L}(\mathcal{BP})$. This AMA can be computed in time $O(|P|^2 \cdot |\Delta| \cdot |\Gamma| \cdot 2^{5|P|})$.*

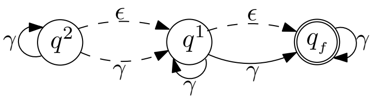


Fig. 2. The result automaton.

Example: Let us illustrate our algorithm by an example. Consider an ABPDS $\mathcal{BP} = (\{q\}, \{\gamma\}, \Delta, \{q\})$, where $\Delta = \{\langle q, \gamma \rangle \hookrightarrow \langle q, \epsilon \rangle\}$. The automaton produced by **Algorithm** 1 is shown in Figure 2. The dashed lines denote the transitions removed by Lines 10 and 11. In the first iteration, $t_1 = q^1 \xrightarrow{\epsilon} q_f$ is added by Line 4, the saturation procedure (lines 5 – 9) adds two transitions $q^1 \xrightarrow{\gamma} q_f$ and $q^1 \xrightarrow{\gamma} q^1$. Then the transition t_1 is removed by Line 10. In the second iteration, $t_2 = q^2 \xrightarrow{\epsilon} q^1$ is added by Line 4. The saturation procedure adds the transitions $t_3 = q^2 \xrightarrow{\gamma} q^1$ and $q^2 \xrightarrow{\gamma} q^2$. Finally, t_2 is removed by Line 10 and t_3 is replaced by $q^2 \xrightarrow{\gamma} q^2$ (this transition already exists in the automaton). Now the termination condition is satisfied and the algorithm terminates. In this case, $\mathcal{BP}$ has no accepting run.

Efficient implementation of Algorithm 1. We show that we can improve the complexity of **Algorithm** 1 as follows:

Improvement 1. For every $q \in Q$ and $\gamma \in \Gamma$, if $t_1 = q \xrightarrow{\gamma} Q_1$ and $t_2 = q \xrightarrow{\gamma} Q_2$ are two transitions in δ such that $Q_1 \subseteq Q_2$, then remove t_2. This means that if $\mathcal{A}$ contains two transitions $t_1 = p \xrightarrow{\gamma} \{q_1, q_2, q_3\}$ and $t_2 = p \xrightarrow{\gamma} \{q_1, q_2\}$, then we can remove t_1 without changing the language of $\mathcal{A}$. Indeed, if a path $q \xrightarrow{\omega}_{\delta} q_f$ uses the transition rule t_1, then there must be necessarily a path $q \xrightarrow{\omega}_{\delta} q_f$ that uses the transition rule t_2 instead of t_1.

Improvement 2. Each transition $q^i \xrightarrow{\gamma} R$ added by the saturation procedure will be substituted by $q^i \xrightarrow{\gamma} \pi^i(R)$ in Line 11. Transitions of the form $q^i \xrightarrow{\gamma} \{q_1^i, q_1^{i-1}\} \cup R$ and $q^i \xrightarrow{\gamma} \{q_1^{i-1}\} \cup R$ have the same substitution $q^i \xrightarrow{\gamma} \{q_1^i\} \cup \pi^i(R)$. We show that each transition $q^i \xrightarrow{\gamma} \{q_1^i, q_1^{i-1}\} \cup R$ can be replaced by $q^i \xrightarrow{\gamma} \{q_1^{i-1}\} \cup R$ in the saturation procedure (i.e., during $loop_2$). Moreover, we show that if both $t_1 = q^i \xrightarrow{\gamma} \{q_1^{i-1}, ..., q_n^{i-1}\} \cup R$ and $t_2 = q^i \xrightarrow{\gamma} \{q_1^i, ..., q_n^i\} \cup R$ exist during $loop_2$, then t_2 can be removed. This is due to the fact that they both have the same substitution rule.

4 CTL Model-Checking for PushDown Systems

We consider in this section "standard" CTL model checking for pushdown systems as considered in the literature, i.e., the case where whether an atomic proposition holds for a given configuration c or not depends only on the control state of c, not on its stack. Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a pushdown system, c_0 its initial configuration, AP a set of atomic propositions, φ a CTL formula, $f : AP \to 2^P$ a function that associates atomic propositions to sets of control states, and $\lambda_f : AP \to 2^{P\times\Gamma^*}$ a labelling function such that for every $a \in AP$, $\lambda_f(a) = \{\langle p, \omega\rangle \mid p \in f(a), \omega \in \Gamma^*\}$. We provide in this section an algorithm to determine whether $(\mathcal{P}, c_0) \models_{\lambda_f} \varphi$. We proceed as follows: Roughly speaking, we compute an Alternating Büchi PushDown System $\mathcal{BP}$ that recognizes the set of configurations c such that $(\mathcal{P}, c) \models_{\lambda_f} \varphi$. Then $(\mathcal{P}, c_0) \models_{\lambda_f} \varphi$ holds iff $c_0 \in \mathcal{L}(\mathcal{BP})$. This can be effectively checked due to Theorem 3 and Proposition 1.

Let $\mathcal{BP}_\varphi = (P', \Gamma, \Delta', F)$ be the ABPDS defined as follows: $P' = P \times cl(\varphi)$; $F = \{[p, a] \mid a \in cl(\varphi) \cap AP$ and $p \in f(a)\} \cup \{[p, \neg a] \mid \neg a \in cl(\varphi), a \in AP$ and $p \notin f(a)\} \cup P \times cl_{\tilde{U}}(\varphi)$, where $cl_{\tilde{U}}(\varphi)$ is the set of formulas of $cl(\varphi)$ of the form $E[\varphi_1 \tilde{U} \varphi_2]$ or $A[\varphi_1 \tilde{U} \varphi_2]$; and Δ' is the smallest set of transition rules such that for every control location $p \in P$, every subformula $\psi \in cl(\varphi)$, and every $\gamma \in \Gamma$, we have:

1. if $\psi = a, a \in AP$ and $p \in f(a)$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi], \gamma\rangle \in \Delta'$,
2. if $\psi = \neg a, a \in AP$ and $p \notin f(a)$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi], \gamma\rangle \in \Delta'$,
3. if $\psi = \psi_1 \wedge \psi_2$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_1], \gamma\rangle \wedge \langle [p, \psi_2], \gamma\rangle \in \Delta'$,
4. if $\psi = \psi_1 \vee \psi_2$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_1], \gamma\rangle \vee \langle [p, \psi_2], \gamma\rangle \in \Delta'$,
5. if $\psi = EX\psi_1$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \bigvee_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} \langle [p', \psi_1], \omega\rangle \in \Delta'$,
6. if $\psi = AX\psi_1$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \bigwedge_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} \langle [p', \psi_1], \omega\rangle \in \Delta'$,
7. if $\psi = E[\psi_1 U \psi_2]$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_2], \gamma\rangle \vee \bigvee_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} (\langle [p, \psi_1], \gamma\rangle \wedge \langle [p', \psi], \omega\rangle) \in \Delta'$,
8. if $\psi = A[\psi_1 U \psi_2]$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_2], \gamma\rangle \vee \bigwedge_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} (\langle [p, \psi_1], \gamma\rangle \wedge \langle [p', \psi], \omega\rangle) \in \Delta'$,
9. if $\psi = E[\psi_1 \tilde{U} \psi_2]$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_2], \gamma\rangle \wedge (\langle [p, \psi_1], \gamma\rangle \vee \bigvee_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} \langle [p', \psi], \omega\rangle) \in \Delta'$,
10. if $\psi = A[\psi_1 \tilde{U} \psi_2]$; $\langle [p, \psi], \gamma\rangle \hookrightarrow \langle [p, \psi_2], \gamma\rangle \wedge (\langle [p, \psi_1], \gamma\rangle \vee \bigwedge_{\langle p,\gamma\rangle\hookrightarrow\langle p',\omega\rangle\in\Delta} \langle [p', \psi], \omega\rangle) \in \Delta'$.

The ABPDS $\mathcal{BP}_\varphi$ above can be seen as the "product" of $\mathcal{P}$ with the formula φ. Intuitively, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[p,\psi],\omega\rangle$ if and only if the configuration $\langle p,\omega\rangle$ satisfies ψ. Let us explain the intuition behind the different items defining Δ'.

Let $\psi = a \in AP$. If $p \in f(a)$ then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ. Thus, $\mathcal{BP}_\varphi$ should accept $\langle[p,a],\omega\rangle$, i.e., have an accepting run from $\langle[p,a],\omega\rangle$. This is ensured by Item 1 that adds a loop in $\langle[p,a],\omega\rangle$, and the fact that $[p,a] \in F$.

Let $\psi = \neg a$, where $a \in AP$. If $p \notin f(a)$ then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ. Thus, $\mathcal{BP}_\varphi$ should accept $\langle[p,\neg a],\omega\rangle$, i.e., have an accepting run from $\langle[p,\neg a],\omega\rangle$. This is ensured by Item 2 and the fact that $[p,\neg a] \in F$.

Item 3 expresses that if $\psi = \psi_1 \wedge \psi_2$, then for every $\omega \in \Gamma^*$, $\mathcal{BP}_\varphi$ has an accepting run from $\langle[p,\psi_1 \wedge \psi_2],\omega\rangle$ iff $\mathcal{BP}_\varphi$ has an accepting run from $\langle[p,\psi_1],\omega\rangle$ and $\langle[p,\psi_2],\omega\rangle$; meaning that $\langle p,\omega\rangle$ satisfies ψ iff $\langle p,\omega\rangle$ satisfies ψ_1 and ψ_2. Item 4 is similar to Item 3.

Item 5 means that if $\psi = EX\psi_1$, then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ iff there exists an immediate sucessor $\langle p',\omega'\rangle$ of $\langle p,\omega\rangle$ such that $\langle p',\omega'\rangle$ satisfies ψ_1. Thus, $\mathcal{BP}_\varphi$ should have an accepting run from $\langle[p,\psi],\omega\rangle$ iff it has an accepting run from $\langle[p',\psi_1],\omega'\rangle$. Similarly, item 6 states that if $\psi = AX\psi_1$, then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ iff $\langle p',\omega'\rangle$ satisfies ψ_1 for every immediate sucessor $\langle p',\omega'\rangle$ of $\langle p,\omega\rangle$.

Item 7 expresses that if $\psi = E[\psi_1 U \psi_2]$, then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ iff either it satisfies ψ_2, or it satisfies ψ_1 and there exists an immediate sucessor $\langle p',\omega'\rangle$ of $\langle p,\omega\rangle$ such that $\langle p',\omega'\rangle$ satisfies ψ. Item 8 is similar to Item 7.

Item 9 expresses that if $\psi = E[\psi_1 \tilde{U} \psi_2]$, then for every $\omega \in \Gamma^*$, $\langle p,\omega\rangle$ satisfies ψ iff it satisfies ψ_2, and either it satisfies also ψ_1, or it has a successor that satisfies ψ. This guarantees that ψ_2 holds either always, or until both ψ_1 and ψ_2 hold. The fact that the state $[p,\psi]$ is in F ensures that paths where ψ_2 always hold are accepting. The intuition behind Item 10 is analogous.

Formally, we can show that:

Theorem 4. *Let $\mathcal{P} = (P,\Gamma,\Delta,\sharp)$ be a PDS, $f : AP \longrightarrow 2^P$ a labelling function, φ a CTL formula, and $\langle p,\omega\rangle$ a configuration of $\mathcal{P}$. Let $\mathcal{BP}_\varphi$ be the ABPDS computed above. Then, $(\mathcal{P},\langle p,\omega\rangle) \models_{\lambda_f} \varphi$ iff $\mathcal{BP}_\varphi$ has an accepting run from the configuration $\langle[p,\varphi],\omega\rangle$.*

It follows from Theorems 3 and 4 that:

Corollary 1. *Given a PDS $\mathcal{P} = (P,\Gamma,\Delta,\sharp)$, a labeling function $f : P \longrightarrow 2^{AP}$, and a CTL formula φ, we can construct an AMA $\mathcal{A}$ in time $O(|P|^2 \cdot |\varphi|^3 \cdot (|P| \cdot |\Gamma| + |\Delta|) \cdot |\Gamma| \cdot 2^{5|P||\varphi|})$ such that for every configuration $\langle p,\omega\rangle$ of $\mathcal{P}$, $(\mathcal{P},\langle p,\omega\rangle) \models_{\lambda_f} \varphi$ iff the AMA $\mathcal{A}$ recognizes the configuration $\langle[p,\varphi],\omega\rangle$.*

The complexity follows from the complexity of **Algorithm 1** and the fact that $\mathcal{BP}_\varphi$ has $O(|P||\varphi|)$ states and $O((|P||\Gamma| + |\Delta|)|\varphi|)$ transitions.

5 CTL Model-Checking for PushDown Systems with Regular Valuations

So far, we considered the "standard" model-checking problem for CTL, where the validity of an atomic proposition in a configuration c depends only on the control state of

c, not on the stack. In this section, we go further and consider an extension where the set of configurations in which an atomic proposition holds is a regular set of configurations.

Let $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$ be a pushdown system, c_0 its initial configuration, AP a set of atomic propositions, φ a CTL formula, and $\lambda : AP \to 2^{P \times \Gamma^*}$ a labelling function such that for every $a \in AP$, $\lambda(a)$ is a regular set of configurations. We say that λ is a regular labelling. We give in this section an algorithm that checks whether $(\mathcal{P}, c_0) \models_\lambda \varphi$. We proceed as previously: Roughly speaking, we compute an ABPDS $\mathcal{BP}'_\varphi$ such that $\mathcal{BP}'_\varphi$ recognizes a configuration c iff $(\mathcal{P}, c) \models_\lambda \varphi$. Then $(\mathcal{P}, c_0)$ satisfies φ iff c_0 is accepted by $\mathcal{BP}'_\varphi$. As previously, this can be checked using Theorem 3 and Proposition 1.

For every $a \in AP$, since $\lambda(a)$ is a regular set of configurations, let $M_a = (Q_a, \Gamma, \delta_a, I_a, F_a)$ be a multi-automaton such that $L(M_a) = \lambda(a)$, and $M_{\neg a} = (Q_{\neg a}, \Gamma, \delta_{\neg a}, I_{\neg a}, F_{\neg a})$ such that $L(M_{\neg a}) = P \times \Gamma^* \setminus \lambda(a)$ be a multi-automaton that recognizes the complement of $\lambda(a)$, i.e., the set of configurations where a does not hold. Since for every $a \in AP$ and every control state $p \in P$, p is an initial state of Q_a and $Q_{\neg a}$; to distinguish between all these initial states, for every $a \in AP$, we will denote in the following the initial state corresponding to p in Q_a (resp. in $Q_{\neg a}$) by p_a (resp. $p_{\neg a}$).

Let $\mathcal{BP}'_\varphi = (P'', \Gamma, \Delta'', F')$ be the ABPDS defined as follows[2]: $P'' = P \times cl(\varphi) \cup \bigcup_{a \in AP^+(\varphi)} Q_a \cup \bigcup_{a \in AP^-(\varphi)} Q_{\neg a}$; $F' = P \times cl_{\tilde{U}}(\varphi) \cup \bigcup_{a \in AP^+(\varphi)} F_a \cup \bigcup_{a \in AP^-(\varphi)} F_{\neg a}$; and Δ'' is the smallest set of transition rules such that for every control location $p \in P$, every subformula $\psi \in cl(\varphi)$, and every $\gamma \in \Gamma$, we have:

1. if $\psi = a$, $a \in AP$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle p_a, \gamma \rangle \in \Delta''$,
2. if $\psi = \neg a$, $a \in AP$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle p_{\neg a}, \gamma \rangle \in \Delta''$,
3. if $\psi = \psi_1 \wedge \psi_2$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_1], \gamma \rangle \wedge \langle [p, \psi_2], \gamma \rangle \in \Delta''$,
4. if $\psi = \psi_1 \vee \psi_2$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_1], \gamma \rangle \vee \langle [p, \psi_2], \gamma \rangle \in \Delta''$,
5. if $\psi = EX\psi_1$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \bigvee_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} \langle [p', \psi_1], \omega \rangle \in \Delta''$,
6. if $\psi = AX\psi_1$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \bigwedge_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} \langle [p', \psi_1], \omega \rangle \in \Delta''$,
7. if $\psi = E[\psi_1 U \psi_2]$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_2], \gamma \rangle \vee \bigvee_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} (\langle [p, \psi_1], \gamma \rangle \wedge \langle [p', \psi], \omega \rangle) \in \Delta''$,
8. if $\psi = A[\psi_1 U \psi_2]$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_2], \gamma \rangle \vee \bigwedge_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} (\langle [p, \psi_1], \gamma \rangle \wedge \langle [p', \psi], \omega \rangle) \in \Delta''$,
9. if $\psi = E[\psi_1 \tilde{U} \psi_2]$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_2], \gamma \rangle \wedge (\langle [p, \psi_1], \gamma \rangle \vee \bigvee_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} \langle [p', \psi], \omega \rangle) \in \Delta''$,
10. if $\psi = A[\psi_1 \tilde{U} \psi_2]$; $\langle [p, \psi], \gamma \rangle \hookrightarrow \langle [p, \psi_2], \gamma \rangle \wedge (\langle [p, \psi_1], \gamma \rangle \vee \bigwedge_{\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle \in \Delta} \langle [p', \psi], \omega \rangle) \in \Delta''$.

Moreover:

11. for every transition $q_1 \xrightarrow{\gamma} q_2$ in $(\bigcup_{a \in AP^+(\varphi)} \delta_a) \cup (\bigcup_{a \in AP^-(\varphi)} \delta_{\neg a})$; $\langle q_1, \gamma \rangle \hookrightarrow \langle q_2, \epsilon \rangle \in \Delta''$,
12. for every $q \in (\bigcup_{a \in AP^+(\varphi)} F_a) \cup (\bigcup_{a \in AP^-(\varphi)} F_{\neg a})$; $\langle q, \sharp \rangle \hookrightarrow \langle q, \sharp \rangle \in \Delta''$.

The ABPDS $\mathcal{BP}'_\varphi$ has an accepting run from $\langle [p, \psi], \omega \rangle$ if and only if the configuration $\langle p, \omega \rangle$ satisfies ψ according to the regular labellings M_a's. Let us explain the intuition behind the rules above. Let $p \in P$, $\psi = a \in AP$, and $\omega \in \Gamma^*$. The ABPDS $\mathcal{BP}'_\varphi$ should accept $\langle [p, a], \omega \rangle$, iff $\langle p, \omega \rangle \in L(M_a)$. To check this, $\mathcal{BP}'_\varphi$ goes to state p_a, the initial state corresponding to p in M_a (Item 1); and then, from this state, it checks whether ω is accepted by M_a. This is ensured by Items 11 and 12. Item 11 allows $\mathcal{BP}'_\varphi$ to mimic a run of M_a on ω: if $\mathcal{BP}'_\varphi$ is in state q_1 with γ on top of its stack, and if $q_1 \xrightarrow{\gamma} q_2$ is a rule in δ_a, then $\mathcal{BP}'_\varphi$ moves to state q_2 while popping γ from the stack.

[2] $AP^+(\varphi)$ and $AP^-(\varphi)$ are as defined in Section 2.1.

Popping γ allows to check the rest of the word. The configuration is accepted if the run (with label ω) in M_a reaches a final state, i.e., if $\mathcal{BP}'_\varphi$ reaches a state $q \in F_a$ with an empty stack, i.e., a stack containing only the bottom stack symbol $\sharp$. Thus, F_a is in F''. Since all the accepting runs of $\mathcal{BP}'_\varphi$ are infinite, we add a loop on every configuration in control state $q \in F_a$ and having $\sharp$ as content of the stack (Item 12).

The intuition behind Item 2 is similar. This item applies for ψ of the from $\neg a$. Items 3–10 are similar to Items 3–10 in the construction underlying Theorem 4. We get:

Theorem 5. *$(\mathcal{P}, \langle p, \omega\rangle) \models_\lambda \varphi$ iff $\mathcal{BP}'_\varphi$ has an accepting run from the configuration $\langle [p, \varphi], \omega\rangle$.*

From this theorem and Theorem 3, it follows that:

Corollary 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta, \sharp)$, a regular labelling function λ, and a CTL formula φ, we can construct an AMA $\mathcal{A}$ such that for every configuration $\langle p, \omega\rangle$ of $\mathcal{P}$, $(\mathcal{P}, \langle p, \omega\rangle) \models_\lambda \varphi$ iff the AMA $\mathcal{A}$ recognizes the configuration $\langle [p, \varphi], \omega\rangle$. This AMA can be computed in time $O(|P|^3 \cdot |\Gamma|^2 \cdot |\varphi|^3 \cdot k^2 \cdot |\Delta| \cdot d \cdot 2^{5(|P||\varphi|+k)})$, where $k = \sum_{a\in \mathrm{AP}^+(\varphi)} |Q_a| + \sum_{a\in \mathrm{AP}^-(\varphi)} |Q_{\neg a}|$ and $d = \sum_{a\in \mathrm{AP}^+(\varphi)} |\delta_a| + \sum_{a\in \mathrm{AP}^-(\varphi)} |\delta_{\neg a}|$.*

The complexity follows from the complexity of **Algorithm 1** and the fact that $\mathcal{BP}'_\varphi$ has $O(|P||\varphi| + k)$ states and $O((|P||\Gamma| + |\Delta|)|\varphi| + d)$ transitions.

Remark 1. Note that to improve the complexity, we represent the regular valuations M_a's using AMAs instead of MAs. This prevents the exponential blow-up when complementing these automata to compute $M_{\neg a}$.

6 Experiments

We implemented all the algorithms presented in the previous sections in a tool. As far as we know, this is the first tool for CTL model-checking for PDSs. We applied our tool to the verification of sequential programs. Indeed, PDSs are well adapted to model sequential (possibly recursive) programs [10,13]. We carried out several experiments. We obtained interesting results. In particular, we were able to find bugs in linux drivers. Our results are reported in Figure 3. **Column** *formula size* gives the size of the formula. **Column** *time(s)* and *mem(kb)* give the time (in seconds) and memory (in kb). **Column** *Recu.* gives the number of iterations of $loop_1$. The last **Column** *result* gives the result whether the formula is satisfied or not (Y is satisfied, otherwise N). The first eleven lines of the table describe experiments done to evaluate **Algorithm 1.** that computes the set of configurations from which an ABPDS has an accepting run. The second part of the table describes experiments for "standard" CTL model-checking in which most of the specifications cannot be expressed in LTL. The last part considers CTL model-checking with regular valuations.

Plotter controls a plotter that creates random bar graphs [21]. We checked three CTL properties for this example (**Plotter1**, **Plotter2** and **Plotter3**). **ATM** is an automatic teller machine controller. We checked that if the pincode is correct, then the ATM will provide money (**ATM1**), and otherwise, it will set an alarm (**ATM2**). **ATM3** checks that the ATM gives the money only if the pincode is correct, and if it is accessed from

Examples		$\|P\| + \|\Gamma\|$ $+\|\Delta\|$	Formula size	Recu	Time(s)	Mem(kb)	Result
Algorithm 1	1	3+3+4	-	3	0	22.34	Y
	2	17+5+24	-	4	0	33.23	N
	3	73+5+73	-	4	0.02	128.28	Y
	4	75+6+75	-	5	0.02	81.36	N
	5	3+4+4	-	4	0	22.36	N
	6	3+4+5	-	3	0	21.54	Y
	7	3+4+4	-	3	0	20.11	Y
	8	3+4+4	-	4	0	27.40	Y
	9	74+6+76	-	5	0.02	87.54	Y
	10	17+5+24	-	3	0	28.46	Y
	11	18+5+28	-	3	0	26.15	Y
Standard	Plotter.1	1+19+24	2	3	0.02	41.56	Y
	Plotter.2	1+19+24	2	3	0	43.52	N
	Plotter.3	1+19+24	14	9	0.03	241.32	Y
	ATM.1	2+18+45	8	6	0.03	169.64	Y
	ATM.2	2+18+45	10	6	0.03	192.53	Y
	Lock.1	6+37+82	7	11	0.11	387.15	Y
	Lock.2	6+37+82	7	11	0.11	379.46	N
	Lock-err	6+37+82	3	9	0.00	186.52	N
	M-WO.1	1+7+12	6	2	0	40.20	Y
	M-WO.2	1+7+12	6	7	0	37.28	N
	File.1	1+5+9	2	3	0	34.77	Y
	File.2	1+5+9	2	4	0.02	32.51	N
	W.G.C.	16+1+40	23	2	0.05	202.01	Y
	btrfs/file.c	2+14+20	3	10	0	64.32	N
	btrfs/file.c-fixed	2+15+22	3	9	0.02	82.52	Y
	bluetooth	32+12+294	5	8	0.12	821.03	N
	w83627ehf	1+20+20	5	9	0.02	132.76	N
	w83627ehf-fixed	1+21+22	5	4	0.03	121.69	Y
	w83697ehf	1+56+57	6	11	0.35	394.61	Y
	advantech	2+16+31	7	6	0.05	120.41	Y
	at91rm9200	4+15+64	7	5	0.06	234.42	N
	at91rm9200-fixed	4+16+67	7	6	0.12	255.62	Y
	at32ap700x	4+25+105	7	8	0.15	356.04	N
	at32ap700x-fixed	4+25+109	7	9	0.22	334.42	Y
	pcf857x	1+98+106	10	18	0.23	541.35	Y
Regular Valuation	ATM.3	2+18+45	8	6	0.20	352.47	Y
	File.3	1+5+9	5	5	0	33.21	Y
	RSM1	1+8+11	25	4	0.06	438.23	Y
	RSM2	1+8+12	30	4	0.48	1231.45	Y
	RSM3	1+11+17	45	4	12.11	6206.73	Y
	RSM4	1+11+18	45	4	0.72	1269.26	Y
	RSM5	1+11+16	35	4	12.14	6212.2	Y
	ieee1394_core_1	1+104+108	12	14	0.20	413.69	Y
	ieee1394_core_2	1+104+108	13	14	0.19	422.17	Y
	ieee1394_core_3	1+104+108	14	17	0.19	438.42	N
	ieee1394_core_4	1+104+109	14	14	0.19	414.27	Y

Fig. 3. The performance of our tool

the main session. Regular valuations are needed to express this property. **Lock** is a lock-unlock program. We checked different properties that ensure that the program is correct. **Lock-err** is a buggy version of the program. **M-WO** is a Micro-Wave Oven controller. We checked that the oven will stop once it is hot, and that it cannot continue heating forever. **File** is a file management program. **W.G.C.** checks to solve the Wolf, Goat and Cabbage problem. **btrfsfile.c** models the source file *file.c* from the linux btrfs file system. We found a lock error in this file. **Bluetooth** is a simplified model of a Bluetooth driver [20]. We also found an error in this system. **w83627ehf**, **w83697ehf** and **advantech** are watchdog linux drivers. **at91rm9200** and **at32ap700x** are Real Time Clock drivers for linux. **pcf857x** corresponds also to a driver. **IEEE1394** is the IEEE 1394 driver in Linux. As described in Figure 3, we found errors in some of these drivers. We needed regular valuations to express the properties of the IEEE 1394 driver. For example, we needed to check that whenever a function *call_hpsb_send_phy_config* is invoked, there is a path where *call_hpsb_send_packet* is called before *call_hpsb_send_phy_config* returns. We need propositions about the stack to express this property. "Standard" CTL is not sufficient. **RSM** are examples written by us to check the efficiency of the regular valuations part of our tool.

7 Related Work

Alternating Büchi Pushdown Systems can be seen as non-deterministic Büchi Pushdown Systems over trees. Emptiness of non-deterministic Büchi Pushdown Systems over trees is solved in triple exponential time by Harel and Raz [15]. Our algorithm is less complex. [2] considers the emptiness problem in Alternating Parity Pushdown Automata. The emptiness problem of nondeterministic parity pushdown tree automata is investigated in [16,3,4]. ABPDSs can be seen as a subclass of these Automata. For ABPDSs, our algorithm is more general than the ones in these works since it allows to characterize and compute the set of configurations from which the ABPDS has an accepting run, whereas the other algorithms allow only to check emptiness

Model-checking pushdown systems against branching time temporal logics has already been intensively investigated in the literature. Several algorithms have been proposed. Walukiewicz [25] showed that CTL model checking is EXPTIME-complete for PDSs. The complexity of our algorithm matches this bound. CTL corresponds to a fragment of the alternation-free μ-calculus and of CTL*. Model checking full μ-calculus for PDSs has been considered in [5,6,24,18]. These algorithms allow only to determine whether a given configuration satisfies the property. They cannot compute the set of all the configurations where the formula holds. As far as CTL is concerned, our algorithm is more general since it allows to compute a finite automaton that characterizes the set of all such configurations. Moreover, the complexity of our algorithm is comparable to the ones of [5,6,24,18] when applied to CTL, it is even better in some cases.

[19,17] considers the global model-checking μ-calculus problem for PDSs, i.e., they compute the set of configurations that satisfy the formula. They reduce this problem to the membership problem in two-way alternating parity tree automata. [17] considers also μ-calculus model-checking with regular valuations. These algorithms are more complex, technically more complicated and less intuitive than our procedure. Indeed,

the complexity of [19,17] is $(|\varphi| \cdot |P| \cdot |\Delta| \cdot |\Gamma|)^{O(|P| \cdot |\Delta| \cdot |\varphi|)^2}$, whereas our complexity is $O(|P|^2 \cdot |\varphi|^3 \cdot (|P| \cdot |\Gamma| + |\Delta|) \cdot |\Gamma| \cdot 2^{5|P||\varphi|})$.

In [1], Bouajjani et al. consider alternating pushdown systems (without the Büchi accepting condition). They provide an algorithm to compute a finite automaton representing the *Pre** of a regular set of configurations for these systems. We use this procedure in $loop_2$ of **Algorithm 1**. [23] showed how to efficiently implement this procedure. We used the ideas in [23] while implementing **Algorithm 1**. In their paper, Bouajjani et al. applied their *Pre** algorithm to compute the set of PDS configurations that satisfy a given alternation-free μ-calculus formula. Their procedure is more complex than ours. It is exponential in $|P| \cdot |\varphi|^2$ whereas our algorithm is exponential only in $|P| \cdot |\varphi|$, where $|P|$ is the number of states of the PDS and $|\varphi|$ is the size of the formula.

It is well known that the model-checking problem for μ-calculus is polynomially reducible to the problem of solving parity games. Parity games for pushdown systems are considered in [8,22] and are solved in time exponential in $(|P||\varphi|)^2$. As far as CTL model-checking is concerned, our method is simpler, less complex, and more intuitive than these algorithms.

Model checking CTL* for PDS is 2EXPTIME-complete (in the size of the formula) [2]. Algorithms for model-checking CTL* specifications for PDSs have been proposed in [14,12,11,2]. [14] considers also CTL* model checking with regular valuations. When applied to CTL formulas, these algorithms are more complex than our techniques. They are double exponential in the size of the formula and exponential in the size of the system; whereas our procedure is only exponential for both sizes (the formula and the system).

LTL model-checking with regular valuations was considered in [12,11]. Their algorithm is based on a reduction to the "standard" LTL model-checking problem for PDSs. The reduction is done by performing a kind of product of the PDS with the different regular automata representing the different constraints on the stack. Compared to these algorithms, our techniques for CTL model-checking with regular valuations are direct, in the sense that they do not necessitate to make the product of the PDS with the different automata of the regular constraints.

References

1. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 133–150. Springer, Heidelberg (1997)
2. Bozzelli, L.: Complexity results on branching-time pushdown model checking. Theor. Comput. Sci. 379(1-2), 286–297 (2007)
3. Bozzelli, L., Murano, A., Peron, A.: Pushdown module checking. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 504–518. Springer, Heidelberg (2005)
4. Bozzelli, L., Murano, A., Peron, A.: Pushdown module checking. Formal Methods in System Design 36(1), 65–95 (2010)
5. Burkart, O., Steffen, B.: Composition, decomposition and model checking of pushdown processes. Nord. J. Comput. 2(2), 89–125 (1995)
6. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 419–429. Springer, Heidelberg (1997)

7. Cachat, T.: Symbolic strategy synthesis for games on pushdown graphs. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 704–715. Springer, Heidelberg (2002)
8. Cachat, T.: Uniform solution of parity games on prefix-recognizable graphs. Electr. Notes Theor. Comput. Sci. 68(6) (2002)
9. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithm for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
10. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 14–30. Springer, Heidelberg (1999)
11. Esparza, J., Kučera, A., Schwoon, S.: Model-checking LTL with regular valuations for pushdown systems. In: Kobayashi, N., Babu, C. S. (eds.) TACS 2001. LNCS, vol. 2215, pp. 316–339. Springer, Heidelberg (2001)
12. Esparza, J., Kucera, A., Schwoon, S.: Model checking ltl with regular valuations for pushdown systems. Inf. Comput. 186(2), 355–376 (2003)
13. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 324. Springer, Heidelberg (2001)
14. Finkel, A., Willems, B., Wolper, P.: A Direct Symbolic Approach to Model Checking Pushdown Systems. In: Infinity 1997. ENTCS, vol. 9. Elsevier Sci. Pub., Amsterdam (1997)
15. Harel, D., Raz, D.: Deciding emptiness for stack automata on infinite trees. Inf. Comput. 113(2), 278–299 (1994)
16. Kupferman, O., Piterman, N., Vardi, M.Y.: Pushdown specifications. In: Baaz, M., Voronkov, A. (eds.) LPAR 2002. LNCS (LNAI), vol. 2514, pp. 262–277. Springer, Heidelberg (2002)
17. Kupferman, O., Piterman, N., Vardi, M.Y.: An automata-theoretic approach to infinite-state systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 202–259. Springer, Heidelberg (2010)
18. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to reasoning about infinite-state systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 36–52. Springer, Heidelberg (2000)
19. Piterman, N., Vardi, M.Y.: Global model-checking of infinite-state systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 387–400. Springer, Heidelberg (2004)
20. Qadeer, S., Wu, D.: Kiss: Keep it simple and sequential. In: PLDI 2004: Programming Language Design and Implementation, pp. 14–24 (2004)
21. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
22. Serre, O.: Note on winning positions on pushdown games with [omega]-regular conditions. Inf. Process. Lett. 85(6), 285–291 (2003)
23. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: Efficient algorithms for alternating pushdown systems with an application to the computation of certificate chains. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 141–153. Springer, Heidelberg (2006)
24. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 62–74. Springer, Heidelberg (1996)
25. Walukiewicz, I.: Model checking CTL properties of pushdown systems. In: Kapoor, S., Prasad, S. (eds.) FST TCS 2000. LNCS, vol. 1974, pp. 127–138. Springer, Heidelberg (2000)

Reasoning about Threads with Bounded Lock Chains

Vineet Kahlon

NEC Laboratories America, USA

Abstract. The problem of model checking threads interacting purely via the standard synchronization primitives is key for many concurrent program analyses, particularly dataflow analysis. Unfortunately, it is undecidable even for the most commonly used synchronization primitive, i.e., mutex locks. Lock usage in concurrent programs can be characterized in terms of lock chains, where a sequence of mutex locks is said to be chained if the scopes of adjacent (non-nested) mutexes overlap. Although the model checking problem for fragments of Linear Temporal Logic (LTL) is known to be decidable for threads interacting via nested locks, i.e., chains of length one, these techniques don't extend to programs with non-nested locks used in crucial applications like databases. We exploit the fact that lock usage patterns in real life programs do not produce unbounded lock chains. For such a framework, we show, by using the new concept of Lock Causality Automata (LCA), that pre^*-closures of regular sets of states can be computed efficiently. Leveraging this new technique then allows us to formulate decision procedures for model checking threads communicating via bounded lock chains for fragments of LTL. Our new results narrow the decidability gap for LTL model checking of threads communicating via locks by providing a more refined characterization for it in terms of boundedness of lock chains rather than the current state-of-the-art, i.e., nestedness of locks (chains of length one).

1 Introduction

With the increasing prevalence of multi-core processors and concurrent multi-threaded software, it is highly critical that dataflow analysis for concurrent programs, similar to the ones for the sequential domain, be developed. For sequential programs, Pushdown Systems (PDSs) have emerged as a powerful, unifying framework for efficiently encoding many inter-procedural dataflow analyses [15, 5]. Given a sequential program, abstract interpretation is first used to get a finite representation of the control part of the program while recursion is modeled using a stack. Pushdown systems then provide a natural framework to model such abstractly interpreted structures. Analogous to the sequential case, inter-procedural dataflow analysis for concurrent multi-threaded programs can be formulated as a model checking problem for interacting PDSs. While for a single PDS the model checking problem is efficiently decidable for very expressive logics, it was shown in [18] that even simple properties like reachability become undecidable for systems with only two threads but where the threads synchronize using CCS-style pairwise rendezvous.

However, it has recently been demonstrated that, in practice, concurrent programs have a lot of inherent structure that if exploited leads to decidability of many important problems of practical interest. These results show that there are important fragments of temporal logics and useful models of interacting PDSs for which efficient decidability

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 450–465, 2011.

results can be obtained. Since formulating efficient procedures for model checking interacting PDSs lies at the core of scalable data flow analysis for concurrent programs, it is important that such fragments be identified for the standard synchronization primitives. Furthermore, of fundamental importance also is the need to delineate precisely the decidability boundary of the model checking problem for PDSs interacting via the standard synchronization primitives.

Nested locks are a prime example of how programming patterns can be exploited to yield decidability of the model checking problem for several important temporal logic fragments for interacting pushdown systems [13, 11]. However, even though the use of nested locks remains the most popular lock usage paradigm there are niche applications, like databases, where lock chaining is required. Chaining occurs when the scopes of two mutexes overlap. When one mutex is acquired the code enters a region where another mutex is required. After successfully locking that second mutex, the first one is no longer needed and is released. Lock chaining is an essential tool that is used for enforcing serialization, particularly in database applications. For instance, the two-phase commit protocol [14] which lies at the heart of serialization in databases uses lock chains of length 2. Other classic examples where non-nested locks occur frequently are programs that use both mutexes and (locks associated with) Wait/Notify primitives (condition variables). It is worth pointing out that the lock usage pattern of bounded lock chains covers almost all cases of practical interest encountered in real-life programs.

We consider the model checking problem for pushdown systems synchronizing via bounded lock chains for LTL properties. Decidability of a sub-logic of LTL hinges on whether it is expressive enough to encode, as a model checking problem, the disjointness of the context-free languages accepted by the PDSs in the given multi-PDS system - an undecidable problem. This, in turn, depends on the temporal operators allowed by the sub-logic thereby providing a natural way to characterize LTL-fragments for which the model checking problem is decidable. We use $L(Op_1, ..., Op_k)$, where $Op_i \in \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \overset{\infty}{\mathsf{F}}\}$, to denote the fragment comprised of formulae of the form $\mathsf{E}f$, where f is an LTL formula in positive normal form (PNF), viz., only atomic propositions are negated, built using the operators $Op_1, ..., Op_k$ and the boolean connectives $\vee$ and $\wedge$. Here X "next-time", F "sometimes", U, "until", G "always", and $\overset{\infty}{\mathsf{F}}$ "infinitely-often" denote the standard temporal operators and E is the "existential path quantifier". Obviously, $L(\mathsf{X}, \mathsf{U}, \mathsf{G})$ is the full-blown LTL.

It has recently been shown that pairwise reachability is decidable for threads interacting via bounded lock chains [10]. In this paper, we extend the envelope of decidability for concurrent programs with bounded lock chains to richer logics. Specifically, we show that the model checking problem for threads interacting via bounded lock chains is decidable not just for reachability but also the fragment of LTL allowing the temporal operators X, F, $\overset{\infty}{\mathsf{F}}$ and the boolean connectives $\wedge$ and $\vee$, denoted by $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$. It is important to note that while pairwise reachability is sufficient for reasoning about simple properties like data race freedom, for more complex properties one needs to reason about richer formulae. For instance, detecting atomicity violations requires reasoning about the fragment of LTL allowing the operators F, $\wedge$ and $\vee$ (see [14]).

Moreover, we also delineate precisely the decidability/undecidability boundary for the problem of model checking dual-PDS systems synchronizing via bounded lock chains. Specifically, we show the following.

1. The model checking problem is undecidable for $L(\mathsf{U})$ and $L(\mathsf{G})$. This implies that in order to get decidability for dual-PDS systems interacting via bounded lock chains, we have to restrict ourselves to the sub-logic $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$. Since systems comprised of PDSs interacting via bounded lock chains are more expressive than those interacting via nested locks (chains of length one) these results follow immediately from the undecidability results for PDSs interacting via nested locks [11].

2. For the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL we show that the model checking problem is decidable.

This settles the model checking problem for threads interacting via bounded lock chains for LTL. The prior state-of-the-art characterization of decidability vs. undecidability for threads interacting via locks was in terms of nestedness vs. non-nestedness of locks. We show that decidability can be re-characterized in terms of boundedness vs. unboundedness of lock chains. Since nested locks form chains of length one, our results are strictly more powerful than the existing ones. Thus, our new results narrow the decidability gap by providing a more refined characterization for the decidability of LTL for threads interacting via locks.

A key contribution of the paper is the new notion of a *Lock Causality Automaton (LCA)* that is used to represent sets of states of the given concurrent program so as to allow efficient temporal reasoning about programs with bounded lock chains. To understand the motivation behind an LCA, we recall that when model checking a single PDS, we exploit the fact that the set of configurations satisfying any given LTL formula is regular and can therefore be captured via a finite automaton or, in the terminology of [5], a multi-automaton. For a concurrent program with two PDSs T_1 and T_2, however, we need to reason about pairs of regular sets of configuration - one for each thread. An LCA is a pair of automata (M_1, M_2), where M_i accepts a regular set of configurations of T_i. The usefulness of an LCA stems from the fact that not only does it allow us to reason about $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ properties for concurrent programs with bounded lock chains, but that it allows us to do so in a compositional manner. Compositional reasoning allows us to reduce reasoning about the concurrent program at hand to each of its individual threads. This is crucial in ameliorating the state explosion problem. The main challenge in reducing model checking of a concurrent program to its individual threads lies in tracking relevant information about threads locally that enables us to reason globally about the concurrent program. For an LCA this is accomplished by tracking regular lock access patterns in individual threads.

To sum up, the key contributions of the paper are

1. The new notion of an LCA that allows us to reason about concurrent programs with bounded lock chains in a compositional manner.

2. A model checking procedure for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL that allows us to narrow the decidability gap for model checking LTL properties for threads communicating via locks.

3. Delineation of the decidability boundary for the LTL model checking problem for threads synchronizing via bounded lock chains.

2 System Model

We consider concurrent programs comprised of threads modeled as Pushdown Systems (PDSs) [5] that interact with each other using synchronization primitives. PDSs are a natural model for abstractly interpreted programs used in key applications like dataflow analysis [15]. A PDS has a finite control part corresponding to the valuation of the variables of a thread and a stack which provides a means to model recursion.

Formally, a PDS is a five-tuple $P = (Q, Act, \Gamma, c_0, \Delta)$, where Q is a finite set of *control locations*, *Act* is a finite set of *actions*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (Q \times \Gamma) \times Act \times (Q \times \Gamma^*)$ is a finite set of *transitions*. If $((p, \gamma), a, (p', w)) \in \Delta$ then we write $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$. A *configuration* of P is a pair $\langle p, w \rangle$, where $p \in Q$ denotes the control location and $w \in \Gamma^*$ the *stack content*. We call c_0 the *initial configuration* of P. The set of all configurations of P is denoted by $\mathcal{C}$. For each action a, we define a relation $\xrightarrow{a} \subseteq \mathcal{C} \times \mathcal{C}$ as follows: if $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle$, then $\langle q, \gamma v \rangle \xrightarrow{a} \langle q', wv \rangle$ for every $v \in \Gamma^*$ – in which case we say that $\langle q', wv \rangle$ results from $\langle q', \gamma v \rangle$ by firing the transition $\langle q, \gamma \rangle \overset{a}{\hookrightarrow} \langle q', w \rangle$ of P.

We model a concurrent program with n threads and m locks[1] $\mathsf{l}_1, ..., \mathsf{l}_m$ as a tuple of the form $\mathcal{CP} = (T_1, ..., T_n, L_1, ..., L_m)$, where T_1,...,T_n are pushdown systems (representing threads) with the same set *Act* of non-*acquire* and non-*release* actions, and for each i, $L_i \subseteq \{\perp, 1, ..., n\}$ is the possible set of values that lock l_i can be assigned. A global configuration of $\mathcal{CP}$ is a tuple $c = (t_1, ..., t_n, l_1, ..., l_m)$ where $t_1, ..., t_n$ are, respectively, the configurations of threads $T_1, ..., T_n$ and $l_1, ..., l_m$ the values of the locks. If no thread holds the lock l_i in configuration c, then $l_i = \perp$, else l_i is the index of the thread currently holding l_i. The initial global configuration of $\mathcal{CP}$ is $(c_1, ..., c_n, \perp, ..., \perp)$, where c_i is the initial configuration of thread T_i. Thus all locks are *free* to start with. We extend the relation $\xrightarrow{a}$ to pairs of global configurations of $\mathcal{CP}$ in the standard way by encoding the interleaved parallel composition of $T_1, ..., T_n$ (see the full paper [1] for the precise definition).

Correctness Properties. We consider correctness properties expressed as double-indexed Linear Temporal Logic (LTL) formulae. Here atomic propositions are interpreted over pairs of control states of different PDSs in the given multi-PDS system.

Conventionally, $\mathcal{CP} \models f$ for a given LTL formula f if and only if f is satisfied along all paths starting at the initial state of $\mathcal{CP}$. Using path quantifiers, we may write this as $\mathcal{CP} \models \mathsf{A}f$. Equivalently, we can model check for the dual property $\neg\mathsf{A}f = \mathsf{E}\neg f = \mathsf{E}g$. Furthermore, we can assume that g is in *positive normal form (PNF)*, viz., the negations are pushed inwards as far as possible using DeMorgan's Laws: $(\neg(p \vee q)) = \neg p \wedge \neg q$, $\neg(p \vee q) = \neg p \wedge \neg q$, $\neg Fp \equiv Gq$, $\neg(pUq) \equiv G\neg q \vee \neg qU(\neg p \wedge \neg q)$.

For Dual-PDS systems, it turns out that the model checking problem is not decidable for the full-blown double-indexed LTL but only for certain fragments. Decidability hinges on the set of temporal operators that are allowed in the given property which, in turn, provides a natural way to characterize such fragments. We use $L(Op_1, ..., Op_k)$, where $Op_i \in \{\mathsf{X}, \mathsf{F}, \mathsf{U}, \mathsf{G}, \overset{\infty}{\mathsf{F}}\}$, to denote the fragment of double-indexed LTL comprised of formulae in positive normal form (where only atomic propositions are negated) built using the operators $Op_1, ..., Op_k$ and the boolean connectives $\vee$ and $\wedge$. Here X "next-time", F "sometimes", U, "until", G "always", and $\overset{\infty}{\mathsf{F}}$ "infinitely-often" denote the standard temporal operators (see [8]). Obviously, $L(\mathsf{X}, \mathsf{U}, \mathsf{G})$ is the full-blown double-indexed LTL.

[1] We do not allow recursive/re-entrant locks.

Outline of Paper. In this paper, we show decidability of the model checking problem for the fragment $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ of LTL for concurrent programs with bounded lock chains. Given an $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formula f, we build automata accepting global states of the given concurrent program satisfying f. Towards that end, we first show how to construct automata for the basic temporal operators F, $\overset{\infty}{\mathsf{F}}$ and X, and the boolean connectives $\wedge$ and $\vee$. Then to compute an automaton for the given property f, we start by building for each atomic proposition *prop* of f, an automaton accepting the set of states of the given concurrent program satisfying *prop*. Leveraging the constructions for the basic temporal operators and boolean connectives we then recursively build the automaton accepting the set of states satisfying f via a bottom-up traversal of the parse tree for f. Then if the initial state of the given concurrent program is accepted by the resulting automaton, the program satisfies f. The above approach, which is standard for LTL model checking of finite state and pushdown systems, exploits the fact that for model checking it suffices to reason about regular sets of configurations of these systems. These sets can be captured using regular automata which then reduces model checking to computing regular automata for each of the temporal operators and boolean connectives. However, for concurrent programs the sets of states that we need to reason about for model checking are not regular and cannot therefore be captured via regular automata. We therefore propose the new notion of a *Lock Causality Automaton (LCA)* that is well suited for reasoning about concurrent programs with bounded lock chains. A key contribution of the paper lies is showing how to construct LCAs for the basic temporal operators and the boolean connectives.

The constructions of LCAs for the various temporal operators depend on computing an LCA accepting the pre^*-closure of the set of states accepted by a given LCA. This in turn, hinges on deciding pairwise CFL-reachability (see sec. 3) of a pair c_1 and c_2 of configurations from another pair d_1 and d_2 of configurations of T_1 and T_2, respectively. Our decision procedure for pairwise CFL-reachability relies on the notion of a *Bidirectional Lock Causality Graph* introduced in the next section. This leads naturally to the notion of an LCA defined in sec. 4. Finally the constructions of LCAs for the basic temporal operators are given in sec. 5 which leads to the model checking procedure for $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$ formulated in sec. 5.1.

3 Pairwise CFL-Reachability

A key step in the computation of pre^*-closure of LCAs is deciding *Pairwise CFL-Reachability.*

Pairwise CFL-Reachability. *Let $\mathcal{CP}$ be a concurrent program comprised of threads T_1 and T_2. Given pairs (c_1, c_2) and (d_1, d_2), with c_i and d_i being control locations of T_i, does there exist a path of $\mathcal{CP}$ leading from a global state with T_i in c_i to one with T_i in d_i in the presence of recursion and scheduling constraints imposed by locks.*

It is known that pairwise CFL-reachability is undecidable for two threads interacting purely via locks but decidable if the locks are nested [12] and, more generally, for programs with bounded length lock chains [10], where a lock chain is defined as below.

Lock Chains. *Given a computation x of a concurrent program, a lock chain of thread T is a sequence of lock acquisition statements $acq_1, ..., acq_n$ fired by T along x in the*

Algorithm 1. Bi-Directional Lock Causality Graph

```
Input: Local paths $x^1$ and $x^2$ of $T_1$ and $T_2$ leading from $c_1$ and $c_2$ to $d_1$ and $d_2$, respectively
for each lock $l$ held at location $d_i$ do
    If $c$ and $c'$ are the last statements to acquire and release $l$ occurring along $x^i$ and $x^{i'}$, respectively, Add edge $c' \rightsquigarrow c$ to $G_{(x^1,x^2)}$.
end for
for each lock $l$ held at location $c_i$ do
    If $c$ and $c'$ are the first statements to release and acquire $l$ occurring along $x^i$ and $x^{i'}$, respectively, add edge $c \rightsquigarrow c'$ to $G_{(x^1,x^2)}$.
end for
repeat
    for each lock $l$ and each edge $d_{i'} \rightsquigarrow d_i$ of $G_{(x^1,x^2)}$ do
        Let $a_{i'}$ be the last statement to acquire $l$ before $d_{i'}$ along $x^{i'}$ and $r_{i'}$ the matching release for $a_{i'}$ and let $r_i$ be the first statement to release $l$ after $d_i$ along $x^i$ and $a_i$ the matching acquire for $r_i$
        if $l$ is held at either $d_i$ or $d_{i'}$ then
            if there does not exist an edge $b_{i'} \rightsquigarrow b_i$ such that $r_{i'}$ lies before $b_{i'}$ along $x^{i'}$ and $a_i$ lies after $b_i$ along $x^i$ then
                add edge $r_{i'} \rightsquigarrow a_i$ to $G_{(x^1,x^2)}$
            end if
        end if
    end for
until no new statements can be added to $G_{(x^1,x^2)}$
for $i \in [1..2]$ do
    Add edges among locations of $x^i$ in $G_{(x^1,x^2)}$ to preserve their relative ordering along $x^i$
end for
```

order listed such that for each i, the matching release of acq_i is fired after acq_{i+1} and before acq_{i+2} along x.

However, the decision procedures for programs with bounded lock chains [10] only apply to the case wherein c_1 and c_2 are *lock-free*, i.e., no lock is held by T_i at c_i. In order to decide the pairwise CFL-reachability problem for the general case, we propose the notion of a *Bi-directional Lock Causality Graph* which is a generalization of the (unidirectional) lock causality graph presented in [10].

Bidirectional Lock Causality Graph (BLCG). Consider the example concurrent program comprised of threads T_1 and T_2 shown in fig. 1. Suppose that we are interested in deciding whether *a7* and *b7* are pairwise reachable starting from the locations *a1* and *b1* of T_1 and T_2, respectively. Note that the set of locks held at *a1* and *b1* are $\{l_1\}$ and $\{l_3, l_5\}$, respectively. For *a7* and *b7* to be pairwise reachable there must exist local paths x^1 and x^2 of T_1 and T_2 leading to *a7* and *b7*, respectively, along which locks can be acquired and released in a consistent fashion. We start by constructing a *bi-directional lock causality graph* $G_{(x^1,x^2)}$ that captures the constraints imposed by locks on the order in which statements along x^1 and x^2 need to be executed in order for T_1 and T_2 to simultaneously reach *a7* and *b7*. The nodes of this graph are (the relevant) locking/unlocking statements fired along x^1 and x^2. For statements c_1 and c_2 of $G_{(x^1,x^2)}$,

```
void T1(){
a1:  lock(l4);
a2:  lock(l5);
a3:  unlock(l4);
a4:  unlock(l5);
a5:  unlock(l1);
a6:  lock(l3);
a7:  Race0;
}
```

```
void T2(void){
b1:  lock(l4);
b2:  unlock(l5);
b3:  unlock(l3);
b4:  lock(l1);
b5:  lock(l2);
b6:  unlock(l4);
b7:  Race1;
}
```

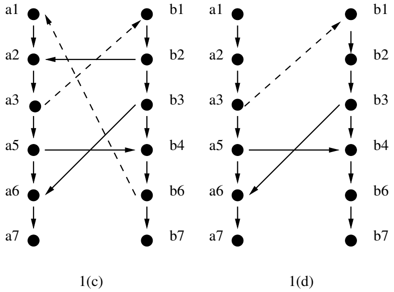


Fig. 1. An Example Program and its Bi-directional Lock Causality Graph

there exists an edge from c_1 to c_2, denoted by $c_1 \rightsquigarrow c_2$, if c_1 must be executed before c_2 in order for T_1 and T_2 to simultaneously reach *a7* and *b7*.

$G_{(x^1,x^2)}$ has two types of edges (i) *Seed* edges and (ii) *Induced* edges.

Seed Edges: Seed edges, which are shown as bold edges in fig. 1(c), can be further classified as (a) *Backward* and (b) *Forward* seed edges.

(a) **Forward Seed Edges:** Consider lock l_1 held at *b7*. Note that once T_2 acquires l_1 at location *b4*, it is not released along the path from *b4* to *b7*. Since we are interested in the pairwise CFL-reachability of $a7$ and $b7$, T_2 cannot progress beyond location *b7* and therefore cannot release l_1. Thus we have that once T_2 acquires l_1 at *b4*, T_1 cannot acquire it thereafter. If T_1 and T_2 are to simultaneously reach *a7* and *b7*, the last transition of T_1 that releases l_1 before reaching *a7*, i.e., *a5*, must be executed before *b4*. Thus $a5 \rightsquigarrow b4$.

(b) **Backward Seed Edges:** Consider lock l_5 held at *b1*. In order for T_1 to acquire l_5 at *a2*, l_5 must first be released by T_2. Thus the first statement of T_1 acquiring l_5 starting at *a1*, i.e., *a2*, must be executed after *b2*. Thus $b2 \rightsquigarrow a2$.

The interaction of locks and seed causality edges can be used to deduce further causality constraints that are captured as *induced* edges (shown as dashed edges in the BLCG in fig. 1(c)). These induced edges are key in guaranteeing both soundness and completeness of our procedure.

Induced Edges: Consider the constraint *b2* $\rightsquigarrow$ *a2*. At location *b2*, lock l_4 is held which was acquired at *b1*. Also, once l_4 is acquired at *b1* it is not released till after T_2 exits *b6*. Thus since l_4 has been acquired by T_2 before reaching *b2* it must be released before *a1* (and hence *a2*) can be executed. Thus, *b6* $\rightsquigarrow$ *a1*.

Computing the Bidirectional Lock Causality Graph. Given finite local paths x^1 and x^2 of threads T_1 and T_2 starting at control locations c_1 and c_2 and leading to control locations d_1 and d_2, respectively, the procedure (see alg. 1) to compute $G_{(x^1,x^2)}$ adds the causality constraints one-by-one (forward seed edges via steps 2-6, backward seed edges via steps 7-11 and induced edges via steps 12-24) till we reach a fixpoint. Throughout the description of alg. 1, for $i \in [1..2]$, we use i' to denote an integer in $[1..2]$ other than i. Note that condition 18 in alg. 1 ensures that we do not add edges representing causality constraints that can be deduced from existing edges.

Necessary and Sufficient Condition for CFL-Reachability. Let x^1 and x^2 be local computations of T_1 and T_2 leading to c_1 and c_2. Since each causality constraint in

$G_{(x^1,x^2)}$ is a *happens-before* constraint, we see that in order for c_1 and c_2 to be pairwise reachable $G_{(x^1,x^2)}$ has to be acyclic. In fact, it turns out that acyclicity is also a sufficient condition (see [1] for the proof).

Theorem 1. (Acyclicity). *Locations d_1 and d_2 are pairwise reachable from locations c_1 and c_2, respectively, if there exist local paths x^1 and x^2 of T_1 and T_2, respectively, leading from c_1 and c_2 to d_1 and d_2, respectively, such that (1) $L_{T_1}(c_1) \cap L_{T_2}(c_2) = \emptyset$ (disjointness of* **backward locksets***), (2) $L_{T_1}(d_1) \cap L_{T_2}(d_2) = \emptyset$ (disjointness of* **forward locksets***), and (3) $G_{(x^1,x^2)}$ is acyclic. Here $L_T(e)$ denotes the set of locks held by thread T at location e.*

Synergy Between Backward and Forward Lock Causality Edges. Note that in order to deduce that *a7* and *b7* are not pairwise reachable it is important to consider causality edges induced by both backward and forward seed edges. Ignoring either of these may cause us to incorrectly deduce that $a7$ and $b7$ are reachable. In the above example if we ignore the backward seed edges then we will construct the unidirectional lock causality graph $L_{(x^1,x^2)}$ shown in fig. 1(d) which is acyclic. Thus the lock causality graph construction of [10] is inadequate in reasoning about bi-directional pairwise reachability.

Bounding the Size of the Lock Causality Graph. Under the assumption of bounded lock chains, we show that the size of the bidirectional lock causality graph is bounded. From alg. 1 it follows that each causality edge is induced either by an existing induced causality edge or a backward or forward seed edge. Thus for each induced causality edge e, there exists a sequence $e_0, ..., e_n$ of causality edges such that e_0 is a seed edge and for each $i \geq 1$, e_i is induced by e_{i-1}. Such a sequence is referred to as a lock causality sequence. Under the assumption of bounded lock chains it was shown in [10] that the length of any lock causality sequence is bounded. Note that the number of seed edges is at most $4|L|$, where $|L|$ is the number of locks in the given concurrent program. Since the number of seed edges is bounded, and since the length of each lock causality sequence is bounded, the number of induced edges in each bi-directional lock causality graph is also bounded leading to the following result.

Theorem 2. (Bounded Lock Causality Graph). *If the length of each lock chain generated by local paths x^1 and x^2 of threads T_1 and T_2, respectively, is bounded then the size (number of vertices) of $G_{(x^1,x^2)}$, is also bounded.*

4 Lock Causality Automata

When model checking a single PDS, we exploit the fact that the set of configurations satisfying a given LTL formula is regular and can therefore be captured via a finite automaton also called a multi-automaton [5]. For a concurrent program with two PDSs, however, we need to reason about pairs of regular sets of configurations. Thus instead of performing pre^*-closures over multi-automata, we need to perform pre^*-closures over automata pairs.

Suppose that we are given a pair (R_1, R_2) of sets, where R_i is a regular set of configurations of thread T_i. The set S_i of configurations of T_i that are (locally) backward reachable from R_i forms a regular set [5]. However, given a pair of configurations (a_1, a_2), where $a_i \in S_i$, even though a_i is backward reachable from some $b_i \in R_i$ in T_i, there is no guarantee that a_1 and a_2 are pairwise backward reachable from b_1 and

b_2 in the concurrent program $\mathcal{CP}$. That happens only if there exist local paths x^1 and x^2 of threads T_1 and T_2, respectively, from a_i to b_i such that $G_{(x^1,x^2)}$ is acyclic. Thus in computing the pre^*-closure S_i of R_i in thread T_i, we need to track relevant lock access patterns that allow us to deduce acyclicity of the lock causality graph $G_{(x^1,x^2)}$.

In order to capture the set of global states of $\mathcal{CP}$ that are backward reachable from (R_1, R_2), we introduce the notion of a *Lock Causality Automaton (LCA)*. An LCA is a pair of automata $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_i$ accepts the regular set of configurations of T_i that are backward reachable from R_i. For $\mathcal{L}$ to accept precisely the set of global states (a_1, a_2) that are pairwise backward reachable from $(b_1, b_2) \in (R_1, R_2)$, we encode the existence of a pair of local paths x^i from a_i to b_i generating an acyclic lock causality graph in the acceptance condition of $\mathcal{L}$. For concurrent programs with nested locks, this was accomplished by tracking forward and backward acquisition histories and incorporating a consistency check for these acquisition histories (a necessary and sufficient condition for pairwise reachability) in the acceptance condition of $\mathcal{L}$ [12]. A key feature of acquisition histories that we exploited was that they are defined locally for each thread and could therefore be tracked during the (local) computation of the pre^*-closure of R_i. In contrast, the lock causality graph depends on lock access patterns of both threads. Thus we need to locally track relevant information about lock accesses in a manner that allows us to re-construct the (global) lock causality graph. Towards that end, the following result is key. Let L be the set of locks in the given concurrent program and let $\Sigma_L = \cup_{l \in L}\{a_l, r_l\}$, where a_l and r_l denote labels of transitions acquiring and releasing lock l, respectively, in the given program.

Theorem 3. (Regular Decomposition) *Let G be a directed bipartite graph over nodes labeled with lock acquire/release labels from the (finite) set Σ_L. Then there exist regular automata G_{11}, ..., G_{1n}, G_{21}, ..., G_{2n} over Σ_L such that the set $\{(x^1, x^2) | x^1 \in \Sigma_L^*, x^2 \in \Sigma_L^*, G_{(x^1,x^2)} = G\}$ can be represented as $\bigcup_i L(G_{i1}) \times L(G_{i2})$, where $L(G_{ij})$ is the language accepted by G_{ij}.*

To prove this result, we introduce the notion of a lock schedule. The motivation behind the definition of a lock schedule is that not all locking events, i.e., lock/unlock statements, along a local computation x of a thread T need occur in a lock causality graph involving x. A lock schedule u is intended to capture only those locking events $u : u_0, ..., u_m$ that occur in a lock causality graph. The remaining locking events, i.e., those occurring between u_i and u_{i+1} along x are specified in terms of its complement set F_i, i.e., symbols from Σ_L that are forbidden to occur between u_i and u_{i+1}. We require that if u_i is the symbol a_l, representing the acquisition of lock l and if its matching release r_l is executed along x then that matching release also occurs along the sequence u, i.e., $u_j = r_l$ for some $j > i$. Also, since l cannot be acquired twice, in order to preserve locking semantics the letters a_l and r_l cannot occur between u_i and u_j along x. This is captured by including a_l and r_l in each of the forbidden sets $F_i, ..., F_{j-1}$.

Definition (Lock Schedule). *A lock schedule is a sequence u_0, ..., $u_m \in \Sigma_L^*$ having for each i, a set $F_i \subseteq \Sigma_L$ associated with u_i such that if $u_i = a_l$ and u_j its matching release, then for each k such that $i \le k < j$ we have $r_l, a_l \in F_k$. We denote such a lock schedule by $u_0 F_0 u_1 ... u_m F_m$.*

We say that a sequence $x \in \Sigma_L^*$ satisfies a given lock schedule $sch = u_0 F_0 u_1 ... u_m F_m$, denoted by $sch \models x$, if $x \in u_0(\Sigma_L \setminus F_0)^* u_1 ... u_m (\Sigma_L \setminus F_m)^*$. The following is an easy consequence of the above definition.

Lemma 4. *The set of sequences in Σ_L^* satisfying a given lock schedule is regular.*

The proof of thm. 3 then follows easily from the following (see [1] for all the proofs).

Theorem 5. *Given a lock causality graph G, we can construct a finite set SCH_G of pairs of lock schedules such that the set of pairs of sequences in Σ_L^* generating G is precisely the set of pairs of sequences in Σ_L^* satisfying at least one schedule pair in SCH_G, i.e., $\{(x^1, x^2)|x^1, x^2 \in \Sigma_L^*, G_{(x^1,x^2)} = G\} = \{(y^1, y^2)|\ y^1, y^2 \in \Sigma_L^*, \textit{for some}$ $(\mathsf{sch}_1, \mathsf{sch}_2) \in \mathsf{SCH}_G, \mathsf{sch}_1 \models y^1$ and $\mathsf{sch}_2 \models y^2\}$.*

Lock Causality Automata. We now formally define the notion of a Lock Causality Automaton. Since for programs with bounded lock chains the number of lock causality graphs is bounded (thm. 2), so is the number of acyclic lock causality graphs. With each acyclic lock causality graph G we can, using thm. 5, associate a finite set $\mathsf{ACYC_G}$ of automata pairs that accept all pairs of sequences in $\Sigma_L^* \times \Sigma_L^*$ generating G. By taking the union over all acyclic lock causality graphs G, we construct the set of all automata pairs that accept all pairs of sequences in $\Sigma_L^* \times \Sigma_L^*$ generating acyclic lock causality graphs. We denote all such pairs by ACYC. Let $(\mathsf{G}_{11}, \mathsf{G}_{21}), ..., (\mathsf{G}_{1n}, \mathsf{G}_{2n})$ be an enumeration of all automata pairs of ACYC.

We recall that a key motivation in defining LCAs is to capture the pre^*-closure, i.e., the set of pairs of configurations that are pairwise backward reachable from a pair of configurations in (R_1, R_2), where R_i is a regular set of configurations of T_i. We therefore define an LCA to be a pair of the form $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_i$ is a multi-automaton accepting the set of configurations of T_i that are backward reachable from configurations in R_i. Note that if (a_1, a_2) is pairwise backward reachable from $(b_1, b_2) \in (R_1, R_2)$ then a_i is accepted by $\mathcal{L}_i$. However, due to constraints imposed by locks not all pairs of the form (c_1, c_2), where c_i is accepted by $\mathcal{L}_i$, are pairwise backward reachable from (b_1, b_2). In order for $\mathcal{L}$ to accept precisely the set of global configurations (a_1, a_2) that are pairwise backward reachable from (b_1, b_2), we encode the existence of local paths x^i from a_i to b_i generating an acyclic lock causality graph in the acceptance condition of $\mathcal{L}$. Towards that end, when performing the backward pre^*-closure in computing $\mathcal{L}_i$ we track not simply the set of configurations c of T_i that are backward reachable from R_i but also the lock schedules encountered in reaching c.

In deciding whether configurations c_1 and c_2 are pairwise backward reachable from b_1 and b_2, where $(b_1, b_2) \in (R_1, R_2)$, we only need to check whether for each $i \in [1..2]$, there exist lock schedules sch_i from c_i to b_i such that $G_{(\mathsf{sch}_1,\mathsf{sch}_2)}$ is acyclic, i.e., for some j, $(\mathsf{sch}_1, \mathsf{sch}_2) \in L(\mathsf{G}_{1j}) \times L(\mathsf{G}_{2j})$. Since, in performing backward pre^*-closure for each thread T_i, we track local computation paths and hence lock schedules in the reverse manner, we have to consider the reverse of the regular languages accepted by G_{ij}. Motivated by this, for each i, j, we let G_{ij}^r be a regular automaton accepting the language resulting by reversing each word in the language accepted by G_{ij}. Then c_1 and c_2 are pairwise backward reachable from b_1 and b_2 if there exists for each i, a (reverse) lock schedule rsch_i along a path y^i from b_i to c_i, such that for some j, rsch_1 is accepted by G_{1j}^r and rsch_2 is accepted by G_{2j}^r. Thus when computing the backward pre^*-closure in thread T_i, instead of tracking the sequence z^i of lock/unlock statements encountered thus far, it suffices to track for each j, the set of possible current local states of the regular automaton G_{ij}^r reached by traversing z^i starting at its initial state. Indeed, for each i, j, let $\mathsf{G}_{ij}^r = (Q_{ij}, \delta_{ij}, \mathsf{in}_{ij}, F_{ij})$, where Q_{ij} is the set of states of G_{ij}^r, δ_{ij} its transition relation, in_{ij} its initial state and F_{ij} its set of final states. Let

$S_{ij}(\mathsf{rsch}_i) = \delta_{ij}(\mathsf{in}_{ij}, \mathsf{rsch}_i)$. Then the above condition can be re-written as follows: c_1 and c_2 are pairwise backward reachable from b_1 and b_2 if there exists for each i, a lock schedule rsch_i along a path y^i from b_i to c_i, such that for some j, $S_{1j}(\mathsf{rsch}_1) \cap F_{1j} \neq \emptyset$ and $S_{2j}(\mathsf{rsch}_2) \cap F_{2j} \neq \emptyset$.

Thus in performing pre^*-closure in thread T_i, we augment the local configurations of T_i to track for each i, j, the current set of states of G_{ij} induced by the lock/unlock sequence seen so far. Hence an augmented configuration of T_i now has the form $\langle(c, FLS, BLS, GS_{i1}, ..., GS_{in}), u\ \rangle$, where FLS and BLS are the forward and backward locksets (see thm. 1) at the start and end points and GS_{ij} is the set of states of G^r_{ij} induced by the lock/unlock sequences seen so far in reaching configuration $\langle c, u\rangle$. To start with GS_{ij} is set to $\{\mathsf{in}_{ij}\}$, the initial state of G^r_{ij}.

Lock Augmented Multi-Automata. Formally, a lock augmented multi-automaton can be defined as follows: Let T_i be the pushdown system $(Q_i, Act_i, \Gamma_i, c_{i0}, \Delta_i)$. A *Lock Augmented T_i-Multi-Automaton* is a tuple $\mathcal{M}_i = (\Gamma_i, P_i, \delta_i, I_i, F_i)$, where P_i is a finite set of states, $\delta_i \subseteq P_i \times \Gamma_i \times P_i$ is a set of transitions, $I_i = \{(c, FLS, BLS, GS_{i1}, ..., GS_{in}) \mid c \in Q_i, BLS, FLS \subseteq L, GS_{ij} \subseteq Q_{ij}\} \subseteq P_i$ is a set of initial states and $F_i \subseteq P_i$ is a set of final states. $\mathcal{M}_i$ accepts an augmented configuration $\langle(c, FLS, BLS, GS_{i1}, ..., GS_{in})), u\ \rangle$ if starting at the initial state $(c, FLS, BLS, GS_{i1}, ..., GS_{in}))$ there is a path in $\mathcal{M}_i$ labeled with u and leading to a final state of $\mathcal{M}_i$. Note that the only difference between a lock augmented multi-automation and the standard multi-automaton as defined in [5] is that the control state is augmented with the lockset information BLS and FLS, and the subsets GS_{ij} used to track lock schedules.

A lock causality automaton is then defined as follows:

Definition (Lock Causality Automaton) *Given threads $T_1 = (Q_1, Act_1, \Gamma_1, \mathbf{c}_1, \Delta_1)$ and $T_2 = (Q_2, Act_2, \Gamma_2, \mathbf{c}_2, \Delta_2)$, a lock causality automaton is a pair $(\mathcal{L}_1, \mathcal{L}_2)$ where $\mathcal{L}_i$ is a lock augmented T_i-multi-automaton.*

The acyclicity check (thm. 1) for pairwise reachability is encoded in the acceptance criterion of an LCA.

Definition (LCA-Acceptance). *We say that an LCA $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$ accepts the pair $(\mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{c}_i = \langle c_i, u_i\rangle$ is a configuration of T_i, if there exist lock sets BLS_i and FLS_i, and sets $GS_{ij} \subseteq Q_{ij}$, such that*

1. for each i, the augmented configuration $\langle(c_i, FLS_i, BLS_i, GS_{i1}, ..., GS_{in}), u_i\rangle$ is accepted by $\mathcal{L}_i$,

2. $FLS_1 \cap FLS_2 = \emptyset$ and $BLS_1 \cap BLS_2 = \emptyset$, and

3. there exists k such that $GS_{1k} \cap F_{1k} \neq \emptyset$ and $GS_{2k} \cap F_{2k} \neq \emptyset$.

Intuitively, condition 1 checks for local thread reachability, condition 2 checks for disjointness of lock sets and condition 3 checks for acyclicity of the lock causality graph induced by the lock schedules leading to $\langle c_1, u_1\rangle$ and $\langle c_2, u_2\rangle$.

5 Computing LCAs for Operators

We now show how to construct LCAs for (i) *boolean Operators:* $\vee$ and $\wedge$, and (ii) *Temporal Operators:* F, $\overset{\infty}{\mathsf{F}}$ and X.

Computing LCA for F. Given an LCA $\mathcal{L} = (\mathcal{L}_1, \mathcal{L}_2)$ our goal is to compute an LCA $\mathcal{M}$, denoted by $pre^*(\mathcal{L})$, accepting the pair $(\mathbf{b}_1, \mathbf{b}_2)$ of augmented configurations that is pairwise backward reachable from some pair $(\mathbf{a}_1, \mathbf{a}_2)$ accepted by $\mathcal{L}$. In other words, $\mathcal{M}$ must accept the pre^*-closure of the set of states accepted by $\mathcal{L}$. We first show how to compute the pre^*-closure of a lock augmented T_i-multi-automaton.

Computing the pre^*-closure of a Lock Augmented Multi-Automaton. Given a lock augmented T_i-multi-automaton $\mathcal{A}$, we show how to compute another lock augmented T_i-multi-automaton $\mathcal{B}$, denoted by $pre^*(\mathcal{A})$, accepting the pre^*-closure of the set of augmented configurations of T_i accepted by $\mathcal{A}$. We recall that each augmented configuration of $\mathcal{A}$ is of the form $\langle(c, FLS, BLS, GS_{i1}, ..., GS_{in}), u\rangle$, where c is a control state of T_i, u its stack content, FLS and BLS are locksets, and GS_{ij} is the set of states of G_{ij} induced by the lock schedules seen so far in reaching configuration $\langle c, u\rangle$. We set $\mathcal{A}_0 = \mathcal{A}$ and construct a finite sequence of lock-augmented multi-automata $\mathcal{A}_0, ..., \mathcal{A}_p$ resulting in $\mathcal{B} = \mathcal{A}_p$. Towards that end, we use $\rightarrow_i$ to denote the transition relation of $\mathcal{A}_i$. For every $i \geq 0$, $\mathcal{A}_{i+1}$ is obtained from $\mathcal{A}_i$ by conserving the sets of states and transitions of $\mathcal{A}_i$ and adding new transitions as follows

1. for each *stack* transition $(c, \gamma) \hookrightarrow (c', w)$ and state q such that $(c', FLS, BLS, GS_{i1}, ..., GS_{in}) \xrightarrow{w}_i q$ we add $(c, FLS, BLS, GS_{i1}, ..., GS_{in}) \xrightarrow{\gamma}_{i+1} q$.
2. for each *lock release* operation $c \overset{r_l}{\hookrightarrow} c'$ and for every state $(c', FLS, BLS, GS_{i1}, ..., GS_{in})$ of $\mathcal{A}_i$, we add a transition $(c, FLS, BLS', GS'_{i1}, ..., GS'_{in}) \xrightarrow{\epsilon}_{i+1} (c', FLS, BLS, GS_{i1}, ..., GS_{in})$ to $\mathcal{A}_{i+1}$, where ϵ is the empty symbol; $BLS' = BLS \cup \{l_i\}$; and for each j, $GS'_{ij} = \delta_{ij}(GS_{ij}, r_l)$.
3. for every *lock acquire* operation $c \overset{a_l}{\hookrightarrow} c'$ and for every state $(c', FLS, BLS\ GS_{i1}, ..., GS_{in})$ of $\mathcal{A}_i$ we add a transition $(c, FLS', BLS'\ GS'_{i1}, ..., GS'_{in}) \xrightarrow{\epsilon}_{i+1} (c', FLS, BLS, GS_{i1}, ..., GS_{in})$ to $\mathcal{A}_{i+1}$, where ϵ is the empty symbol; $BLS' = BLS \setminus \{l\}$; $FLS' = (FLS \cup \{l\}) \setminus BLS$; and for each j, $GS'_{ij} = \delta_{ij}(GS_{ij}, a_l)$.

In the above pre^*-closure computation, the stack transitions do not affect the 'lock-augmentations' and are therefore handled in the standard way. For a lock acquire (release) transition labeled with $a_l(r_l)$ we need to track the access patterns in order to determine acyclicity of the induced LCGs. Thus in steps 2 and 3 for each GS_{ij}, we compute the set $\delta_{ij}(GS_{ij}, a_l)$ of its successor states via the symbol $r_l(a_l)$ in the regular automaton G^r_{ij} tracking reverse schedules. Moreover, the backward lockset in any configuration is simply the set of locks for which release statements have been encountered during the backward traversal but not the matching acquisitions. Thus if a release statement r_l for lock l is encountered, l is included in BLS (step 2). If later on the acquisition statement a_l is encountered then l is dropped from the BLS (step 3). Finally, the forward lockset is simply the set of locks acquired along a path that are not released. Thus a lock is included in FLS if a lock acquisition symbol is encountered during the backward traversal such that its release has not yet been encountered, i.e., $r_l \notin BLS$. Thus $FLS' = (FLS \cup \{l\}) \setminus BLS$ (step 3).

LCA for F. Given an LCA $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, we define $pre^*(\mathcal{A})$ to be the LCA $(pre^*(\mathcal{A}_1), pre^*(\mathcal{A}_2))$.

Computation of $\wedge$. Let A and B be sets of pairs of configurations accepted by LCAs $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$, respectively. We show how to construct an LCA accepting $A \cap B$ via the standard product construction.

For $1 \leq i \leq 2$, let $T_i = (Q_i, Act_i, \Gamma_i, \mathbf{c}_i, \Delta_i)$, $\mathcal{A}_i = (\Gamma_i^{\mathcal{A}}, P_i^{\mathcal{A}}, \delta_i^{\mathcal{A}}, I_i^{\mathcal{A}}, F_i^{\mathcal{A}})$ and $\mathcal{B}_i = (\Gamma_i^{\mathcal{B}}, P_i^{\mathcal{B}}, \delta_i^{\mathcal{B}}, I_i^{\mathcal{B}}, F_i^{\mathcal{B}})$. Note that for $1 \leq i \leq 2$, $\Gamma_i^{\mathcal{A}} = \Gamma_i^{\mathcal{B}} = \Gamma_i$ and $I_i^{\mathcal{A}} = I_i^{\mathcal{B}} = I_i$. Then we define the LCA $\mathcal{N} = (\mathcal{N}_1, \mathcal{N}_2)$, where $\mathcal{N}_i$ is a multi-automaton accepting $A \cap B$, as the tuple $(\Gamma_i^{\mathcal{N}}, P_i^{\mathcal{N}}, \delta_i^{\mathcal{N}}, I_i^{\mathcal{N}}, F_i^{\mathcal{N}})$, where

(i) $\Gamma_i^{\mathcal{N}} = \Gamma_i$, (ii) $P_i^{\mathcal{N}} = P_i^{\mathcal{A}} \times P_i^{\mathcal{B}}$, (iii) $I_i^{\mathcal{N}} = I_i$, (iv) $F_i^{\mathcal{N}} = F_i^{\mathcal{A}} \times F_i^{\mathcal{B}}$, and (v) $\delta_i^{\mathcal{N}} = \{(s_1, s_2) \xrightarrow{a} (t_1, t_2) | \; s_1 \xrightarrow{a} t_1 \in \delta_i^{\mathcal{A}}, s_2 \xrightarrow{a} t_2 \in \delta_i^{\mathcal{B}}\}$.

A minor technicality is that in order to satisfy the requirement in the definition of a lock-augmented multi-automaton that $I_i \subseteq P_i^{\mathcal{N}}$, we 're-name' states of the form (s, s), where $s \in I_i^{\mathcal{A}}$ as simply s. The correctness of the construction follows from the fact that it is merely the standard product construction with minor changes.

Computation of $\vee$**.** Similar to the above case (see [1]).

Dual Pumping. Let $\mathcal{CP}$ be a concurrent program comprised of the threads $T_1 = (P_1, Act, \Gamma_1, c_1, \Delta_1)$ and $T_2 = (P_2, Act, \Gamma_2, c_2, \Delta_2)$ and let f be an LTL property. Let $\mathcal{BP}$ denote the Büchi system formed by the product of $\mathcal{CP}$ and $\mathcal{B}_{\neg f}$, the Büchi automaton corresponding to $\neg f$. Then LTL model checking reduces to deciding whether there exists an accepting path of $\mathcal{BP}$.

The Dual Pumping Lemma allows us to reduce the problem of deciding whether there exists an accepting computation of $\mathcal{BP}$, to showing the existence of a finite lollipop-like witness with a special structure comprised of a stem ρ which is a finite path of $\mathcal{BP}$, and a pseudo-cycle which is a sequence v of transitions with an accepting state of $\mathcal{BP}$ having the following two properties (i) executing v returns each thread of the concurrent program to the same control location with the same symbol at the top of its stack as it started with, and (ii) executing it does not drain the stack of any thread, viz., any symbol that is not at the top of the stack of a thread to start with is not popped during the execution of the sequence. For ease of exposition we make the assumption that along all infinite runs of $\mathcal{BP}$ any lock that is acquired is eventually released. This restriction can be dropped in the same manner as in [12].

Theorem 6. (Dual Pumping Lemma). *$\mathcal{BP}$ has an accepting run starting from an initial configuration c if and only if there exist $\alpha \in \Gamma_1, \beta \in \Gamma_2$; $u \in \Gamma_1^*, v \in \Gamma_2^*$; an accepting configuration g; configurations lf_0, lf_1, lf_2 and lf_3 in which all locks are free; lock values $l_1, ..., l_m, l'_1, ..., l'_m$; control states $p', p''' \in P_1$, $q', q'' \in P_2$; $u', u'', u''' \in \Gamma_1^*$; and $v', v'', v''' \in \Gamma_2^*$ satisfying the following conditions*

1. $c \Rightarrow (\langle p, \alpha u\rangle, \langle q', v'\rangle, l_1, ..., l_m)$
2. $(\langle p, \alpha\rangle, \langle q', v'\rangle, l_1, ..., l_m) \Rightarrow lf_0 \Rightarrow (\langle p', u'\rangle, \langle q, \beta v\rangle, l'_1, ..., l'_m)$
3. $(\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$
   $\Rightarrow lf_1 \Rightarrow g \Rightarrow lf_2$
   $\Rightarrow (\langle p, \alpha u''\rangle, \langle q'', v''\rangle, l_1, ..., l_m) \Rightarrow lf_3$
   $\Rightarrow (\langle p''', u'''\rangle, \langle q, \beta v'''\rangle, l'_1, ..., l'_m)$

Let ρ, σ, ν be the sequences of global configurations realizing conditions 1, 2 and 3, respectively (see fig. 2). We first define sequences of transitions spliced from ρ, σ and ν that we will concatenate to construct an accepting path of $\mathcal{BP}$: (1) $\mathbf{l_{11}}$: the local sequence of T_1 fired along σ. (2) $\mathbf{l_{12}}$: the local sequence of T_1 fired along ν between $c_{21} = (\langle p', u'\rangle, \langle q, \beta\rangle, l'_1, ..., l'_m)$ and lf_1. (3) $\mathbf{l_{13}}$: the local sequence of T_1 fired along ν between lf_2 and $c_{12} = (\langle p, \alpha u''\rangle, \langle q'', v''\ \rangle, l_1, ..., l_m)$. (4) $\mathbf{l_{21}}$: the local sequence of

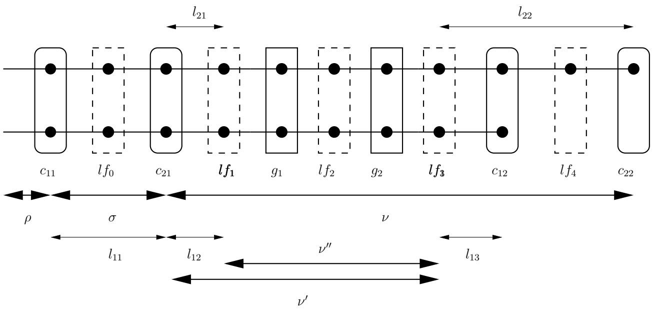


Fig. 2. Pumpable Witness

T_2 fired along ν between $c_{21} = (\langle p', u' \rangle, \langle q, \beta \rangle, l'_1, ..., l'_m)$ and lf_1. (5) $\mathbf{l_{22}}$: the local sequence of T_2 fired along ν between lf_2 and $c_{22} = (\langle p''', u''' \rangle, \langle q, \beta\ v''' \rangle, l_1, ..., l_m)$. (6) ν': the sequence of global transitions fired along ν till lf_2. (7) ν'': the sequence of global transitions fired along ν between lf_1 and lf_2.

Then $\pi : \rho\ \sigma\ \nu'\ (\ l_{13}\ l_{11}\ l_{12}\ l_{22}\ l_{21}\ \nu''\ )^{\omega}$ is a scheduling realizing an accepting valid run of $\mathcal{BP}$. Intuitively, thread T_1 is pumped by firing the sequence $l_{13}l_{11}l_{12}$ followed by the local computation of T_1 along ν''. Similarly, T_2 is pumped by firing the sequence $l_{22}l_{21}$ followed by the local computation of T_2 along ν''. The lock free configurations $lf_0, ..., lf_3$ are *breakpoints* that help in scheduling to ensure that π is a valid path. Indeed, starting at lf_2, we first let T_1 fire the local sequences l_{31}, l_{11} and l_{12}. This is valid as T_2 which currently does not hold any lock does not execute any transition and hence does not compete for locks with T_1. Executing these sequences causes T_1 to reach the local configuration of T_1 in lf_1 which is lock free. Thus T_2 can now fire the local sequences l_{22} and l_{21} to reach the local configuration of T_2 in lf_1 after which we let $\mathcal{CP}$ fire ν'' and then repeat the procedure.

It is worth noting that if the lock chains are unbounded in length then the existence of breakpoints as above is not guaranteed.

Constructing an LCA for $\overset{\infty}{\mathsf{F}}$. Conditions 1, 2 and 3 in the statement of the Dual Pumping Lemma can easily be re-formulated via a combination of $\cap$, $\cup$ and pre^*-closure computations for regular sets of configurations. This immediately implies that the computation of an LCA for $\overset{\infty}{\mathsf{F}}$ can be reduced to that for F, $\wedge$ and $\vee$ (see [12] for details).

Computation of X can be handled exactly as in [12].

5.1 The Model Checking Procedure for $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$

Given an LCA $\mathcal{L}_g$ accepting the set of states satisfying a formula g of $L(\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}})$, we formulated for each operator $\mathsf{Op} \in \{\mathsf{X}, \mathsf{F}, \overset{\infty}{\mathsf{F}}\}$, a procedure for computing an LCA $\mathcal{L}_{\mathsf{Op}g}$ accepting the set of all configurations that satisfy $\mathsf{Op}g$. Given a property f, by recursively applying these procedures starting from the atomic propositions and proceeding inside out in f we can construct the LCA $\mathcal{L}_f$ accepting the set of states of $\mathcal{CP}$ satisfying f In composing LCAs for different operators a technical issue that arises is of maintaining consistency across the various operators. This has already been handled before in the literature (see [1]). Finally, $\mathcal{CP}$ satisfies f if the initial global state of $\mathcal{CP}$ is accepted by $\mathcal{L}_f$.

6 Conclusion

Among prior work on the verification of concurrent programs, [7] attempts to generalize the techniques given in [5] to model check pushdown systems communicating via CCS-style pairwise rendezvous. However, since even reachability is undecidable for such a framework, the procedures are not guaranteed to terminate, in general, but only for certain special cases, some of which the authors identify. The key idea here is to restrict interaction among the threads so as to bypass the undecidability barrier. Another natural way to obtain decidability is to explore the state space of the given concurrent multi-threaded program for a bounded number of context switches among the threads both for model checking [17, 3] and dataflow analysis [16] or by restricting the allowed set of schedules [2].

The framework of Asynchronous Dynamic Pushdown Networks has been proposed recently [6]. It allows communication via shared variables which makes the model checking problem undecidable. Decidability is ensured by allowing only a bounded number of updates to the shared variables. Networks of pushdown systems with varying topologies for which the reachability problem is decidable have also been studied [4]. Dataflow analysis for asynchronous programs wherein threads can fork off other threads but where threads are not allowed to communicate with each other has also been explored [19, 9] and was shown to be EXPSPACE-hard, but tractable in practice.

In this paper, we have identified fragments of LTL for which the model checking problem is decidable for threads interacting via bounded lock chains thereby delineating precisely the decidability boundary for the problem. A desirable feature of our technique is that it enables compositional reasoning for the concurrent program at hand thereby ameliorating the state explosion problem. Finally, our new results enable us to provide a more refined characterization of the decidability of LTL model checking in terms of boundedness of lock chains as opposed to nestedness of locks.

References

[1] http://www.cs.utexas.edu/users/kahlon/papers/concur11.pdf

[2] Atig, M.F., Bouajjani, A.: On the Reachability Problem for Dynamic Networks of Concurrent Pushdown Systems. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 1–2. Springer, Heidelberg (2009)

[3] Atig, M.F., Bouajjani, A., Qadeer, S.: Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)

[4] Atig, M.F., Touili, T.: Verifying Parallel Programs with Dynamic Communication Structures. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 145–154. Springer, Heidelberg (2009)

[5] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

[6] Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. FSTTCS, pp. 348–359. Springer, Heidelberg (2005)

[7] Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: IJFCS, vol. 14(4), p. 551 (2003)

[8] Emerson, E.A.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, vol. B, pp. 997–1072 (1998)

[9] Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL (2007)

[10] Kahlon, V.: Boundedness vs. Unboundedness of Lock Chains: Characterizing CFL-Reachability of Threads Communicating via Locks. In: LICS (2009)

[11] Kahlon, V., Gupta, A.: An Automata-Theoretic Approach for Model Checking Threads for LTL Properties. In: LICS (2006)

[12] Kahlon, V., Gupta, A.: On the Analysis of Interacting Pushdown Systems. In: POPL (2007)

[13] Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)

[14] Kidd, N., Lammich, P., Touili, T., Reps, T.W.: A decision procedure for detecting atomicity violations for communicating processes with locks. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 125–142. Springer, Heidelberg (2009)

[15] Lal, A., Balakrishnan, G., Reps, T.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)

[16] Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural Analysis of Concurrent Programs Under a Context Bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)

[17] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

[18] Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. In: ACM TOPLAS (2000)

[19] Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)

A Temporal Logic for the Interaction of Strategies[⋆,⋆⋆,⋆⋆⋆]

Farn Wang[1,2], Chung-Hao Huang[2], and Fang Yu[3]

[1] Dept. of Electrical Engineering, National Taiwan University
[2] Graduate Institute of Electronic Engineering, National Taiwan University
[3] Dept. of Management Information Systems, National Chengchi University

Abstract. We propose an extension to *ATL* (*alternating-time logic*), called *BSIL* (*basic strategy-interaction logic*), for the specification of interaction among the strategies of agents in a multi-agent system. BSIL allows for the specifications of one system strategy that can cooperate with several strategies of the environment for different requirements. We argue that such properties are important in practice and rigorously show that such properties are not expressible in *ATL**, *GL* (*game logic*), and *AMC* (*alternating μ-calculus*). Specifically, we show that BSIL is more expressive than ATL but incomparable with ATL*, GL, and AMC in expressiveness. We show that a memoryful strategy is necessary for fulfilling a specification in BSIL. We also show that the model-checking problem of BSIL is PSPACE-complete and is of lower complexity than those of ATL*, GL, AMC, and the general strategy logics. This may imply that BSIL can be useful in closing the gap between real-world projects and the game-theoretical results. We then show the plausibility and feasibility of our techniques by reporting our implementation and experiment with our PSPACE model-checking algorithm for BSIL. Finally, we discuss an extension of BSIL.

Keywords: games, turn-based, logic, model-checking, expressiveness.

1 Introduction

The specification and verification of open systems focuses on the design of system interfaces that allow for the fulfillment of various functions and prevent other bad behaviors from happening. The theoretical challenges in designing such systems have drawn the attention of researchers in game theory. From the perspective of game theory, the design problem of such open systems can be modeled as a multi-agent game. Some players represent the system while other players represent the environment (or the users). The system wins the game in an execution (or a *play* in the jargon of game theory) if all the system specifications are fulfilled along the execution. The goal of the system design, from the perspective of game theory, is to design a computable strategy of the system

[⋆] The work is partially supported by NSC (Grant NSC 97-2221-E-002-129-MY3, Taiwan, ROC) and by Research Center for Information Technology Innovation, Academia Sinica, Taiwan, ROC.

[⋆⋆] The authors would like to thank Prof. Moshe Vardi for his valuable suggestions.

[⋆⋆⋆] A full version of the work with all the lemma proofs can be found in http://cc.ee.ntu.edu.tw/~farn.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 466–481, 2011.

that enforces all the system specifications. Such a strategy is called a *winning strategy* for the system.

At the moment, there are various game-theoretical languages, including *ATL* (*alternating-time logic*), *ATL**, *AMC* (*alternating μ-calculus*), *GL* (*game logic*) [1], and *SL* (*strategy logics*) [4,3,6], for the specification of open systems. Each language also comes with a verification algorithm that helps in deciding whether a winning strategy for the system exists. However, there is a gap between the need of the industry and the technology to offer from the previous research. Frankly speaking, none of those languages represents a proper combination of expressiveness for close interaction among agent strategies and efficiency for specification verification. ATL, ATL*, AMC, and GL [1] allow us to specify that some players together have a strategy to fulfill something. This is far from what the industry need in specification. Consider the following example of a banking system.

Example 1. **Banking system** A bank needs to specify that their banking system, embodied as a system strategy, allows a client to use a strategy to withdraw money, to use a strategy to deposit money, and to use a strategy to query for balance. Moreover, the same system strategy should forbid any illegal operation on the banking system. Specifically, the same system strategy must accommodate all the client's strategies for good behaviors while prevent client's strategy for bad behaviors from damaging the system. We actually prove in this work that *ATL* (*alternating-time logic*), *ATL**, *AMC* (*alternating μ-calculus*), and *GL* (*game logic*) [1] do not support the specifications in this regard. For example, it is not possible to specify with those languages that the system strategies used both in a withdrawal transaction and in a deposit transaction must be the same. As a result, the verification algorithms of those languages actually do not help as much as we wish in verifying real-world open systems. ■

To solve the expressiveness problem in the above example, strategy logics were proposed in [4,3,6] that allow for the flexible quantification of high-order strategy variables in logic formulas. However, their verification complexities are prohibitively high and hinder them from practical application. In retrospection, the specification problem of the above-mentioned properties has a deep root in game theory. Consider a game among 3 prisoners initially in jail.

Example 2. **Prisoners' dilemma** A prisoner may deny charges or may cooperate with the police. If all deny, they are all acquitted of the charges. If more than one choose to cooperate, all will stay in jail. If all but one deny, then all will be in jail except the sole cooperating one will be a dirty witness and be acquitted. We may want to specify that the three prisoners may collaborate with each other, will all deny, and will not be in jail. Let j_a be the proposition for prisoner a in jail. This can be expressed respectively in Alur, Henzinger, and Kupferman's ATL, ATL*, GL, and AMC [1] as follows.[1]

$$\begin{array}{ll} \text{ATL, ATL}^*\text{:} & \langle\{1,2,3\}\rangle \bigwedge_{a\in[1,3]} \Diamond\neg j_a \\ \text{GL:} & \exists\{1,2,3\}.\bigwedge_{a\in[1,3]} \forall\Diamond\neg j_a \\ \text{AMC:} & \mathbf{lfp}X.\langle\{1,2,3\}\rangle \bigcirc \bigwedge_{a\in[1,3]}(X \vee \neg j_a) \end{array}$$

[1] Note that the three example formulas are not equivalent.

Here "$\langle\{1,2,3\}\rangle$" and "$\exists\{1,2,3\}$" are both existential quantifiers on the collaborative strategy among prisoners $1,2$, and 3. Such a quantifier is called a *strategy quantifier* (*SQ*) for convenience. "**lfp**" is the least fixpoint operator. Even though we can specify strategies employed by sets of prisoners and there is a natural relationship (containment) between sets with such logics, there is no way to relate strategies to each other. For example, if prisoners 1 and 2 are really loyal to prisoner 3, they can both deny the charges, make sure that prisoner 3 will not be in jail, and let prisoner 3 to decide whether they will be in jail. The research of strategies for related properties has a long history in game theory. If we recall example 1, it is easy to see the similarity and link between this example and the banking system specification problem. This observation may suggest that finding a language with an appropriate balance between the expressiveness and the verification complexity is still a central challenge yet to be overcome. ■

To meet the challenge, we propose an extension to ATL, called *BSIL* (*basic strategy-interaction logic*), by introducing a new modal operator called *strategy interaction quantifier* (*SIQ*). In the following, we use several examples in the prisoners' dilemma to explain BSIL. A formula for the property in example 2 follows.

$$\langle\{1,2\}\rangle((\langle+\emptyset\rangle\Diamond\neg j_3)\wedge(\langle+\{3\}\rangle\Diamond\neg(j_1\vee j_2))\wedge\langle+\{3\}\rangle\Box(j_1\wedge j_2))$$

Here "$\langle+\{3\}\rangle$" is an existential SIQ on strategies of prisoner 3 for collaborating with the strategies of prisoners declared in the parent formula. Similarly, "$\langle+\emptyset\rangle$" means that no collaboration of any prisoner is needed. We also call an SIQ an SQ. In BSIL formulas, we specifically require that no SIQ can appear as a topmost SQ in a path subformula.

If prisoner 1 really hates the others, he can always cooperate with the police, make sure that prisoners 2 and 3 will be in jail, and let them decide whether he will be in jail. This property can be expressed in BSIL as follows.

$$\langle\{1\}\rangle((\langle+\emptyset\rangle\Box(j_2\wedge j_3))\wedge(\langle+\{2,3\}\rangle\Diamond\neg j_1)\wedge\langle+\{2,3\}\rangle\Box j_1)$$

At last, we can also use BSIL to express the deterministic Nash equilibrium, a scenario in which a unilateral change of actions by a prisoner does not improve her/his payoff. In the example, an equilibrium is that all prisoners keep on cooperating with the police. Such an equilibrium results in the following property in BSIL.

$$\langle\{1,2,3\}\rangle((\textstyle\bigwedge_{a\in[1,3]}\langle+\emptyset\rangle\Box j_a)\wedge\bigwedge_{a\in[1,3]}\langle+\{a\}\rangle\Box\bigwedge_{b\in[1,3];b\neq a}j_b)$$

Note that we let the inner quantifications "$\langle+\{1\}\rangle$", "$\langle+\{2\}\rangle$", and "$\langle+\{3\}\rangle$" to overrule the strategy binding by "$\langle\{1,2,3\}\rangle$."

In this work, we establish that BSIL is incomparable with ATL*, GL, and AMC in expressiveness. Although strategy logics proposed by Costa, Laroussinie, Markey ite-CLM10, Chatterjee, Henzinger, Piterman [3], Mogavero, Murano, and Vardi [6] are superclasses to BSIL with their flexible quantification of strategies and binding to strategy variables, their model-checking[2] complexity are all doubly exponential time hard. In contrast, BSIL enjoys a PSPACE-complete model-checking complexity for turn-based game graphs. This may imply that BSIL could be a better balance between expressiveness and verification efficiency than ATL*, GL, AMC [1], and SL [3,6]. Moreover, the

[2] A model-checking problem is to check whether a given model (game graphs in this work) satisfies a logic formulas (in ATL and its extensions in this work).

deterministic Nash equilibria expressed in BSIL in the above may also imply that BSIL could be a valuable and useful subclass of strategy logics [3,6].[3]

We also establish some other properties of BSIL. We show that the strategies for BSIL properties against turn-based games need be memoryful. We prove that the BSIL model-checking problem is PSPACE-complete. We have also implemented our model-checking algorithm and carried out experiments. Finally, by lifting the restriction that no SIQ may appear as a topmost SQ in a path subformula, we can extend BSIL to *strategy interaction logic* (*SIL*) by allowing the SIQs to be used directly after temporal modal operators. For example, we may have the following SIL formula.

$$\langle\{1\}\rangle\Box((p \rightarrow \langle +\{2\}\rangle \bigcirc q) \wedge (\neg p \rightarrow [+\{2\}] \bigcirc \neg q))$$

SIL has the expressiveness for the interaction of a player's stratregy with infinitely many strategies used by players at different states along a play. We also show that SIL is strictly more expressive than ATL* and its model-checking problem is doubly exponential time hard.

Here is our presentation plan. Section 2 explains turn-based game graphs for the description of multi-agent systems and presents BSIL. Section 3 shows that BSIL is more expressive than ATL [1] but not comparable with ATL*, AMC, and GL [1] in expressiveness. Section 4 shows that BSIL model-checking problem needs memoryful strategies and is PSPACE-hard. Section 5 presents a PSPACE algorithm for BSIL model-checking and establishes that BSIL model-checking problem is PSPACE-complete. Section 6 extends BSIL to SIL. Section 7 is the conclusion.

2 System Models and BSIL

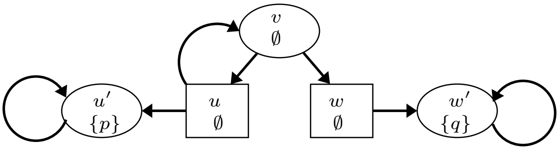
2.1 Turn-Based Game Graphs

A *turn-based* game is played by many agents. Assume that the number of agents is m and we index the agents with integers 1 through m. It is formally presented as a tuple $A = \langle m, Q, q_0, \omega, P, \lambda, R\rangle$ with the following restrictions. m is the number of agents in the game. Q is a finite set of states. $q_0 \in Q$ is the *initial state* of A. $\omega : Q \mapsto [1, m]$ is a function that specifies the owner of each state. Only the owner of a state makes choice at the state. P is a finite set of atomic propositions. $\lambda : Q \mapsto 2^P$ is a proposition labeling function. $R \subseteq Q \times Q$ is the set of transitions. In figure 1, we have the graphical representation of a turn-based game graph with initial state v. The ovals and squares represent states while the arcs represent state transitions. We also put down the λ values inside the corresponding states.

A state predicate of P is a Boolean combination of elements in P. We let $SP(P)$ be the set of state predicates of P. The satisfaction of a state predicate η at a state q, in symbols $q \models \eta$, is defined in a standard way.

For convenience, given a game graph $A = \langle m, Q, q_0, \omega, P, \lambda, R\rangle$, we denote $m, Q, q_0, \omega, P, \lambda$, and R respectively as $m_A, Q_A, q_{0A}, \omega_A, P_A, \lambda_A$, and R_A.

[3] Another work worthy of mentioning is the stochastic game logic (SGL) by Baier, Brázdil Gröser, and Kucera [2] with limited expressiveness for strategy interaction. However, for memoryful strategies, the model-checking problem of SGL is undecidable.



○ belongs to agent 1 and □ belongs to agent 2.

Fig. 1. A turn-based game graph

A *play* is an infinite path in a game graph. A play is *initial* if it begins with the initial state. Given a play $\rho = \bar{q}_0\bar{q}_1\ldots$, for every $k \geq 0$, we let $\rho(k) = \bar{q}_k$. Also, given $h \leq k$, we let $\rho[h,k]$ denote $\rho(h)\ldots\rho(k)$ and $\rho[h,\infty)$ denote the infinite tail of ρ from $\rho(h)$. A *play prefix* is a finite segment of a play from the beginning of the play. Given a play prefix $\rho = \bar{q}_0\bar{q}_1\ldots\bar{q}_n$, we let $|\rho| = n+1$. For convenience, we use $last(\rho)$ to denote the last state in ρ, i.e., $\rho(|\rho|-1)$.

For an agent $a \in [1,m]$, a *strategy* σ for a is a function from Q_A^* to Q_A such that for every $\rho \in Q_A^*$, $\sigma(\rho) \in Q_A$ with $(last(\rho), \sigma(\rho)) \in R_A$. A set of agents is called an *agency*. A *congregate strategy* (or *C-strategy*) Σ of an agency $M \subseteq [1,m]$ is a partial function from $[1,m]$ to the set of strategies such that for every $a \in [1,m]$, $a \in M$ iff[4] $\Sigma(a)$ is defined. The composition of two C-strategies Σ, Π, in symbols $\Sigma \circ \Pi$, is defined with the following restrictions for every $a \in [1,m]$.

- If $\Pi(a)$ is defined, then $\Sigma \circ \Pi(a) = \Pi(a)$.
- If $\Sigma(a)$ is defined and $\Pi(a)$ is undefined, then $\Sigma \circ \Pi(a) = \Sigma(a)$.
- If $\Sigma(a)$ and $\Pi(a)$ are both undefined, then $\Sigma \circ \Pi(a)$ is also undefined.

Later, we will use composition of C-strategies to model inheritance of strategy bindings from ancestor formulas.

A play ρ is compatible with a strategy σ of an agent $a \in [1,m]$ iff for every $k \in [0,\infty)$, $\omega(\rho(k)) = a$ implies $\sigma(\rho[0..k]) = \rho(k+1)$. The play is compatible with a C-strategy Σ of agency M iff for every $a \in M$, the play is compatible with $\Sigma(a)$ of a.

2.2 BSIL Syntax

For a turn-based game graph A of m agents, we have three types of formulas: *state formulas*, *tree formulas*, and *path formulas*. State formulas describe properties of states. Tree formulas describe interaction of strategies. Path formulas describe properties of traces. A BSIL formula ϕ is constructed with the following three syntax rules, respectively for state formula ϕ, tree formula θ, and path formula ψ.

$$\begin{aligned}\phi &::= p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle M\rangle\theta \mid \langle M\rangle\psi\\ \theta &::= \neg\theta_1 \mid \theta_1 \vee \theta_2 \mid \langle +M\rangle\theta_1 \mid \langle +M\rangle\psi\\ \psi &::= \bigcirc\phi \mid \phi_1\mathrm{U}\phi_2 \mid \phi_1\mathrm{W}\phi_2\end{aligned}$$

Here p is an atomic proposition in P_A. M is a subset of $[1,m]$. $\langle M\rangle\psi$ means that there exist strategies of the agents in M that make all plays satisfy ψ. Formulas $\langle +L\rangle\theta_1$ and $\langle +L\rangle\psi$ must happen as subformulas of a state formula $\langle M\rangle\theta$. Intuitively, they mean that there exist strategies of L that work with the strategies bound to $\langle M\rangle\theta$ to make θ_1 and ψ true respectively.

[4] "*iff*" is a shorthand for "*if and only if*."

Formulas ϕ are called *BSIL formulas*. Note that we strictly require that strategy interaction cannot cross path modal operators. This restriction is important and allows us to analyze the interaction of strategies locally in a state and then enforce the interaction along all paths from the state. For convenience, we also have the following shorthands.

$$\begin{aligned}
\mathit{true} &\equiv p \vee (\neg p) & \mathit{false} &\equiv \neg \mathit{true}\\
\phi_1 \wedge \phi_2 &\equiv \neg((\neg\phi_1) \vee (\neg\phi_2)) & \phi_1 \Rightarrow \phi_2 &\equiv (\neg\phi_1) \vee \phi_2\\
\Diamond\phi_1 &\equiv \mathit{true}\, \mathbf{U}\phi_1 & \Box\phi_1 &\equiv \phi_1\mathbf{W}\mathit{false}\\
[M]\bigcirc\phi_1 &\equiv \neg\langle M\rangle \bigcirc \neg\phi_1 & [+M]\bigcirc\phi_1 &\equiv \neg\langle +M\rangle \bigcirc \neg\phi_1\\
[M]\phi_1\mathbf{U}\phi_2 &\equiv \neg\langle M\rangle\neg\phi_2\mathbf{W}\neg(\phi_1 \vee \phi_2) & [+M]\phi_1\mathbf{U}\phi_2 &\equiv \neg\langle_M\rangle\neg\phi_2\mathbf{W}\neg(\phi_1 \vee \phi_2)\\
[M]\phi_1\mathbf{W}\phi_2 &\equiv \neg\langle M\rangle\neg\phi_2\mathbf{U}\neg(\phi_1 \vee \phi_2) & [+M]\phi_1\mathbf{W}\phi_2 &\equiv \neg\langle_M\rangle\neg\phi_2\mathbf{U}\neg(\phi_1 \vee \phi_2)
\end{aligned}$$

Operators $\langle\ldots\rangle$ and $[\ldots]$ are all *SQ*s. Operators $\langle+\ldots\rangle$ and $[+\ldots]$ are *SIQ*s. All BSIL formulas are *well-formed* since all their SIQs occur inside an SQ beginning with $\langle M\rangle$ for some M.

2.3 BSIL Semantics

BSIL subformulas are interpreted with respect to C-strategies. A state or a tree formula ϕ is satisfied at a state q with C-strategy Σ, in symbols $A, q \models_\Sigma \phi$, if and only if the following inductive constraints are satisfied.

- $A, q \models_\Sigma p$ iff $p \in \lambda(q)$.
- For state or tree formula ϕ_1, $A, q \models_\Sigma \neg\phi_1$ iff $A, q \models_\Sigma \phi_1$ is false.
- For state or tree formulas ϕ_1 and ϕ_2, $A, q \models_\Sigma \phi_1 \vee \phi_2$ iff either $A, q \models_\Sigma \phi_1$ or $A, q \models_\Sigma \phi_2$.
- $A, q \models_\Sigma \langle M\rangle\theta$ iff there exists a C-strategy Π of M with $A, q \models_\Pi \theta$.
- $A, q \models_\Sigma \langle +M\rangle\theta$ iff there exists a C-strategy Π of M with $A, q \models_{\Sigma\circ\Pi} \theta$. Here function composition $\Sigma \circ \Pi$ models inheritance of strategy bindings Σ from ancestor formulas.
- $A, q \models_\Sigma \langle M\rangle\psi$ iff there exists a C-strategy Π of M such that for all plays ρ from q compatible with Π, $\rho \models_\Pi \psi$ which means that ρ satisfies path formula ψ with C-strategy Π.
- $A, q \models_\Sigma \langle +M\rangle\psi$ iff there exists a C-strategy Π of M such that for all plays ρ from q compatible with $\Sigma \circ \Pi$, $\rho \models_{\Sigma\circ\Pi} \psi$.

Note that we also let a play ρ satisfy a path formula ψ with the inheritance of a C-strategy. This is in fact not necessary for BSIL semantics since all such inheritance will be overruled by SQs immediately following the temporal modal operators. However, this is necessary when we later extend BSIL by allowing for the inhertance of strategies across the temporal modal operators.

A play ρ satisfies a path formula ψ with C-strategy Σ, in symbols $\rho \models_\Sigma \psi$, iff the following restrictions hold.

- $\rho \models_\Sigma \bigcirc\phi_1$ iff $A, \rho(1) \models_\Sigma \phi_1$.
- $\rho \models_\Sigma \phi_1\mathbf{U}\phi_2$ iff there exists an $h \geq 0$ with $A, \rho(h) \models_\Sigma \phi_2$ and for all $j \in [0, h)$, $A, \rho(j) \models_\Sigma \phi_1$.
- $\rho \models_\Sigma \phi_1\mathbf{W}\phi_2$ iff either $\rho \models_\Sigma \phi_1\mathbf{U}\phi_2$ or for all $h \geq 0$, $A, \rho(h) \models_\Sigma \phi_1$.

For convenience, we let $\perp$ be a null C-strategy, i.e., a function that is undefined on everything. If ϕ_1 is a BSIL (state) formula and $A, q \models_{\perp} \phi_1$, then we may simply write $A, q \models \phi_1$. If $A, q_0 \models \phi_1$, then we also write $A \models \phi_1$.

3 Expressiveness of BSIL

In this section, we establish that BSIL is incomparable with ATL*, AMC, GL [1] in expressiveness. It is easy to see that BSIL is a super-class of ATL. Thus we have the following lemma.

Lemma 1. *BSIL is at least as expressive as ATL.* ■

Lemmas 2 and 3 establish that ATL* and BSIL are incomparable.

Lemma 2. *For every ATL* formula ϕ, there are two game graphs that ϕ cannot distinguish while $\langle\{1\}\rangle((\langle+\{2\}\rangle\Box p) \wedge \langle+\{2\}\rangle\Box q)$ can.*
Proof : The proof is by an inductive construction of two families $A_0, \ldots, A_k, \ldots$ and $B_0, \ldots, B_k, \ldots$ of game graphs such that no ATL* formula with k SQs can distinguish A_k and B_k. ■

Lemmas 1 and 2 together establish that ATL is strictly less expressive than BSIL.

Lemma 3. *ATL* formula $\langle\{1\}\rangle\Box\Diamond p$ is not equivalent to any BSIL formula.*
Proof : The proof is similar to the proof for the inexpressibility of $\langle\{1\}\rangle\Box\Diamond p$ with ATL [1]. ■

The following two lemmas show the relation between GL and BSIL.

Lemma 4. *For every GL formula ϕ, there are two game graphs that ϕ cannot distinguish while $\langle\{1\}\rangle((\langle+\{2\}\rangle\Box p) \wedge \langle+\{2\}\rangle\Box q)$ can.*
Proof : The proof is similar to the one for lemma 2. ■

Lemma 5. *GL formula $\exists\!\!\exists\{1\}.((\exists\Box p) \wedge \exists\Box q)$ is not equivalent to any BSIL formula.*
Proof : The proof basically follows the same argument in [1] that $\exists\!\!\exists\{1\}.((\exists\Box p) \wedge \exists\Box q)$ is not equivalent to any ATL* formula. ■

To establish that AMC is not as expressive as BSIL, we basically follow the proof style for lemma 2 and use the same two families of game graphs. The statement of the lemma requires notations for proposition variables and other details in AMC.

Lemma 6. *For every AMC formula ϕ, there are two game graphs that ϕ cannot distinguish while $\langle\{1\}\rangle((\langle+\{2\}\rangle\Box p) \wedge \langle+\{2\}\rangle\Box q)$ can.*
Proof : The proof is similar to the one for lemma 2. ■

By the same argument in [1], for one-agent game, BSIL coincides with CTL and is not as expressive as AMC.

Lemma 7. *For game graphs of one agent, AMC is strictly more expressive than BSIL.*
Proof : For one-agent games, AMC is equivalent to μ-calculus and BSIL is equivalent to CTL which is strictly less expressive than μ-calculus. ■

A comment on lemmas 2, 4, and 6 is that the path modal formulas in the lemmas can be changed independently to $\Diamond p$ and $\Diamond q$ without affecting the validity of the lemma. This can be used to show that the example properties in the introduction are indeed inexpressible in ATL*, GL, and AMC.

4 Some Properties of BSIL Model Checking Problem

In this section, we first show that a strategy synthesizing a BSIL formula needs memory. A strategy σ is *memory-less* iff for every two play prefixes ρ and ρ', $\mathit{last}(\rho) = \mathit{last}(\rho')$ implies $\sigma(\rho) = \sigma(\rho')$.

Lemma 8. *There is a BSIL model-checking problem instance that cannot be satisfied with a memory-less strategy of an agent.*

Proof : Consider the 2-agent game graph in figure 1. Again we use ovals to represent those nodes owned by agent 1 and squares to represent those by agent 2. We want to check property $\langle\{1\}\rangle((\langle +\{2\}\rangle\Diamond p) \wedge (\langle +\{2\}\rangle\Diamond q))$. It is clear that no memory-less strategy of agent 1 satisfies this property. However, a strategy of agent 1 that chooses arc (v, u) at least once and eventually chooses arc (v, w) satisfies the BSIL property. ■

In the following, we establish the complexity lower-bound of the model-checking problem of BSIL formulas. This is done by reducing the QBF (quantified Boolean formula) satisfiability problem [5] to BSIL model-checking problem of 3-agent game graphs. We assume a QBF property $\eta \equiv \mathcal{Q}_1 p_1 \ldots \mathcal{Q}_l p_l (C_1 \wedge C_2 \wedge \ldots \wedge C_n)$ with a set $P = \{p_1, \ldots, p_l\}$ of atomic propositions and the following restrictions.

- For each $k \in [1, l]$, $\mathcal{Q}_k$ is either $\exists$ or $\forall$.
- For each $k \in [1, n]$, C_k is a clause $l_{k,1} \vee \ldots \vee l_{k,h_k}$ where for each $j \in [1, h_k]$, $l_{k,j}$ is a *literal*, i.e., either an atomic proposition or a negated atomic proposition.

Intuitively, the reduction is to translate the QBF formula to a game graph and a BSIL formula for a traversal requirement on the game graph. The game graph has some special components corresponding to the truth values of each atomic proposition. Suppose that Γ_p and $\Gamma_{\neg p}$ respectively represent the subgraphs for the truth and falsehood of an atomic proposition p. The rest of the game graph is partitioned into subgraphs Ω_p responsible for the interpretation of atomic proposition p for all $p \in P$. Then the QBF formula actually can be interpreted as a requirement for covering those Γ_p's and $\Gamma_{\neg p}$'s with the decisions in those Ω_p's. For example, the following formula $\eta \equiv \exists p \forall q \exists r((p \vee q \vee r) \wedge (\neg p \vee \neg r))$ can be read as "*there exists a decision in Ω_p such that for every decision in Ω_q, there exists a decision in Ω_r such that*

- *one of Γ_p, Γ_q, and Γ_r is covered; and*
- *one of $\Gamma_{\neg p}$ and $\Gamma_{\neg r}$ is covered.*

Of course, we have to make sure that Γ_p and $\Gamma_{\neg p}$ cannot be covered in the same play for each p. The details of constructing those Γ_p's, $\Gamma_{\neg p}$'s, and Ω_p's can be found in the proof of the following lemma that establishes the PSPACE complexity lower-bound.

Lemma 9. *BSIL model-checking problem for turn-based game graphs is PSPACE-hard.* ■

5 BSIL Model-Checking Algorithm

There are two restrictions on BSIL formulas that enable us of the design of a PSPACE model-checking algorithm. Firstly, similar to ATL and CTL, each path modal operator must occur immediately after an SQ. Secondly, an SIQ cannot occur as a topmost SQ in a path formula. These two restrictions together suggest that as in the model-checking algorithms of ATL [1], we can evaluate the proper subformulas of path modal formulas independently and then treat them as auxiliary propositions. Moreover, as in the evaluation of $\langle\{\ldots\}\rangle\Diamond$-formulas in ATL model-checking, if a $\Diamond$-formula can be enforced with a strategy, it can be enforced in a finite number of steps along every play compatible with the strategy in a computation tree. Once the bound b of this "finite number" of steps is determined, then we can enumerate all the strategies embedded in the computation tree up to depth b and try to find one that enforces a BSIL formula.

One thing in exploring a computation tree for BSIL different from that for ATL [1] is that we have to consider the interaction of strategies. For example, we may have a subformula $\langle\{1\}\rangle((\langle+\{2\}\rangle\Diamond p)\wedge\langle+\{2\}\rangle\Box q)$ to enforce. Then in exploring the computation tree, we may follow two strategies of agent 2, one to enforce $\Diamond p$ and the other to enforce $\Box q$, that always make the same decision until we reach a tree node v owned by agent 2. This can be conceptualized as passing the obligations of $\Diamond p$ and $\Box q$ along the path from the root to v. Then at node v, the two strategies may differ in their decisions and pass down the two obligations to different branches. In subsection 5.1, we explain some basic concepts in labeling the children with path obligations passed down from their parent in a computation tree to obey the interaction among strategies declared in a BSIL formula.

Then in subsection 5.2, we present our algorithm into two parts, one for the checking of BSIL state formulas and the other for that of BSIL tree formulas. In subsection 5.3, we prove the correctness of the algorithm. In subsection 5.4, we show that our algorithm is in PSPACE.

5.1 Computing Path Obligations Passed Down Computation Tree

We need some special techniques in checking tree formulas. We adopt the concept of strategy variables from [3,6]. Given $\{a_1,\ldots,a_n\}\subseteq[1,m]$, we use $\{a_1\mapsto s_1,\ldots,a_n\mapsto s_n\}$ to denote the *strategy variable binding* (*SV-binding* for short) that binds agents $a_1,\ldots,a_n$ respectively to strategy variables $s_1,\ldots,s_n$. Given an SV-binding B, $B\circ\{a_1\mapsto s_1,\ldots,a_n\mapsto s_n\}$ is the SV-binding that is identical to B except that agent a_i is bound to s_i for every $i\in[1,n]$. Given an agency $M\subseteq[1,m]$, $B^{\neg M}$ is defined as the subset of B defined on $[1,m]-M$. Specifically, $B^{\neg M}$ is $\{a\mapsto s\mid a\mapsto s\in B, a\notin M\}$. Also, we let *def*$(B)$ be the index set of agents with a binding in B.

Given an SV-binding B, and a state, tree, or path formula ϕ, $B\phi$ is called a *bound formula*. $B\phi$ is a *bound literal* if ϕ is a path formula. A Boolean combination of bound literals is called a *Boolean bound formula* (*BB-formula*). The strategy variables in BB-formula are only used to tell whether two path properties are to be enforced with the same strategy. For example, property $\langle\{1\}\rangle((\langle+\{2\}\rangle\Diamond p)\wedge\langle+\{2\}\rangle\Diamond q)$ can be rewritten as BB-formula $(\{1\mapsto s_1,2\mapsto s_2\}\Diamond p)\wedge\{1\mapsto s_1,2\mapsto s_3\}\Diamond q$ which says that agent 1 must use the same strategy to fulfill both $\Diamond p$ and $\Diamond q$ while agent 2 may use different strategies to fulfill the two path properties.

Suppose that we are given a BB-formula ϕ, with strategy variables $s_1,\ldots,s_n$, and a function π that assigns each of $s_1,\ldots,s_n$ a strategy. Similar to the semantics of strategy logics [6] with strategy variables, we can also define the satisfaction of ϕ at a state q with π, in symbols $A,q\models^{\pi}\phi$, as follows.

Table 1. Rewriting rules for BB-formulas

$\mathit{bf}(B\neg\neg\phi)$	$\equiv \mathit{bf}(B\phi)$
$\mathit{bf}(\neg B(\phi_1 \vee \phi_2))$	$\equiv \mathit{bf}(B\neg\phi_1) \wedge \mathit{bf}(B\neg\phi_2)$
$\mathit{bf}(\neg B(\phi_1 \wedge \phi_2))$	$\equiv \mathit{bf}(B\neg\phi_1) \vee \mathit{bf}(B\neg\phi_2)$
$\mathit{bf}(B\neg\langle M\rangle\phi)$	$\equiv \mathit{bf}(B[M]\neg\phi)$
$\mathit{bf}(B\neg[M]\phi)$	$\equiv \mathit{bf}(B\langle M\rangle\neg\phi)$
$\mathit{bf}(B[M]\phi)$	$\equiv \mathit{bf}(B\langle[1,m]-M\rangle\phi)$
$\mathit{bf}(B\langle\{a_1,\ldots,a_n\}\rangle\phi)$	$\equiv \mathit{bf}(\{a_1 \mapsto \mathit{newvar}(),\ldots,a_n \mapsto \mathit{newvar}()\}\phi)$
$\mathit{bf}(B\neg\langle +M\rangle\phi)$	$\equiv \mathit{bf}(B[+M]\neg\phi)$
$\mathit{bf}(B\neg[+M]\phi)$	$\equiv \mathit{bf}(B\langle +M\rangle\neg\phi)$
$\mathit{bf}(B[+M]\phi)$	$\equiv \mathit{bf}(B^{\neg M} \circ \{a_1 \mapsto \mathit{newvar}(),\ldots,a_n \mapsto \mathit{newvar}()\}\phi)$, if $[1,m]-(\mathit{def}(B)\cup M)=\{a_1,\ldots,a_n\}$
$\mathit{bf}(B\langle +\{a_1,\ldots,a_n\}\rangle\phi)$	$\equiv \mathit{bf}(B \circ \{a_1 \mapsto \mathit{newvar}(),\ldots,a_n \mapsto \mathit{newvar}()\}\phi)$
$\mathit{bf}(B \bigcirc \phi)$	$\equiv B \bigcirc \mathit{bf}(\emptyset\phi)$
$\mathit{bf}(B\phi_1\mathrm{U}\phi_2)$	$\equiv B\mathit{bf}(\emptyset\phi_1)\mathrm{U}\mathit{bf}(\emptyset\phi_2)$
$\mathit{bf}(B\phi_1\mathrm{W}\phi_2)$	$\equiv B\mathit{bf}(\emptyset\phi_1)\mathrm{W}\mathit{bf}(\emptyset\phi_2)$
$\mathit{bf}(Bp) \equiv p$	; $\mathit{bf}(B\neg p) \equiv \neg p$
$\mathit{bf}(B\mathit{true}) \equiv \mathit{true}$	; $\mathit{bf}(B\mathit{false}) \equiv \mathit{false}$

- $A, q \models^{\pi} \phi_1 \vee \phi_2$ iff either $A, q \models^{\pi} \phi_1$ or $A, q \models^{\pi} \phi_2$.
- $A, q \models^{\pi} \phi_1 \wedge \phi_2$ iff both $A, q \models^{\pi} \phi_1$ and $A, q \models^{\pi} \phi_2$.
- Given an SV-binding B and a path formula ϕ_1 with a C-strategy $\Sigma = \{a \mapsto \pi(s) \mid a \mapsto s \in B\}$, $A, q \models^{\pi} B\phi_1$ iff for all plays ρ compatible with Σ from q, $\rho \models_{\Sigma} \phi_1$.

In table 1, we present equivalence rules to rewrite state, tree, and path formulas to BB-formulas with procedure *bf*(). For convenience, we need a procedure *newvar*() that returns a strategy variable that has not been used before. The following lemma shows the correctness of the rules in table 1.

Lemma 10. *Given a state q and a BSIL formula ϕ with strategy variables $s_1,\ldots,s_n$ in bf($\emptyset\phi$), $A, q \models_{\perp} \phi$ iff there is a function π such that $A, q \models^{\pi}$ bf($\emptyset\phi$).* ■

For convenience of algorithm presentation, we also assume that there is a procedure that rewrites a BB-formula to an equivalent BB-formula in disjunctive normal form. Specifically, a *disjunctive normal BB-formula* (*DNBB-formula*) is the disjunction of conjunctions of bound literals. The rewriting of a BB-formula ϕ to a DNBB-formula can be done with repetitive application of distribution law of conjunction to disjunction until no more change is possible.

Example 3. DNBB-formula rewriting: We have the following rewriting process for a BSIL formula for five agents.

$$\mathit{bf}\left(\emptyset\langle\{1\}\rangle\left(\begin{array}{l}\langle+\{2\}\rangle([+\{3\}](\langle+\{4\}\rangle\Box p)\vee[+\{2\}]\Diamond q)\\ \wedge\ [+\{3\}](\langle+2\rangle\Diamond r\wedge\langle+\{5\}\rangle\Box q)\end{array}\right)\right)$$

$$\equiv \mathit{bf}\left(\{1\mapsto s_1\}\left(\begin{array}{l}\langle+\{2\}\rangle([+\{3\}](\langle+\{4\}\rangle\Box p)\vee[+\{2\}]\Diamond q)\\ \wedge\ [+\{3\}](\langle+\{2\}\rangle\Diamond r\wedge\langle+\{5\}\rangle\Box q)\end{array}\right)\right)$$

$$\equiv \left(\begin{array}{l}\mathit{bf}(\{1\mapsto s_1\}\langle+\{2\}\rangle([+\{3\}](\langle+\{4\}\rangle\Box p)\vee[+\{2\}]\Diamond q))\\ \wedge\ \mathit{bf}(\{1\mapsto s_1\}[+\{3\}](\langle+\{2\}\rangle\Diamond r\wedge\langle+\{5\}\rangle\Box q))\end{array}\right)$$

$$\equiv \left(\begin{array}{l}\mathit{bf}(\{1\mapsto s_1, 2\mapsto s_2\}([+\{3\}](\langle+\{4\}\rangle\Box p)\vee[+\{2\}]\Diamond q))\\ \wedge\ \mathit{bf}(\{1\mapsto s_1, 2\mapsto s_3, 4\mapsto s_4, 5\mapsto s_5\}(\langle+\{2\}\rangle\Diamond r\wedge\langle+\{5\}\rangle\Box q))\end{array}\right)$$

$$\equiv \left(\begin{array}{l} \left(\begin{array}{l} \quad\{1 \mapsto s_1, 2 \mapsto s_2, 4 \mapsto s_{13}, 5 \mapsto s_7\}\Box p \\ \vee \{1 \mapsto s_1, 3 \mapsto s_8, 4 \mapsto s_9, 5 \mapsto s_{10}\}\Diamond q \end{array} \right) \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_{11}, 4 \mapsto s_4, 5 \mapsto s_5\}\Diamond r \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_3, 4 \mapsto s_4, 5 \mapsto s_{12}\}\Box q \end{array} \right)$$

$$\equiv \left(\begin{array}{l} \quad \left(\begin{array}{l} \quad\{1 \mapsto s_1, 2 \mapsto s_2, 4 \mapsto s_{13}, 5 \mapsto s_7\}\Box p \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_{11}, 4 \mapsto s_4, 5 \mapsto s_5\}\Diamond r \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_3, 4 \mapsto s_4, 5 \mapsto s_{12}\}\Box q \end{array} \right) \\ \vee \left(\begin{array}{l} \quad\{1 \mapsto s_1, 3 \mapsto s_8, 4 \mapsto s_9, 5 \mapsto s_{10}\}\Diamond q \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_{11}, 4 \mapsto s_4, 5 \mapsto s_5\}\Diamond r \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_3, 4 \mapsto s_4, 5 \mapsto s_{12}\}\Box q \end{array} \right) \end{array} \right) \text{; distribution of } \wedge \text{ over } \vee$$

This DNBB-formula sheds some light in analyzing BSIL formulas. As can be seen, the formula is satisfied iff one of the outermost disjuncts is satisfied. Without loss of generality, we examine the first disjunct: $\eta_1 \equiv \left(\begin{array}{l} \quad\{1 \mapsto s_1, 2 \mapsto s_2, 4 \mapsto s_{13}, 5 \mapsto s_7\}\Box p \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_{11}, 4 \mapsto s_4, 5 \mapsto s_5\}\Diamond r \\ \wedge \{1 \mapsto s_1, 2 \mapsto s_3, 4 \mapsto s_4, 5 \mapsto s_{12}\}\Box q \end{array} \right)$
There are the following three C-strategies involved in the satisfaction of the formula.

- Σ_1 for $\{1 \mapsto s_1, 2 \mapsto s_2, 4 \mapsto s_{13}, 5 \mapsto s_7\}$ of $\{1, 2, 4, 5\}$ used to satisfy $\Box p$.
- Σ_2 for $\{1 \mapsto s_1, 2 \mapsto s_{11}, 4 \mapsto s_4, 5 \mapsto s_5\}$ of $\{1, 2, 4, 5\}$ used to satisfy $\Diamond r$.
- Σ_3 for $\{1 \mapsto s_1, 2 \mapsto s_3, 4 \mapsto s_4, 5 \mapsto s_{12}\}$ of $\{1, 2, 4, 5\}$ used to satisfy $\Box q$.

This disjunct says the following interaction restrictions.

- Σ_1, Σ_2, and Σ_3 must agree in their choices at nodes owned by agent 1.
- Σ_2 and Σ_3 must agree in their choices at nodes owned by agent 4.

In the following, we use the observation in this example to construct structures from DNBB-formulas for the model-checking of conjunctive DNBB-formulas. ■

For the convenience of our algorithm presentation, we represent a conjunctive DNBB-formula η as a set of bound literals. Our goal is to design a computation tree exploration procedure that given a set Ψ of bound literals, labels each node in the tree with a subset of Ψ for the set of path formulas that some C-strategies have to enforce without violating the restrictions of strategy interaction imposed in Ψ through the strategy variables. In the design of the procedure, one central component is how to label the children of a node with appropriate subsets of Ψ as inherited path obligations. This is accomplished with the procedure *Suc_set*(A, q, Ψ) in the following. Given a node q in the computation tree and a set Ψ, the procedure nondeterministically returns an assignment of bound literals to children of q to enforce the bound literals in Ψ without violating the strategy interaction of bound literals.

```
Suc_set($A, q, \Psi$) // $\lambda()$ has been extended with satisfied child subformulas at each state.
Let $\Delta$ be $\{(q', \emptyset) \mid (q, q') \in R\}$ and $\Phi$ be $\Psi$.
while $\Phi$ is not empty do
    Pick an element $B\psi \in \Phi$ and let $\Phi$ be $\Phi - \{B\psi\}$.
    if ($\psi$ is either $\phi_1 \mathrm{U} \phi_2$ or $\phi_1 \mathrm{W} \phi_2$) with $\phi_2 \in \lambda(q)$ then continue.
```

```
 5:     else if ($\psi$ is either $\phi_1 \mathrm{U} \phi_2$ or $\phi_1 \mathrm{W} \phi_2$) with $\phi_1 \notin \lambda(q)$ then return $\emptyset$.
 6:     end if
 7:     if $\omega(q) \notin def(B)$ then
 8:        if $\psi$ is not $\bigcirc \phi_1$ then
 9:           for $(q', \Psi') \in \Delta$ do replace $(q', \Psi')$ with $(q', \Psi' \cup \{B\psi\})$ in $\Delta$. end for
10:        else if there is an $(q, q') \in R$ with $\phi_1 \notin \lambda(q')$ then
11:           return $\emptyset$.
12:        end if
13:     else if $\exists B'\psi' \in \Psi - \Phi, \exists (q', \Psi') \in \Delta (B'\psi' \in \Psi' \wedge \omega(q) \in def(B) \cap def(B'))$
        then
14:        if $\psi$ is not $\bigcirc \phi_1$ then replace $(q', \Psi')$ with $(q', \Psi' \cup \{B\psi\})$ in $\Delta$;
15:        else if $\phi_1 \notin \lambda(q')$ then return $\emptyset$. end if
16:     else
17:        Nondeterministically pick a $(q', \Psi') \in \Delta$.
18:        if $\psi$ is not $\bigcirc \phi_1$ then replace $(q', \Psi')$ with $(q', \Psi' \cup \{B\psi\})$ in $\Delta$;
19:        else if $\phi_1 \notin \lambda(q')$ then return $\emptyset$. end if
20:     end if
21:  end while
22:  return $\Delta$.
```

The loop at statement 2 iterates through all the path obligations at the current node and passes them down to the children if necessary. Statements 4 checks if $B\psi$ is fulfilled. Statements 5 checks if $B\psi$ is violated. When a violation happens, the assignment of path obligations to children fails and we return $\emptyset$. The if-statement at line 7 is for nodes where no strategy choice is to be made. Statement 13 is for the case when we have already made a choice for a $B'\psi'$ that should share the same strategy decision with $B\psi$ at q. Thus the choice for $B\psi$ has to be consistent with that for $B'\psi'$. Statement 16 handles the case when there is no such $B'\psi'$.

5.2 Procedures for Checking BSIL Properties

The procedure in the following checks a BSIL state property ϕ at a state q of A.

```
Check_BSIL($A, q, \phi$)
if $\phi$ is $p$ then
   if $\phi \in \lambda(q)$ then return true. else return false. end if
else if $\phi$ is $\phi_1 \vee \phi_2$ then
   return Check_BSIL($A, q, \phi_1$) $\vee$ Check_BSIL($A, q, \phi_2$)
else if $\phi$ is $\neg \phi_1$ then
   return $\neg$Check_BSIL($A, q, \phi_1$)
else if $\phi$ is $\langle M \rangle \theta$ for a tree or path formula $\theta$ then
   return Check_tree($A, q, \langle M \rangle \theta$)
end if
```

The procedure is straightforward and works inductively on the structure of the input formula. For convenience, we need procedure Check_set$(A, Q'\phi_1)$ in the following that checks a BSIL property ϕ_1 at each state in Q'.

```
Check_set$(A, Q', \phi_1)$
if $\phi_1 \notin P_A \cup \{\mathit{true}, \mathit{false}\}$ then
    for each $q' \in Q'$ do
        if Check_BSIL$(A, q', \phi_1)$ then
            Let $\lambda(q')$ be $(\lambda(q') \cup \{\phi_1\}) - \{\neg\phi_1\}$.
        else
            Let $\lambda(q')$ be $(\lambda(q') - \{\phi_1\}) \cup \{\neg\phi_1\}$.
        end if
    end for
end if
```

Then we use procedure Check_tree$(A, q, \langle M\rangle\theta)$ in the following to check if a state q satisfies $\langle M\rangle\psi$.

```
Check_tree$(A, q, \langle M\rangle\theta)$
 1: Rewrite $\mathit{bf}(\emptyset\langle M\rangle\theta)$ to DNBB-formula $\eta_1 \vee \ldots \vee \eta_n$.
 2: for $i \in [1, n]$ do
 3:     Represent $\eta_i$ as a set $\Psi$ of bound literals.
 4:     for each $B\psi$ in $\Psi$. do
 5:         if $\psi$ is $\bigcirc\phi_1$ then
 6:             Check_set$(A, \{q' \mid (q, q') \in R_A\}, \phi_1)$.
 7:         else if $\psi$ is either $\phi_1 \mathrm{U} \phi_2$ or $\phi_1 \mathrm{W} \phi_2$ then
 8:             Check_set$(A, Q_A, \phi_1)$.
 9:             Check_set$(A, Q_A, \phi_2)$.
10:         end if
11:     end for
12:     if Rec_tree$(A, q, \Psi)$ then  return $\mathit{true}$. end if
13: end for
14: return $\mathit{false}$.
```

We first rewrite $\langle M\rangle\theta$ to its DNBB-formula at statement 1 by calling $\mathit{bf}(\emptyset\langle M\rangle\theta)$ and using the distribution law of conjunctions over disjunctions. We then iteratively check with the loop starting from statement 2 if $\langle M\rangle\theta$ is satisfied due to one of its conjunctive DNBB-formula components of $\langle M\rangle\psi$. At statement 3, we construct the set Ψ of bound literals of the component. We evaluate the subformulas with the inner loop starting at statement 4. Finally at statement 12, we explore the computation tree, with procedure Rec_tree(A, q, Ψ) in the following, and pass down the path obligations to the children according to the restrictions of the SV-binding in Ψ.

```
Rec_tree($A, q, \Psi$)
if $(q, \Psi)$ coincides with an ancestor in the exploration then
    if there is no $B\phi_1 \mathrm{U} \phi_2$ in $\Psi$ then return true; else return false. end if
end if
Let Suc_set$(A, q, \Psi)$ be $\Delta$.
if $\Delta = \emptyset$ then return false end if
for each $(q', \Psi') \in \Delta$ with $\Psi' \neq \emptyset$ do
    if Rec_tree$(A, q', \Psi')$ is false then return false. end if
end for
return true.
```

Note that procedure `Rec_tree`(A, q, Ψ) is nondeterministic since it employs *Suc_set*(A, q, Ψ) to nondeterministically calculate an assignment Δ of path obligations to the children of q.

5.3 Correctness Proof of the Algorithm

For the proof of the correctness of the algorithm, we define *strategy interaction trees* (*SI-trees*) in the following. An SI-tree for a set Ψ of bound literals and a game graph $A = \langle m, Q, I, \omega, P, \lambda, R\rangle$ from a state $q \in Q$ is a labeled computation tree $\langle V, r, \alpha, E, \beta\rangle$ with the following restrictions.

- V is the set of nodes in the tree.
- $r \in V$ is the root of the tree.
- $\alpha : V \mapsto Q$ labels each tree node with a state. Also $\alpha(r) = q$.
- $E \subseteq V \times V$ is the set of arcs of the tree such that for each $(q, q') \in R$, there exists an $(v, v') \in E$ with $\alpha(v) = q$ and $\alpha(v') = q'$.
- $\beta : V \mapsto 2^{\Psi}$ labels each node with a subset of Ψ for path formulas in ψ that need to be fulfilled at a node. Moreover, we have the following restrictions on β.
    - $\beta(r) = \Psi$.
    - For every $v \in V$, there exists a $\Delta =$ *Suc_set*$(A, \alpha(v), \beta(v))$ such that for every $(q', \Psi') \in \Delta$, there exists a $(v, v') \in E$ with $\alpha(v') = q'$ and $\beta(v') = \Psi'$.

The SI-tree is *fulfilled* iff for every path $v_0 v_1 \ldots v_k \ldots$ along the tree from the root, there exists an $h \geq 0$ such that for every $j \geq h$, there is no $B\phi_1 \mathrm{U} \phi_2 \in \beta(v_j)$.

We have the following property between an SI-tree and execution of procedure `Rec_tree`(A, q, Ψ) from the root of an SI-tree.

Lemma 11. *For a set Ψ of bound literals,* `Rec_tree`(A, q, Ψ) *returns true iff there exists a fulfilled SI-tree for Ψ and A from q.* ■

Lemma 12. *Given a conjunctive DNBB-formula η represented as a set Ψ of bound literals, there exists a function π on strategy variables in η with $A, q \models^{\pi} \eta$ iff there exists a fulfilled SI-tree for A and Ψ from q.* ■

The correctness of procecure `Rec_tree`(A, q, Ψ) then directly follows from lemmas 11 and 12. Then with a structure induction on a given BSIL formula and the correctness of procedure `Rec_tree`(A, q, Ψ), we have the following lemma for the correctness of procedure `Check_BSIL`(A, q, ϕ).

Lemma 13. *Given a game graph A, a state q in A, and a BSIL formula ϕ,* $\texttt{Check\_BSIL}(A, q, \phi)$ *iff* $A, q \models_\perp \phi$. ■

5.4 Complexities of the Algorithm

The algorithm that we presented in subsections 5.1 and 5.2 can run in PSPACE mainly because we can implement procedure $\texttt{Rec\_tree}(A, q, \Psi)$ with a stack of polynomial height. To see this, please be reminded that we use procedure *Suc_set*(A, q, Ψ) to calculate the assignment of bound literals to the children to q in the computation tree. Specifically, procedure *Suc_set*(A, q, Ψ) nondeterministically returns a set Δ with elements of the form (q', Ψ') such that $(q, q') \in R$ and $\Psi' \subseteq \Psi$. Thus along any path in the SI-tree, the sets of literal bounds never increase. Moreover, when there is a node in the exploration of SI-tree that coincides with an ancestor, we backtrack in the exploration. This implies that along any path segment longer than $|Q|$, either one of the following two conditions hold.

- A backtracking happens at the end of the segment.
- The sets of bound literals along the segment must decrease in size at least once.

These conditions lead to the observation that with procedure *Suc_set*(A, q, Ψ), the recursive exploration of a path can grow no longer than $1 + |\Psi| \cdot |Q|$. This leads to the following lemma.

Lemma 14. *The BSIL model-checking algorithm in subsections 5.1 and 5.2 is in PSPACE.* ■

Following lemmas 9 and 14, we then get the following lemma.

Lemma 15. *The model-checking problem of a BSIL formula against a turn-based game graph is PSPACE-complete.* ■

A rough analysis of the time complexity of our algorithm follows. Let $|\phi|$ be the length of a BSIL formula ϕ. At each call to *Suc_set*(), the size of Ψ is at most $|\phi|$. The number of root-to-leaf paths in an SI-tree is at most $|\psi|$ since we only have to pass down $|\psi|$ bound literals. We can use the positions of the common ancestors of the leaves of such paths to analyze the number of the configurations of such SI-trees. The common ancestors can happen anywhere along the root-to-leaf paths. Thus, there are $(1 + |\phi| \cdot |Q|)^{|\phi|}$ ways to arrange the positions of the common ancestors since the length of paths are at most $1 + |\phi| \cdot |Q|$. The number of ways that the bound literals can be assigned to the leaves is at most $|\phi|^{|\phi|}$. The number of state labeling of the nodes on the paths is at most $|Q|^{|\phi| \cdot (1+|\phi| \cdot |Q|)}$. Thus, given a Ψ, the total number of different SI-trees is $O(|Q|^{|\phi| \cdot (1+|\phi| \cdot |Q|)} |\phi|^{|\phi|} (1 + |\phi| \cdot |Q|)^{|\phi|}) = O(|Q|^{|\phi| \cdot (2+|\phi| \cdot |Q|)} |\phi|^{2|\phi|})$. There are $O(2^{|\phi|})$ different possible values of Ψ. There are at most $|\phi|$ SI-trees to construct for the model-checking task. Thus the total time complexity of our algorithm is $O(|\phi| 2^{|\phi|} |Q|^{|\phi| \cdot (2+|\phi| \cdot |Q|)} |\phi|^{2|\phi|})$.

6 Strategy Interaction Logic

In a BSIL formula, the topmost SIQ inside a path modal formula must not be a pure SQ. Lifting this restriction, we get *SIL* (*Strategy Interaction Logic*). For a game of m agents, an SIL formula ϕ has the following syntax.

$$\begin{aligned}\phi ::= p \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle M\rangle\phi_1 \mid \langle M\rangle \bigcirc \phi_1 \mid \langle M\rangle\phi_1 \mathrm{U}\phi_2 \mid \langle M\rangle\phi_1\mathrm{W}\phi_2 \\ \mid \langle +M\rangle\phi_1 \mid \langle +M\rangle \bigcirc \phi \mid \langle +M\rangle\phi_1\mathrm{U}\phi_2 \mid \langle +M\rangle\phi_1\mathrm{W}\phi_2\end{aligned}$$

The shorthands and semantics of SIL are exactly as those of BSIL. SIL is more expressive than ATL* for turn-based games. The reason is that SIQ "$\langle +\emptyset\rangle$" can be used to inherit strategies from parent modal operators to child ones. For example, $\langle\{1\}\rangle\Box\Diamond p$ is equivalent to $\langle\{1\}\rangle\Box\langle+\emptyset\rangle\Diamond p$.

Lemma 16. *SIL is strictly more expressive than ATL* for turn-based games.* ■

One implication of lemma 16 is that the model-checking problem of SIL is at least as hard as that of ATL*, which is doubly exponential time complete.

7 Conclusion

BSIL can be useful in describing close interaction among strategies of agents in a multi-agent system. Although it can express properties that ATL*, GL, and AMC cannot, its model-checking problem incurs a much lower complexity. Future work in this direction may include further extension to BSIL.

Acknowledgment. The authors would like to thank Professor Moshe Vardi and the anonymous reviewers of CONCUR 2011 for their valuable comments and suggestions.

References

1. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. Journal of the ACM (JACM) 49(5), 672–713 (2002)
2. Baier, C., Brázdil, T., Gröser, M., Kucera, A.: Stochastic game logic. In: QEST, pp. 227–236. IEEE Computer Society, Los Alamitos (2007)
3. Chatterjee, K., Henzinger, T.A., Piterman, N.: Strategy logic. Information and Computation 208, 677–693 (2010)
4. Costa, A.D., Laroussinie, F., Markey, N.: Atl with strategy contexts: Expressiveness and model checking. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, pp. 120–132. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Company, New York (1979)
6. Mogavero, F., Murano, A., Vardi, M.Y.: Reasoning about strategies. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010). Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, pp. 133–144. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)

The Complexity of Nash Equilibria in Limit-Average Games*

Michael Ummels[1] and Dominik Wojtczak[2,3]

[1] LSV, CNRS & ENS Cachan, France
ummels@lsv.ens-cachan.fr
[2] University of Liverpool, UK
d.k.wojtczak@liv.ac.uk
[3] Oxford University Computing Laboratory, UK

Abstract. We study the computational complexity of Nash equilibria in concurrent games with limit-average objectives. In particular, we prove that the existence of a Nash equilibrium in randomised strategies is undecidable, while the existence of a Nash equilibrium in pure strategies is decidable, even if we put a constraint on the payoff of the equilibrium. Our undecidability result holds even for a restricted class of concurrent games, where nonzero rewards occur only on terminal states. Moreover, we show that the constrained existence problem is undecidable not only for concurrent games but for turn-based games with the same restriction on rewards. Finally, we prove that the constrained existence problem for Nash equilibria in (pure or randomised) stationary strategies is decidable and analyse its complexity.

1 Introduction

Concurrent games provide a versatile model for the interaction of several components in a distributed system, where the components perform actions in parallel [17]. Classically, such a system is modelled by a family of concurrent two-player games, one for each component, where one component tries to fulfil its specification against the coalition of all other components. In practice, this modelling is often too pessimistic because it ignores the specifications of the other components. We argue that a distributed system is more faithfully modelled by a multiplayer game where each player has her own objective, which is independent of the other players' objectives.

Another objection to the classical theory of verification and synthesis has been that specifications are *qualitative*: either the specification is fulfilled, or it is violated. Examples of such specifications include reachability properties, where a certain set of target states has to be reached, or safety properties, where a certain set of states has to be avoided. In practice, many specifications are of

* This work was supported by ESF RNP "Games for Design and Verification" (GAMES), the French project ANR-06-SETI-003 (DOTS) and EPSRC grant EP/G050112/1.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 482–496, 2011.

a *quantitative* nature, examples of which include minimising average power consumption or maximising average throughput. Specifications of the latter kind can be expressed by assigning (positive or negative) rewards to states or transitions and considering the *limit-average* reward gained from an infinite play. In fact, concurrent games where a player's payoff is defined in such a way have been a central topic in game theory (see the related work section below).

The most common solution concept for games with multiple players is that of a Nash equilibrium [20]. In a Nash equilibrium, no player can improve her payoff by changing her strategy unilaterally. Unfortunately, Nash equilibria do not always exist in concurrent games, and if they exist, they may not be unique. In applications, one might look for an equilibrium where some players receive a high payoff while other players receive a low payoff. Formulated as a decision problem, given a game with k players and thresholds $\overline{x}, \overline{y} \in (\mathbb{Q} \cup \{\pm\infty\})^k$, we want to know whether the game has a Nash equilibrium whose payoff lies in-between $\overline{x}$ and $\overline{y}$; we call this decision problem NE.

The problem NE comes in several variants, depending on the type of strategies one considers: On the one hand, strategies may be *randomised* (allowing randomisation over actions) or *pure* (not allowing such randomisation). On the other hand, one can restrict to *stationary* strategies, which only depend on the last state. Indeed, we show that these restrictions give rise to distinct decision problems, which have to be analysed separately.

Our results show that the complexity of NE highly depends on the type of strategies that realise the equilibrium. In particular, we prove the following results, which yield an almost complete picture of the complexity of NE:

1. NE for pure stationary strategies is NP-complete.
2. NE for stationary strategies is decidable in PSPACE, but hard for both NP and SqrtSum.
3. NE for arbitrary pure strategies is NP-complete.
4. NE for arbitrary randomised strategies is undecidable and, in fact, not recursively enumerable.

All of our lower bounds for NE and, in particular, our undecidability result hold already for a subclass of concurrent games where Nash equilibria are guaranteed to exist, namely for *turn-based* games. If this assumption is relaxed and Nash equilibria are not guaranteed to exist, we prove that even the plain existence problem for Nash equilibria is undecidable. Moreover, many of our lower bounds hold already for games where non-zero rewards only occur on terminal states, and thus also for games where each player wants to maximise the *total sum* of the rewards.

As a byproduct of our decidability proof for pure strategies, we give a polynomial-time algorithm for deciding whether in a multi-weighted graph there exists a path whose limit-average weight vector lies between two given thresholds, a result that is of independent interest. For instance, our algorithm can be used for deciding the emptiness of a *multi-threshold mean-payoff language* [2].

Due to space constraints, most proofs are either only sketched or omitted entirely. For the complete proofs, see [27].

Related Work. Concurrent and, more generally, stochastic games go back to [23], who proved the existence of the *value* for *discounted two-player zero-sum* games. This result was later generalised by [13] who proved that every discounted game has a Nash equilibrium. [16] introduced limit-average objectives, and [19] proved the existence of the value for stochastic two-player zero-sum games with limit-average objectives. Unfortunately, as demonstrated by [12], these games do, in general, not admit a Nash equilibrium (see Example 1). However, [29, 30] proved that, for all $\varepsilon > 0$, every two-player stochastic limit-average game admits an *ε-equilibrium*, i.e. a pair of strategies where each player can gain at most ε from switching her strategy. Whether such equilibria always exist in games with more than two players is an important open question [21].

Determining the complexity of Nash equilibria has attracted much interest in recent years. In particular, a series of papers culminated in the result that computing a Nash equilibrium of a finite two-player game in *strategic form* is complete for the complexity class PPAD [6, 8]. The constrained existence problem, where one looks for a Nash equilibrium with certain properties, has also been investigated for games in strategic form. In particular, [7] showed that deciding whether there exists a Nash equilibrium where player 0's payoff exceeds a given threshold and related decision problems are NP-complete for two-player games in strategic form.

For concurrent games with limit-average objectives, most algorithmic results concern two-player zero-sum games. In the turn-based case, these games are commonly known as *mean-payoff games* [10, 32]. While it is known that the value of such a game can be computed in pseudo-polynomial time, it is still open whether there exists a polynomial-time algorithm for solving mean-payoff games. A related model are *multi-dimensional mean-payoff games* where one player tries to maximise several mean-payoff conditions at the same time [5]. In particular, [28] showed that the value problem for these games is coNP-complete.

One subclass of concurrent games with limit-average objectives that has been studied in the multiplayer setting are concurrent games with reachability objectives. In particular, [3] showed that the constrained existence problem for Nash equilibria is NP-complete for these games (see also [25, 14]). We extend their result to limit-average objectives. However, we assume that strategies can observe actions (a common assumption in game theory), which they do not. Hence, while our result is more general w.r.t. the type of objectives we consider, their result is more general w.r.t. the type of strategies they allow.

In a recent paper [26], we studied the complexity of Nash equilibria in *stochastic* games with reachability objectives. In particular, we proved that NE for pure strategies is undecidable in this setting. Since we prove here that this problem is decidable in the non-stochastic setting, this undecidability result can be explained by the presence of probabilistic transitions in stochastic games. On the other hand, we prove in this paper that randomisation in strategies also leads to undecidability, a question that was left open in [26].

2 Concurrent Games

Concurrent games are played by finitely many players on a finite state space. Formally, a concurrent game is given by

- a finite nonempty set Π of *players*, e.g. $\Pi = \{0, 1, \ldots, k-1\}$,
- a finite nonempty set S of *states*,
- for each player i and each state s a nonempty set $\Gamma_i(s)$ of *actions* taken from a finite set Γ,
- a *transition function* $\delta \colon S \times \Gamma^{\Pi} \to S$,
- for each player $i \in \Pi$ a *reward function* $r_i \colon S \to \mathbb{R}$.

For computational purposes, we assume that all rewards are rational numbers with numerator and denominator given in binary. We say that an action profile $\overline{a} = (a_i)_{i\in\Pi}$ is *legal* at state s if $a_i \in \Gamma_i(s)$ for each $i \in \Pi$. Finally, we call a state s *controlled* by player i if $|\Gamma_j(s)| = 1$ for all $j \neq i$, and we say that a game is *turn-based* if each state is controlled by (at least) one player. For turn-based games, an action of the controlling player prescribes to go to a certain state. Hence, we will usually omit actions in turn-based games.

For a tuple $\overline{x} = (x_i)_{i\in\Pi}$, where the elements x_i belong to an arbitrary set X, and an element $x \in X$, we denote by $\overline{x}_{-i}$ the restriction of $\overline{x}$ to $\Pi \setminus \{i\}$ and by $(\overline{x}_{-i}, x)$ the unique tuple $\overline{y} \in X^{\Pi}$ with $y_i = x$ and $\overline{y}_{-i} = \overline{x}_{-i}$.

A play of a game $\mathcal{G}$ is an infinite sequence $s_0\overline{a}_0s_1\overline{a}_1 \ldots \in (S \cdot \Gamma^{\Pi})^{\omega}$ such that $\delta(s_j, \overline{a}_j) = s_{j+1}$ for all $j \in \mathbb{N}$. For each player, a play $\pi = s_0\overline{a}_0s_1\overline{a}_1 \ldots$ gives rise to an infinite sequence of rewards. There are different criteria to evaluate this sequence and map it to a *payoff*. In this paper, we consider the *limit-average* (or *mean-payoff*) criterion, where the payoff of π for player i is defined by

$$\phi_i(\pi) := \liminf_{n\to\infty} \frac{1}{n} \sum_{j=0}^{n-1} r_i(s_j).$$

Note that this payoff mapping is *prefix-independent*, i.e. $\phi_i(\pi) = \phi_i(\pi')$ if π' is a suffix of π. An important special case are games where non-zero rewards occur only on *terminal* states, i.e. states s with $\delta(s, \overline{a}) = s$ for all (legal) $\overline{a} \in \Gamma^{\Pi}$. These games were introduced by [12] under the name *recursive games*, but we prefer to call them *terminal-reward games*. Hence, in a terminal-reward game, $\phi_i(\pi) = r_i(s)$ if π enters a terminal state s and $\phi_i(\pi) = 0$ otherwise.

Often, it is convenient to designate an *initial* state. An *initialised* game is thus a tuple $(\mathcal{G}, s_0)$ where $\mathcal{G}$ is a concurrent game and s_0 is one of its states.

Strategies and Strategy Profiles. For a finite set X, we denote by $\mathcal{D}(X)$ the set of probability distributions over X. A *(randomised) strategy* for player i in $\mathcal{G}$ is a mapping $\sigma \colon (S \cdot \Gamma^{\Pi})^{*} \cdot S \to \mathcal{D}(\Gamma)$ assigning to each possible *history* $xs \in (S \cdot \Gamma^{\Pi})^{*} \cdot S$ a probability distribution $\sigma(xs)$ over actions such that $\sigma(xs)(a) > 0$ only if $a \in \Gamma_i(s)$. We write $\sigma(a \mid xs)$ for the probability assigned to $a \in \Gamma$ by the distribution $\sigma(xs)$. A *(randomised) strategy profile of* $\mathcal{G}$ is a tuple $\overline{\sigma} = (\sigma_i)_{i\in\Pi}$ of strategies in $\mathcal{G}$, one for each player.

A strategy σ for player i is called *pure* if for each $xs \in (S \cdot \Gamma^{\Pi})^* \cdot S$ the distribution $\sigma(xs)$ is *degenerate*, i.e. there exists $a \in \Gamma_i(s)$ with $\sigma(a \mid xs) = 1$. Note that a pure strategy can be identified with a function $\sigma\colon (S \cdot \Gamma^{\Pi})^* \cdot S \to \Gamma$. A strategy profile $\overline{\sigma} = (\sigma_i)_{i \in \Pi}$ is called *pure* if each σ_i is pure, in which case we can identify $\overline{\sigma}$ with a mapping $(S \cdot \Gamma^{\Pi})^* \cdot S \to \Gamma^{\Pi}$. Note that, given an initial state s_0 and a pure strategy profile $\overline{\sigma}$, there exists a unique play $\pi = s_0 \overline{a}_0 s_1 \overline{a}_1 \dots$ such that $\overline{\sigma}(s_0 \overline{a}_0 \dots \overline{a}_{j-1} s_j) = \overline{a}_j$ for all $j \in \mathbb{N}$; we call π the play *induced* by $\overline{\sigma}$ from s_0.

A strategy σ is called *stationary* if σ depends only on the last state: $\sigma(xs) = \sigma(s)$ for all $xs \in (S \cdot \Gamma^{\Pi})^* \cdot S$. A strategy profile $\overline{\sigma} = (\sigma_i)_{i \in \Pi}$ is called *stationary* if each σ_i is stationary. Finally, we call a strategy (profile) *positional* if it is both pure and stationary.

The Probability Measure Induced by a Strategy Profile. Given an initial state $s_0 \in S$ and a strategy profile $\overline{\sigma} = (\sigma_i)_{i \in \Pi}$, the *conditional probability* of $\overline{a} \in \Gamma^{\Pi}$ given the history $xs \in (S \cdot \Gamma^{\Pi})^* \cdot S$ is $\overline{\sigma}(\overline{a} \mid xs) := \prod_{i \in \Pi} \sigma_i(a_i \mid xs)$. The probabilities $\overline{\sigma}(\overline{a} \mid xs)$ induce a probability measure on the Borel σ-algebra over $(S \cdot \Gamma^{\Pi})^{\omega}$ as follows: The probability of a basic open set $s_1 \overline{a}_1 \dots s_n \overline{a}_n \cdot (S \cdot \Gamma^{\Pi})^{\omega}$ equals the product $\prod_{j=1}^{n} \overline{\sigma}(\overline{a}_j \mid s_1 \overline{a}_1 \dots \overline{a}_{j-1} s_j)$ if $s_1 = s_0$ and $\delta(s_j, \overline{a}_j) = s_{j+1}$ for all $1 \leq j < n$; in all other cases, this probability is 0. By *Carathéodory's extension theorem*, this extends to a unique probability measure assigning a probability to every Borel subset of $(S \cdot \Gamma^{\Pi})^{\omega}$, which we denote by $\mathrm{Pr}^{\overline{\sigma}}_{s_0}$. Via the natural projection $(S \cdot \Gamma^{\Pi})^{\omega} \to S^{\omega}$, we obtain a probability measure on the Borel σ-algebra over S^{ω}. We abuse notation and denote this measure also by $\mathrm{Pr}^{\overline{\sigma}}_{s_0}$; it should always be clear from the context to which measure we are referring to. Finally, we denote by $\mathrm{E}^{\overline{\sigma}}_{s_0}$ the expectation operator that corresponds to $\mathrm{Pr}^{\overline{\sigma}}_{s_0}$, i.e. $\mathrm{E}^{\overline{\sigma}}_{s_0}(f) = \int f \, \mathrm{d}\mathrm{Pr}^{\overline{\sigma}}_{s_0}$ for all Borel measurable functions $f\colon (S \cdot \Gamma^{\Pi})^{\omega} \to \mathbb{R} \cup \{\pm\infty\}$ or $f\colon S^{\omega} \to \mathbb{R} \cup \{\pm\infty\}$. In particular, we are interested in the quantities $p_i := \mathrm{E}^{\overline{\sigma}}_{s_0}(\phi_i)$. We call p_i the *(expected) payoff* of $\overline{\sigma}$ for player i and the vector $(p_i)_{i \in \Pi}$ the (expected) payoff of $\overline{\sigma}$.

Drawing Concurrent Games. When drawing a concurrent game as a graph, we will adhere to the following conventions: States are usually depicted as circles, but terminal states are depicted as squares. The initial state is marked by a dangling incoming edge. An edge from s to t with label $\overline{a}$ means that $\delta(s, \overline{a}) = t$ and that $\overline{a}$ is legal at s. However, the label $\overline{a}$ might be omitted if it is not essential. In turn-based games, the player who controls a state is indicated by the label next to it. Finally, a label of the form $i\colon x$ next to state s indicates that $r_i(s) = x$; if this reward is 0, the label will usually be omitted.

3 Nash Equilibria

To capture rational behaviour of selfish players, Nash [20] introduced the notion of — what is now called — a *Nash equilibrium*. Formally, given a game $\mathcal{G}$ and an

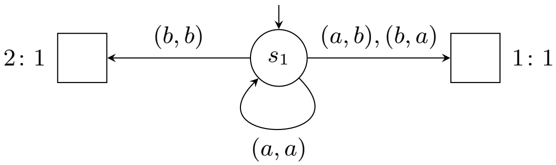


Fig. 1. A terminal-reward game that has no Nash equilibrium

initial state s_0, a strategy τ for player i is a *best response* to a strategy profile $\overline{\sigma}$ if τ maximises the expected payoff for player i, i.e.

$$\mathrm{E}_{s_0}^{\overline{\sigma}_{-i},\tau'}(\phi_i) \leq \mathrm{E}_{s_0}^{\overline{\sigma}_{-i},\tau}(\phi_i)$$

for all strategies τ' for player i. A strategy profile $\overline{\sigma} = (\sigma_i)_{i \in \Pi}$ is a *Nash equilibrium* of $(\mathcal{G}, s_0)$ if for each player i the strategy σ_i is a best response to $\overline{\sigma}$. Hence, in a Nash equilibrium no player can improve her payoff by (unilaterally) switching to a different strategy. As the following example demonstrates, Nash equilibria are not guaranteed to exist in concurrent games.

Example 1. Consider the terminal-reward game $\mathcal{G}_1$ depicted in Fig. 1, which is a variant of the game *hide-or-run* as presented by [9]. We claim that $(\mathcal{G}_1, s_1)$ does not have a Nash equilibrium. First note that, for each $\varepsilon > 0$, player 1 can ensure a payoff of $1 - \varepsilon$ by the stationary strategy that selects action b with probability ε. Hence, every Nash equilibrium (σ, τ) of $(\mathcal{G}_1, s_1)$ must have payoff $(1, 0)$. Now consider the least k such that $p := \sigma(b \mid (s_1(a,a))^k s_1) > 0$. By choosing action b with probability 1 for the history $(s_1(a,a))^k s_1$ and choosing action a with probability 1 for all other histories, player 2 can ensure payoff p, a contradiction to (σ, τ) being a Nash equilibrium.

It follows from Nash's theorem [20] that every game whose arena is a tree (or a DAG) has a Nash equilibrium. Another important special case of concurrent limit-average games where Nash equilibria always exist are turn-based games. For these games, [24] proved not only the existence of arbitrary Nash equilibria but of pure finite-state ones.

To measure the complexity of Nash equilibria in concurrent games, we introduce the following decision problem, which we call NE:

> Given a game $\mathcal{G}$, a state s_0 and thresholds $\overline{x}, \overline{y} \in (\mathbb{Q} \cup \{\pm\infty\})^{\Pi}$, decide whether $(\mathcal{G}, s_0)$ has a Nash equilibrium with payoff $\geq \overline{x}$ and $\leq \overline{y}$.

Note that we have not put any restriction on the type of strategies that realise the equilibrium. It is natural to restrict the search space to profiles of pure, stationary or positional strategies. These restrictions give rise to different decision problems, which we call PureNE, StatNE and PosNE, respectively.

Before we analyse the complexity of these problems, let us convince ourselves that these problems are not just different faces of the same coin. We first show that the decision problems where we look for equilibria in randomised strategies are distinct from the ones where we look for equilibria in pure strategies.

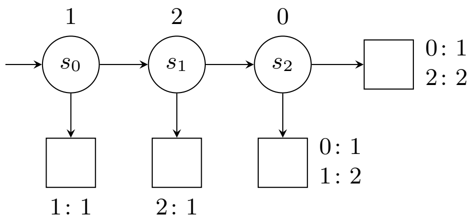


Fig. 2. A game with no pure Nash equilibrium where player 0 wins with positive probability

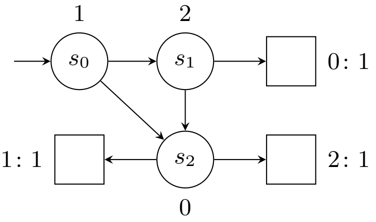


Fig. 3. A game with no stationary Nash equilibrium where player 0 wins with positive probability

Proposition 2. *There exists a turn-based terminal-reward game that has a stationary Nash equilibrium where player* 0 *receives payoff* 1 *but that has no pure Nash equilibrium where player* 0 *receives payoff* > 0*.*

Proof. Consider the game depicted in Fig. 2 played by three players 0, 1 and 2. Clearly, the stationary strategy profile where from state s_2 player 0 selects both outgoing transitions with probability $\frac{1}{2}$ each, player 1 plays from s_0 to s_1 and player 2 plays from s_1 to s_2 is a Nash equilibrium where player 0 receives payoff 1. However, in any pure strategy profile where player 0 receives payoff > 0, either player 1 or player 2 receives payoff 0 and could improve her payoff by switching her strategy at s_0 or s_1, respectively. □

Now we show that it makes a difference whether we look for an equilibrium in stationary strategies or not.

Proposition 3. *There exists a turn-based terminal-reward game that has a pure Nash equilibrium where player* 0 *receives payoff* 1 *but that has no stationary Nash equilibrium where player* 0 *receives payoff* > 0*.*

Proof. Consider the game $\mathcal{G}$ depicted in Fig. 3 and played by three players 0, 1 and 2. Clearly, the pure strategy profile that leads to the terminal state with payoff 1 for player 0 and where player 0 plays "right" if player 1 has deviated and "left" if player 2 has deviated is a Nash equilibrium of $(\mathcal{G}, s_0)$ with payoff 1 for player 0. Now consider any stationary equilibrium of $(\mathcal{G}, s_0)$ where player 0 receives payoff > 0. If the stationary strategy of player 0 prescribes to play "right" with positive probability, then player 2 can improve her payoff by playing to s_2 with probability 1, and otherwise player 1 can improve her payoff by playing to s_2 with probability 1, a contradiction. □

It follows from Proposition 2 that NE and StatNE are different from PureNE and PosNE, and it follows from Proposition 3 that NE and PureNE are different from StatNE and PosNE. Hence, all of these decision problems are pairwise distinct, and their decidability and complexity has to be studied separately.

4 Positional Strategies

In this section, we show that the problem PosNE is NP-complete. Since we can check in polynomial time whether a positional strategy profile is a Nash equilibrium (using a result of [18]), membership in NP is straightforward.

Theorem 4. *PosNE is in NP.*

A result by [5], Lemma 15 implies that PosNE is NP-hard, even for turn-based games with rewards taken from $\{-1, 0, 1\}$ (but the number of players is unbounded). We strengthen their result by showing that the problem remains NP-hard if there are only three players and rewards are taken from $\{0, 1\}$.

Theorem 5. *PosNE is NP-hard, even for turn-based three-player games with rewards* 0 *and* 1*.*

Proof. We reduce from the Hamiltonian cycle problem. Given a graph $G = (V, E)$, we define a turn-based three-player game $\mathcal{G}$ as follows: the set of states is V, all states are controlled by player 0, and the transition function corresponds to E (i.e. $\Gamma_0(v) = vE$ and $\delta(v, \overline{a}) = w$ if and only if $a_0 = w$). Let $n = |V|$ and $v_0 \in V$. The reward of state v_0 to player 1 equals 1. All other rewards for player 0 and player 1 equal 0. Finally, player 2 receives reward 0 at v_0 and reward 1 at all other states. We claim that there is a Hamiltonian cycle in G if and only if $(\mathcal{G}, v_0)$ has a positional Nash equilibrium with payoff $\geq (0, 1/n, (n-1)/n)$. □

By combining our reduction with a game that has no positional Nash equilibrium, we can prove the following stronger result for non-turn-based games.

Corollary 6. *Deciding the existence of a positional Nash equilibrium in a concurrent limit-average game is NP-complete, even for three-player games.*

5 Stationary Strategies

To prove the decidability of StatNE, we appeal to results established for the *existential theory of the reals*, the set of all existential first-order sentences that hold in the ordered field $\mathfrak{R} := (\mathbb{R}, +, \cdot, 0, 1, \leq)$. The best known upper bound for the complexity of the associated decision problem is PSPACE [4], which leads to the following theorem.

Theorem 7. *StatNE is in* PSPACE*.*

The next theorem shows that StatNE is NP-hard, even for turn-based games with rewards 0 and 1. Note that this does not follow from the NP-hardness of PosNE, but requires a different proof.

Theorem 8. *StatNE is NP-hard, even for turn-based games with rewards* 0 *and* 1*.*

Proof. We employ a reduction from SAT, which resembles a reduction in [25]. Given a Boolean formula $\varphi = C_1 \wedge \cdots \wedge C_m$ in conjunctive normal form over propositional variables $X_1, \ldots, X_n$, where w.l.o.g. $m \geq 1$ and each clause is nonempty, we build a turn-based game $\mathcal{G}$ played by players $0, 1, \ldots, n$ as follows: The game $\mathcal{G}$ has states $C_1, \ldots, C_m$ controlled by player 0 and for each clause C and each literal L that occurs in C a state (C, L), controlled by player i if $L = X_i$ or $L = \neg X_i$; additionally, the game contains a terminal state $\perp$. There are transitions from a clause C_j to each state (C_j, L) such that L occurs in C_j and from there to $C_{(j \bmod m)+1}$, and there is a transition from each state of the form $(C, \neg X)$ to $\perp$. Each state except $\perp$ has reward 1 for player 0, whereas $\perp$ has reward 0 for player 0. For player i, each state except states of the form (C, X_i) has reward 1; states of the form (C, X_i) have reward 0. The structure of $\mathcal{G}$ is depicted in Fig. 4. Clearly, $\mathcal{G}$ can be constructed from φ in polynomial time. We claim that φ is satisfiable if and only if $(\mathcal{G}, C_1)$ has a stationary Nash equilibrium with payoff ≥ 1 for player 0. □

By combining our reduction with the game from Example 1, which has no Nash equilibrium, we can prove the following stronger result for concurrent games.

Corollary 9. *Deciding the existence of a stationary Nash equilibrium in a concurrent limit-average game with rewards* 0 *and* 1 *is NP-hard.*

So far we have shown that StatNE is contained in PSPACE and hard for NP, leaving a considerable gap between the two bounds. In order to gain a better understanding of StatNE, we relate this problem to the *square root sum problem* (SqrtSum), an important problem about numerical computations. Formally, SqrtSum is the following decision problem: Given numbers $d_1, \ldots, d_n, k \in \mathbb{N}$, decide whether $\sum_{i=1}^{n} \sqrt{d_i} \geq k$. Recently, [1] showed that SqrtSum belongs to the fourth level of the *counting hierarchy*, a slight improvement over the previously known PSPACE upper bound. However, it has been an open question since

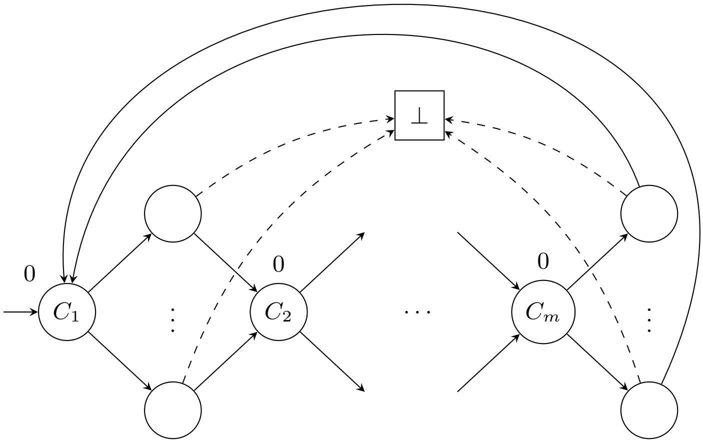


Fig. 4. Reducing SAT to StatNE

the 1970s as to whether SqrtSum falls into the polynomial hierarchy [15, 11]. We give a polynomial-time reduction from SqrtSum to StatNE for turn-based terminal-reward games. Hence, StatNE is at least as hard as SqrtSum, and showing that StatNE resides inside the polynomial hierarchy would imply a major breakthrough in understanding the complexity of numerical computations. While our reduction is similar to the one in [26], it requires new techniques to simulate stochastic states.

Theorem 10. *SqrtSum is polynomial-time reducible to StatNE for turn-based 8-player terminal-reward games.*

Again, we can combine our reduction with the game from Example 1 to prove a stronger result for games that are not turn-based.

Corollary 11. *Deciding whether a concurrent 8-player terminal reward game has a stationary Nash equilibrium is hard for SqrtSum.*

Remark 12. By appealing to results on Markov decision processes with limit-average objectives (see e.g. [22]), the positive results of Sects. 4 and 5 can be extended to stochastic games (with the same complexity bounds).

6 Pure Strategies

In this section, we show that PureNE is decidable and, in fact, NP-complete. Let $\mathcal{G}$ be a concurrent game, $s \in S$ and $i \in \Pi$. We define

$$\mathrm{pval}_i^{\mathcal{G}}(s) = \inf_{\overline{\sigma}} \sup_{\tau} \mathrm{E}_s^{\overline{\sigma}_{-i},\tau}(\phi_i),$$

where $\overline{\sigma}$ ranges over all *pure* strategy profiles of $\mathcal{G}$ and τ ranges over all strategies of player i. Intuitively, $\mathrm{pval}_i^{\mathcal{G}}(s)$ is the lowest payoff that the coalition $\Pi \setminus \{i\}$ can inflict on player i by playing a pure strategy.

By a reduction to a turn-based two-player zero-sum game, we can show that there is a positional strategy profile that attains this value.

Proposition 13. *Let $\mathcal{G}$ be a concurrent game, and $i \in \Pi$. There exists a positional strategy profile $\overline{\sigma}^*$ such that $\mathrm{E}_s^{\overline{\sigma}^*_{-i},\tau}(\phi_i) \leq \mathrm{pval}_i^{\mathcal{G}}(s)$ for all states s and all strategies τ of player i.*

Given a payoff vector $\overline{z} \in (\mathbb{R} \cup \{\pm\infty\})^{\Pi}$, we define a directed graph $G(\overline{z}) = (V, E)$ (with self-loops) as follows: $V = S$, and there is an edge from s to t if and only if there is an action profile $\overline{a}$ with $\delta(s, \overline{a}) = t$ such that (1) $\overline{a}$ is legal at s and (2) $\mathrm{pval}_i^{\mathcal{G}}(\delta(s, (\overline{a}_{-i}, b))) \leq z_i$ for each player i and each action $b \in \Gamma_i(s)$. Following [3], we call any $\overline{a}$ that fulfils (1) and (2) *$\overline{z}$-secure* at s.

Lemma 14. *Let $\overline{z} \in (\mathbb{R} \cup \{\pm\infty\})^{\Pi}$. If there exists an infinite path π in $G(\overline{z})$ from s_0 with $z_i \leq \phi_i(\pi)$ for each player i, then $(\mathcal{G}, s_0)$ has a pure Nash equilibrium with payoff $\phi_i(\pi)$ for player i.*

Proof. Let $\pi = s_0 s_1 \dots$ be an infinite path in $G(\overline{z})$ from s_0 with $z_i \leq \phi_i(\pi)$ for each player i. We define a pure strategy profile $\overline{\sigma}$ as follows: For histories of the form $x = s_0 \overline{a}_0 s_1 \dots s_{k-1} \overline{a}_{k-1} s_k$, we set $\overline{\sigma}(x)$ to an action profile $\overline{a}$ with $\delta(s_k, \overline{a}) = s_{k+1}$ that is $\overline{z}$-secure at s_k. For all other histories $x = t_0 \overline{a}_0 t_1 \dots t_{k-1} \overline{a}_{k-1} t_k$, consider the least j such that $s_{j+1} \neq t_{j+1}$. If $\overline{a}_j$ differs from a $\overline{z}$-secure action profile $\overline{a}$ at s_j in precisely one entry i, we set $\overline{\sigma}(x) = \overline{\sigma}^*(t_k)$, where $\overline{\sigma}^*$ is a (fixed) positional strategy profile such that $\mathrm{E}_s^{\overline{\sigma}^*_{-i},\tau}(\phi_i) \leq \mathrm{pval}_i^{\mathcal{G}}(s)$ for all $s \in S$ (which is guaranteed to exist by Proposition 13); otherwise, $\overline{\sigma}(x)$ can be chosen arbitrarily. It is easy to see that $\overline{\sigma}$ is a Nash equilibrium with induced play π. □

Lemma 15. *Let $\overline{\sigma}$ be a pure Nash equilibrium of $(\mathcal{G}, s_0)$ with payoff $\overline{z}$. Then there exists an infinite path π in $G(\overline{z})$ from s_0 with $\phi_i(\pi) = z_i$ for each player i.*

Proof. Let $s_0 \overline{a}_0 s_1 \overline{a}_1 \dots$ be the play induced by $\overline{\sigma}$. We claim that $\pi := s_0 s_1 \dots$ is a path in $G(\overline{z})$. Otherwise, consider the least k such that (s_k, s_{k+1}) is not an edge in $G(\overline{z})$. Hence, there exists no $\overline{z}$-secure action profile at $s := s_k$. Since $\overline{a}_k$ is certainly legal at s, there exists a player i and an action $b \in \Gamma_i(s)$ such that $\mathrm{pval}_i^{\mathcal{G}}(\delta(s, (\overline{a}_{-i}, b))) > z_i$. But then player i can improve her payoff by switching to a strategy that mimics σ_i until s is reached, then plays action b, and after that mimics a strategy τ with $\mathrm{E}_{\delta(s,(\overline{a}_{-i},b))}^{\overline{\sigma}_{-i},\tau}(\phi_i) > z_i$. This contradicts the assumption that $\overline{\sigma}$ is a Nash equilibrium. □

Using Lemmas 14 and 15, we can reduce the task of finding a pure Nash equilibrium to the task of finding a path in a multi-weighted graph whose limit-average weight vector falls between two thresholds. The latter problem can be solved in polynomial time by solving a linear programme with one variable for each pair of a weight function and an edge in the graph.

Theorem 16. *Given a finite directed graph $G = (V, E)$ with weight functions $r_0, \dots, r_{k-1} \colon V \to \mathbb{Q}$, $v_0 \in V$, and $\overline{x}, \overline{y} \in (\mathbb{Q} \cup \{\pm\infty\})^k$, we can decide in polynomial time whether there exists an infinite path $\pi = v_0 v_1 \dots$ in G with $x_i \leq \liminf_{n\to\infty} \frac{1}{n} \sum_{j=0}^{n-1} r_i(v_j) \leq y_i$ for all $i = 0, \dots, k-1$.*

We can now describe a nondeterministic algorithm to decide the existence of a pure Nash equilibrium with payoff $\geq \overline{x}$ and $\leq \overline{y}$ in polynomial time. The algorithm starts by guessing, for each player i, a positional strategy profile $\overline{\sigma}^i$ of $\mathcal{G}$ and computes $p_i(s) := \sup_\tau \mathrm{E}_s^{\overline{\sigma}^i_{-i},\tau}(\phi_i)$ for each $s \in S$; these numbers can be computed in polynomial time using the algorithm given by [18]. The algorithm then guesses a vector $\overline{z} \in (\mathbb{R} \cup \{\pm\infty\})^{\Pi}$ by setting z_i either to x_i or to $p_i(s)$ for some $s \in S$ with $x_i \leq p_i(s)$, and constructs the graph $G'(\overline{z})$, which is defined as $G(\overline{z})$ but with $p_i(s)$ substituted for $\mathrm{pval}_i^{\mathcal{G}}(s)$. Finally, the algorithm determines (in polynomial time) whether there exists an infinite path π in $G(\overline{z})$ from s_0 with $z_i \leq \phi_i(\pi) \leq y_i$ for all $i \in \Pi$. If such a path exists, the algorithm accepts; otherwise it rejects.

Theorem 17. *PureNE is in NP.*

Proof. We claim that the algorithm described above is correct, i.e. sound and complete. To prove soundness, assume that the algorithm accepts its input. Hence, there exists an infinite path π in $G'(\overline{z})$ from s_0 with $z_i \leq \phi_i(\pi) \leq y_i$. Since $\mathrm{pval}_i^{\mathcal{G}}(s) \leq p_i(s)$ for all $i \in \Pi$ and $s \in S$, the graph $G'(\overline{z})$ is a subgraph of $G(\overline{z})$. Hence, π is also an infinite path in $G(\overline{z})$. By Lemma 14, we can conclude that $(\mathcal{G}, s_0)$ has a pure Nash equilibrium with payoff $\geq \overline{z} \geq \overline{x}$ and $\leq \overline{y}$.

To prove that the algorithm is complete, let $\overline{\sigma}$ be a pure Nash equilibrium of $(\mathcal{G}, s_0)$ with payoff $\overline{z}$, where $\overline{x} \leq \overline{z} \leq \overline{y}$. By Proposition 13, the algorithm can guess positional strategy profiles $\overline{\sigma}^i$ such that $p_i(s) = \mathrm{pval}_i^{\mathcal{G}}(s)$ for all $s \in S$. If the algorithm additionally guesses the payoff vector $\overline{z}'$ defined by $z_i' = \max\{x_i, \mathrm{pval}_i^{\mathcal{G}}(s) : s \in S, \mathrm{pval}_i^{\mathcal{G}}(s) \leq z_i\}$ for all $i \in \Pi$, then the graph $G(\overline{z})$ coincides with the graph $G(\overline{z}')$ (and thus with $G'(\overline{z}')$). By Lemma 15, there exists an infinite path π in $G(\overline{z})$ from s_0 such that $z_i' \leq z_i = \phi_i(\pi) \leq y_i$ for all $i \in \Pi$. Hence, the algorithm accepts. □

The following theorem shows that PureNE is NP-hard. In fact, NP-hardness holds even for turn-based games with rewards 0 and 1.

Theorem 18. *PureNE is NP-hard, even for turn-based games with rewards* 0 *and* 1*.*

Proof. Again, we reduce from SAT. Given a Boolean formula $\varphi = C_1 \wedge \cdots \wedge C_m$ in conjunctive normal form over propositional variables $X_1, \ldots, X_n$, where w.l.o.g. $m \geq 1$ and each clause is nonempty, let $\mathcal{G}$ be the turn-based game described in the proof of Theorem 8 and depicted in Fig. 4. We claim that φ is satisfiable if and only if $(\mathcal{G}, C_1)$ has a pure Nash equilibrium with payoff ≥ 1 for player 0. □

It follows from Theorems 17 and 18 that PureNE is NP-complete. By combining our reduction with a game that has no pure Nash equilibrium, we can prove the following stronger result for non-turn-based games.

Corollary 19. *Deciding the existence of a pure Nash equilibrium in a concurrent limit-average game is NP-complete, even for games with rewards* 0 *and* 1*.*

Note that Theorem 18 and Corollary 19 do not apply to terminal-reward games. In fact, PureNE is decidable in P for these games, which follows from two facts about terminal-reward games: (1) the numbers $\mathrm{pval}_i^{\mathcal{G}}(s)$ can be computed in polynomial time (using a reduction to a turn-based two-player zero-sum game and applying a result of [31]), and (2) the only possible vectors that can emerge as the payoff of a pure strategy profile are the zero vector and the reward vectors at terminal states.

Theorem 20. *PureNE is in P for terminal-reward games.*

7 Randomised Strategies

In this section, we show that the problem NE is undecidable and, in fact, not recursively enumerable for turn-based terminal-reward games.

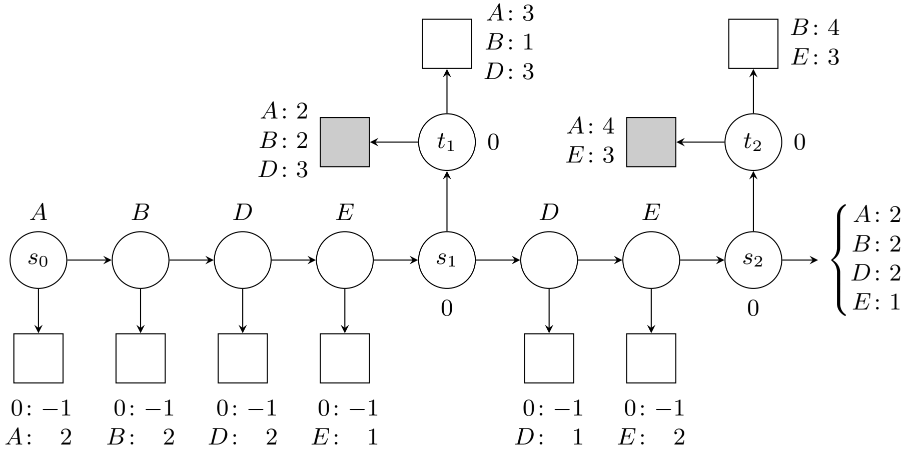


Fig. 5. Incrementing a counter

Theorem 21. *NE is not recursively enumerable, even for turn-based 14-player terminal-reward games.*

Proof (Sketch). The proof is by a reduction from the non-halting problem for two-counter machines: we show that one can compute from a deterministic two-counter machine $\mathcal{M}$ a turn-based 14-player terminal-reward game $(\mathcal{G}, s_0)$ such that the computation of $\mathcal{M}$ is infinite if and only if $(\mathcal{G}, s_0)$ has a Nash equilibrium where player 0 receives payoff ≥ 0.

To get a flavour of the full proof, let us consider a one-counter machine $\mathcal{M}$ that contains an increment instruction. A (simplified) part of the game $\mathcal{G}$ is depicted in Fig. 5. The counter values before and after the increment operation are encoded by the probabilities $c_1 = 2^{-i_1}$ and $c_2 = 2^{-i_2}$ that player 0 plays from t_1, respectively t_2, to the neighbouring grey state. We claim that $c_2 = \frac{1}{2}c_1$, i.e. $i_2 = i_1 + 1$, in any Nash equilibrium $\overline{\sigma}$ of $(\mathcal{G}, s_0)$ where player 0 receives payoff ≥ 0. First note that player 0 has to choose both outgoing transitions with probability $\frac{1}{2}$ each at s_1 and s_2 because otherwise player D or player E would have an incentive to play to a state where player 0 receives payoff < 0. Now consider the payoffs $a = \mathrm{E}^{\overline{\sigma}}_{s_0}(\phi_A)$ and $b = \mathrm{E}^{\overline{\sigma}}_{s_0}(\phi_B)$ for players A and B. We have $a, b \geq 2$ because otherwise one of these two players would have an incentive to play to a state where player 0 receives payoff < 0. On the other hand, the payoffs of players A and B sum up to at most 4 in every terminal state. Hence, $a + b \leq 4$ and therefore $a = b = 2$. Finally, the expected payoff for player A equals

$$a = \tfrac{1}{2}\bigl(c_1 \cdot 2 + (1 - c_1) \cdot 3\bigr) + \tfrac{1}{4} \cdot c_2 \cdot 4 + \tfrac{1}{4} \cdot 2 = 2 - \tfrac{1}{2}c_1 + c_2 \,.$$

Obviously, $a = 2$ if and only if $c_2 = \frac{1}{2}c_1$. □

For games that are not turn-based, by combining our reduction with the game from Example 1, we can show the stronger theorem that deciding the existence of *any* Nash equilibrium is not recursively enumerable.

Corollary 22. *The set of all initialised concurrent 14-player terminal-reward games that have a Nash equilibrium is not recursively enumerable.*

8 Conclusion

We have analysed the complexity of Nash equilibria in concurrent games with limit-average objectives. In particular, we showed that randomisation in strategies leads to undecidability, while restricting to pure strategies retains decidability. This is in contrast to stochastic games, where pure strategies lead to undecidability [26]. While we provided matching and lower bounds in most cases, there remain some problems where we do not know the exact complexity. Apart from StatNE, these include the problem PureNE when restricted to a bounded number of players.

References

1. Allender, E., Bürgisser, P., Kjeldgaard-Pedersen, J., Miltersen, P.B.: On the complexity of numerical analysis. SIAM Journal on Computing 38(5), 1987–2006 (2009)
2. Alur, R., Degorre, A., Maler, O., Weiss, G.: On omega-languages defined by mean-payoff conditions. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 333–347. Springer, Heidelberg (2009)
3. Bouyer, P., Brenguier, R., Markey, N.: Nash equilibria for reachability objectives in multi-player timed games. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 192–206. Springer, Heidelberg (2010)
4. Canny, J.: Some algebraic and geometric computations in PSPACE. In: STOC 1988, pp. 460–469. ACM Press, New York (1988)
5. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS 2010. LIPICS, vol. 8. Schloss Dagstuhl (2010)
6. Chen, X., Deng, X., Teng, S.-H.: Settling the complexity of computing two-player Nash equilibria. Journal of the ACM 56(3) (2009)
7. Conitzer, V., Sandholm, T.: Complexity results about Nash equilibria. In: IJCAI 2003, pp. 765–771. Morgan Kaufmann, San Francisco (2003)
8. Daskalakis, C., Goldberg, P.W., Papadimitriou, C.H.: The complexity of computing a Nash equilibrium. SIAM Journal on Computing 39(1), 195–259 (2009)
9. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theoretical Computer Science 386(3), 188–217 (2007)
10. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. International Journal of Game Theory 8, 109–113 (1979)
11. Etessami, K., Yannakakis, M.: On the complexity of Nash equilibria and other fixed points. SIAM Journal on Computing 39(6), 2531–2597 (2010)
12. Everett, H.: Recursive games. In: Dresher, M., Tucker, A.W., Wolfe, P. (eds.) Contributions to the Theory of Games III. Annals of Mathematical Studies, vol. 39, pp. 47–78. Princeton University Press, Princeton (1957)
13. Fink, A.M.: Equilibrium in a stochastic n-person game. Journal of Science in Hiroshima University 28(1), 89–93 (1964)
14. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 190–204. Springer, Heidelberg (2010)

15. Garey, M.R., Graham, R.L., Johnson, D.S.: Some NP-complete geometric problems. In: STOC 1976, pp. 10–22. ACM Press, New York (1976)
16. Gillette, D.: Stochastic games with zero stop probabilities. In: Dresher, M., Tucker, A.W., Wolfe, P. (eds.) Contributions to the Theory of Games III. Annals of Mathematical Studies, vol. 39, pp. 179–187. Princeton University Press, Princeton (1957)
17. Henzinger, T.A.: Games in system design and verification. In: TARK 2005, pp. 1–4. National University of Singapore (2005)
18. Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Discrete Mathematics 23(3), 309–311 (1978)
19. Mertens, J.-F., Neyman, A.: Stochastic games. International Journal of Game Theory 10(2), 53–66 (1981)
20. Nash Jr., J.F.: Equilibrium points in N-person games. Proceedings of the National Academy of Sciences of the USA 36, 48–49 (1950)
21. Neyman, A., Sorin, S. (eds.): Stochastic Games and Applications. NATO Science Series C, vol. 570. Springer, Heidelberg (2003)
22. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons, Chichester (1994)
23. Shapley, L.S.: Stochastic games. Proceedings of the National Academy of Sciences of the USA 39, 1095–1100 (1953)
24. Thuijsman, F., Raghavan, T.E.S.: Perfect-information stochastic games and related classes. International Journal of Game Theory 26, 403–408 (1997)
25. Ummels, M.: The complexity of nash equilibria in infinite multiplayer games. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 20–34. Springer, Heidelberg (2008)
26. Ummels, M., Wojtczak, D.: The complexity of nash equilibria in simple stochastic multiplayer games. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 297–308. Springer, Heidelberg (2009)
27. Ummels, M., Wojtczak, D.: The complexity of Nash equilibria in limit-average games. Tech. Rep. LSV-11-15, ENS Cachan (2011)
28. Velner, Y., Rabinovich, A.: Church synthesis problem for noisy input. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 275–289. Springer, Heidelberg (2011)
29. Vielle, N.: Two-player stochastic games I: A reduction. Israel Journal of Mathematics 119(1), 55–91 (2000)
30. Vielle, N.: Two-player stochastic games II: The case of recursive games. Israel Journal of Mathematics 119(1), 93–126 (2000b)
31. Washburn, A.R.: Deterministic graphical games. Journal of Mathematical Analysis and Applications 153, 84–96 (1990)
32. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158(1-2), 343–359 (1996)

Two Variable vs. Linear Temporal Logic in Model Checking and Games

Michael Benedikt, Rastislav Lenhardt and James Worrell

Department of Computer Science, University of Oxford, UK

Abstract. Verification tasks have non-elementary complexity for properties of linear traces specified in first-order logic, and thus various limited logical languages are employed. In this paper we consider two restricted specification logics, linear temporal logic (LTL) and two-variable first-order logic (FO^2). LTL is more expressive, but FO^2 is often more succinct, and hence it is not clear which should be easier to verify. In this paper we take a comprehensive look at the issue, giving a comparison of verification problems for FO^2, LTL, and the subset of LTL expressively equivalent to FO^2, unary temporal logic (UTL). We give two logic-to-automata translations which can be used to give upper bounds for FO^2 and UTL; we apply these to get new bounds for both non-deterministic systems (hierarchical and recursive state machines, games) and for probabilistic systems (Markov chains, recursive Markov chains, and Markov decision processes). We couple this with lower-bound arguments for FO^2 and UTL. Our results give both a unified approach to understanding the behavior of FO^2 and UTL, along with a nearly comprehensive picture of the complexity of verification for these logics.

1 Introduction

The complexity of verification problems clearly depends on the specification language for describing properties. Arguably the most important such language is *linear temporal logic* (LTL). LTL has a simple syntax, one can verify LTL properties over Kripke structures in polynomial space, and one can check satisfiability within the same complexity. Kamp [Kam68] showed that LTL has the same expressiveness as first-order logic over words. For example, the first-order property "since we are born, we live until we die":

$$\forall x.\ (born(x) \rightarrow \exists y \geq x.\ die(y) \wedge \forall z.\ (x \leq z < y \rightarrow live(z)))$$

is expressed in LTL by the formula $\Box(born \rightarrow live\, \mathcal{U}\ die)$.

In contrast with LTL, model checking first-order queries has non-elementary complexity [Sto74]—thus LTL could be thought of as a tractable syntactic fragment of FO. Another approach to obtaining tractability within first-order logic is by maintaining first-order syntax, but restricting the number of variables in subformulas to two. The resulting specification language FO^2 has also been shown to have dramatically lower complexity than full first-order logic. In particular, Etessami, Vardi and Wilke [EVW02] showed that satisfiability for FO^2 is NEXPTIME-complete and that FO^2 is strictly less expressive than FO (and, thus less than LTL

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 497–511, 2011.

also). Indeed, [EVW02] shows that FO^2 has the same expressive power as *unary temporal logic* (UTL): the fragment of LTL with only the unary operators "previous", "next", "sometime in the past", "sometime in the future".

Although FO^2 is less expressive than LTL there are some properties that are significantly easier to express in FO^2 than in LTL. Indeed, it is easy to show that there can be an exponential blow-up in transforming an FO^2 formula into an equivalent UTL formula, or even to an LTL formula. We thus have three languages UTL $\subseteq$ LTL and FO^2, with UTL and FO^2 equally expressive, and with FO^2 incomparable in succinctness with LTL.

Are verification tasks easier to perform in LTL, or in FO^2? This is the main issue we review in this paper. There are well-known examples of problems that are easier in LTL than in FO^2: in particular satisfiability, which is PSPACE-complete for LTL and NEXPTIME-complete for FO^2 [EVW02]. We will show that there are also tasks where FO^2 is more tractable than LTL.

We conduct a comprehensive analysis of the complexity of FO^2 and UTL verification problems. We begin with model checking problems for Kripke structures and for recursive state machines (RSMs), giving the complexity for UTL, and FO^2, which we compare to the (known) results for LTL. We then turn to two-player games where the winning conditions are given by UTL or FO^2 formula — here we isolate the complexity of determining which player has a winning strategy. We then move from non-deterministic systems to probabilistic systems. We start with Markov chains and recursive Markov chains, the analogs of Kripke structures and RSMs in the probabilistic case. Finally we consider one-player stochastic games, looking at the question of whether the player can devise a strategy that is winning with a given probability.

Complete proofs of the results described below will be given in a long version of this paper.

Organization: Section 2 contains preliminaries. Section 3 presents the two fundamental translations used in our upper bounds. The first is a translation of FO^2 formulas to parity automata, which extends an argument of Etessami, Vardi and Wilke [EVW02] bounding the number of types in words that satisfy a given formula. This argument will be used in many of our upper bounds for non-deterministic systems. A second translation takes a UTL formula and produces a large disjoint union of Büchi automata with special properties. The latter translation will be particularly useful for probabilistic systems. Section 4 gives upper and lower bounds for non-deterministic systems. Section 5 gives results for probabilistic systems.

2 Logics and Models

We consider a first-order signature with unary predicates $P_0, P_1, \ldots$ and binary predicates $<$ (less than) and suc (successor). An ω-word $u = u_0 u_1 \ldots$ over alphabet $\Sigma = 2^{\{P_1,\ldots,P_m\}}$ represents a first-order structure $\langle \mathbb{N}, <, \mathrm{suc}, \mathbf{P}_1, \ldots, \mathbf{P}_m \rangle$ where predicate P_i is interpreted by the set $\mathbf{P}_i = \{n \in \mathbb{N} : P_i \in u_n\}$. Fixing two distinct variables x and y, we denote by FO^2 the set of first-order formulas over the above signature involving only the variables x and y. We denote by $\mathrm{FO}^2[<]$ the sublogic in which the binary predicate suc is not used.

The formulas of linear temporal logic (with past operators) LTL are built from atomic propositions using boolean connectives and the temporal operators $\bigcirc$ (*next*), $\ominus$ (*previously*), $\Diamond$ (*eventually*), $\diamondminus$ (*sometime in the past*), $\mathcal{U}$ (*Until*), and $\mathcal{S}$ (*Since*). Formally, LTL is defined by the following grammar:

$$\varphi ::= P_i \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{S}\,\varphi \mid \Diamond\varphi \mid \diamondminus\varphi \mid \bigcirc\varphi \mid \ominus\varphi,$$

where $P_0, P_1, \ldots$ are propositional variables. Unary temporal logic (UTL) denotes the subset without $\mathcal{U}$ and $\mathcal{S}$, while $\mathrm{TL}[\Diamond, \diamondminus]$ denotes the *stutter-free* subset of UTL without $\bigcirc$ and $\ominus$. For the semantics of LTL see, e.g., [Eme90].

For φ a temporal logic formula or an FO^2 formula with one free variable, we denote by $L(\varphi)$ the set $\{w \in \Sigma^\omega : (w, 0) \models \varphi\}$ of infinite words that satisfy φ at the initial position.

The quantifier depth of an FO^2 formula φ is denoted $qdp(\varphi)$ and the operator depth of a UTL formula φ is denoted $odp(\varphi)$. In either case the length of the formula is denoted $|\varphi|$. Etessami, Vardi and Wilke gave a linear translation of UTL into FO^2. In the other direction, they showed the following:

Theorem 1 ([EVW02]). *Every* FO^2 *formula* $\varphi(x)$ *can be converted to an equivalent UTL formula* φ' *with* $|\varphi'| \in 2^{O(|\varphi|(qdp(\varphi)+1))}$ *and* $odp(\varphi') \leq 2\,qdp(\varphi)$. *The translation runs in time polynomial in the size of the output.*

Next we collect together definitions of the various different types of state machine that we consider in this paper. For non-deterministic machines we will be interested in the existence of an accepting path through the machine that satisfies a formula, while for probabilistic models we want to know the probability of such paths.

Hierarchical and Recursive State Machines. A recursive state machine (RSM) A over a set of propositions P is given by a tuple $(A_1, \ldots, A_k)$ where each component state machine $A_i = (N_i \cup B_i, Y_i, X_i, En_i, Ex_i, \delta_i)$ contains (i) a set N_i of *nodes* and a disjoint set B_i of *boxes*; (ii) an indexing function $Y_i : B_i \mapsto \{1, \ldots, k\}$ that assigns to every box an index of one of the component machines, $A_1, \ldots, A_k$; (iii) a labelling function $X_i : N_i \mapsto 2^P$; (iv) A set of *entry nodes* $En_i \subseteq N_i$ and a set of *exit nodes* $Ex_i \subseteq N_i$; (v) A *transition relation* δ_i, where transitions are of the form (u, v) where the source u is either a node of N_i, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j for $j = Y_i(b)$. We require that the destination v be either a node in N_i or a pair (b, e), where b is a box in B_i and e is an entry node in En_j for $j = Y_i(b)$.

The semantics can be found in [ABE$^+$05]. A hierarchical state machine (HSM) is an RSM in which the dependency relation between boxes is acyclic.

Markov Chains. A (labelled) *Markov chain* $M = (\Sigma, X, V, E, P, p_0)$ consists of an *alphabet* Σ, a set X of *states*; a *valuation* $V : X \to \Sigma$; a set $E \subseteq X \times X$ of *edges*; a *transition probability* P_{xy} for each pair of states $(x, y) \in E$ such that for each state x, $\sum_y P_{xy} = 1$; an *initial probability distribution* p_0 on the set of states X.

Given a language $L \subseteq \Sigma^\omega$, we denote by $P_M(L)$ the probability of the set of trajectories of M whose image under V lies in L. We consider the complexity of the following model checking problem: Given a Markov chain M and a UTL- or FO^2-formula φ, calculate $P_M(L(\varphi))$.

Recursive Markov Chains. Recursive Markov chains (RMCs) are defined as RSMs, except that the transition relation consists of triples $(u, p_{u,v}, v)$ where u and v are as with RSMs, $p_{u,v}$ are non-negative reals with $\Sigma_v p_{u,v} = 1$ or 0 for every u. The semantics of an RMC can be found in [EY05].

Markov Decision Processes. A *Markov decision process (MDP)* $M = (\Sigma, X, N, R, V, E, P, p_0)$ consists of an *alphabet* Σ, a set X of *states*, which is partitioned into a set N of *non-deterministic states* and a set R of *randomising states*; a *valuation* $V : X \to \Sigma$, a set $E \subseteq X \times X$ of *edges*, a *transition probability* P_{xy} for each pair of states $(x, y) \in E$, $x \in R$ such that $\sum_y P_{xy} = 1$; an *initial probability distribution* p_0. This model is considered in [CY95] under the name *Concurrent Markov chain.*

We can view non-deterministic states as being controlled by a scheduler, which given a trajectory leading to a non-deterministic state s chooses a transition out of s. There are two basic qualitative model checking problems: the *universal problem* asks that a given formula be satisfied with probability 1 for all schedulers; the *existential problem* asks that the formula be satisfied with probability 1 for some scheduler. The latter corresponds to the problem of designing a system that behaves correctly in a probabilistic environment.

In the *quantitative model checking problem*, we ask for the maximal probability for the formula to be satisfied on a given MDP when the scheduler chooses optimal moves in the non-deterministic states.

Two-player Games. A *two-player game* $G = (\Sigma, X, X_1, X_2, V, E, x_0)$ consists of an *alphabet* Σ; a set X of *states*, which is partitioned into a set X_1 of states controlled by *Player I* and a set X_2 controlled by *Player II*; a set of $E \subseteq X \times X$ of *transitions*; a *valuation* $V : X \to \Sigma$; an *initial state* x_0.

The game starts in the initial state and then the player who controls the current state, taking into account the whole history of the game, chooses one of the possible transitions. The verification problem of interest is whether Player I has a strategy such that for all infinite plays the induced infinite word $u \in \Sigma^\omega$ satisfies a given formula φ.

3 Translating UTL and FO^2 to Automata

We give two translations that will be the core of our upper bound techniques, capturing key insights about FO^2 and UTL formulas.

3.1 From FO^2 to Deterministic Parity Automata

We begin with a translation of FO^2 formulas to "small" deterministic parity automata. The translation relies on a small-model property of FO^2 that underlies the NEXPTIME satisfiability result of Etessami, Vardi, and Wilke [EVW02]. We give the translation first for the fragment $\mathrm{FO}^2[<]$ without successor and show later how to handle the full logic.

We consider strings over alphabet $\Sigma = 2^P$, where P is the set of unary predicates appearing in the input $\mathrm{FO}^2[<]$ formula. We write $u \sim_n v$ for two strings u, $v \in \Sigma^* \cup \Sigma^\omega$ if for all $\mathrm{FO}^2[<]$-formulas $\varphi(x)$ of quantifier depth at most n we have $(u, 0) \models \varphi$ iff $(v, 0) \models \varphi$. We refer to $\sim_n$-equivalence classes as *n-types.*

The following small-model theorem is given in [EVW02]. It is also implicit in Theorem 6.2 of [WI09].

Theorem 2. *(i) For any string $u \in \Sigma^*$ and positive integer n there exists $v \in \Sigma^*$ such that $u \sim_n v$ and $|v| \in 2^{O(|P|n)}$; (ii) for any infinite string $u \in \Sigma^\omega$ and positive integer n there are finite strings v and w, with $|v|, |w| \in 2^{O(|P|n)}$, such that $u \sim_n vw^\omega$.*

We prove the following related result.

Lemma 1. *For any string $u \in \Sigma^\omega$ and positive integer n there exists $v \in \Sigma^*$ with $|v| \in 2^{O(|P|n)}$ such that $v \sim_n u'$ for infinitely many prefixes u' of u, and $u \sim_n vw^\omega$, where w is a list of the letters occurring infinitely often in u.*

By Lemma 1 to know whether $u \in \Sigma^\omega$ satisfies an $\mathrm{FO}^2[<]$-formula of quantifier depth n it suffices to know some n-type which occurs infinitely often among prefixes of u and which letters occur infinitely often in u.

Theorem 3. *Given an $\mathrm{FO}^2[<]$ formula φ with quantifier depth n, there exists a deterministic parity automaton $\mathcal{A}_\varphi$ accepting the language $L(\varphi)$ such that $\mathcal{A}_\varphi$ has $2^{2^{O(|P|n)}}$ states, $2^{O(|P|)}$ priorities, and can be computed from φ in time $|\varphi|^{O(1)} \cdot 2^{2^{O(|P|n)}}$.*

Proof (Sketch). As $\mathcal{A}_\varphi$ reads an input string u it stores a representative of the n-type of the prefix read so far. By Theorem 2(i) the number of such representatives is bounded by $2^{2^{O(|P|n)}}$. Applying Lemma 1, we use a parity acceptance condition to determine whether u satisfies φ, based on which representatives and input letters occur infinitely often.

Extension to FO^2 with successor. By Theorem 1, given an FO^2 formula φ of quantifier depth n there is an equivalent UTL formula φ' of at most exponential size and operator depth at most $2n$. Moreover, φ' can be transformed to a normal form such that all next-time $\bigcirc$ and last-time $\ominus$ operators are pushed inside the other operators. Now we can look at φ' also as a $\mathrm{TL}[\Diamond, \diamondminus]$-formula over an extended set of predicates $P' = \{\bigcirc^k p, \ominus^k p \mid p \in P, k \leq n\}$. By a straightforward transformation we get an equivalent $\mathrm{FO}^2[<]$ formula φ' over P'. Overall, this transformation creates exponentially larger formulas, but the quantifier depth is only doubled and the set of predicates is quadratic. Applying Theorem 3 for φ' over set of predicates P' gives:

Theorem 4. *Given an FO^2 formula φ with quantifier depth n, there is a deterministic parity automaton having $2^{2^{O(n^2|P|)}}$ states and $2^{O(n|P|)}$ priorities that accepts the language $L(\varphi)$.*

3.2 From UTL to Büchi Automata

A Büchi automaton A is said to be *deterministic in the limit* if all accepting states and their descendants are deterministic; A is *unambiguous* if for each state s each word is accepted along at most one run that starts at s.

Let φ be a formula of TL$[\Diamond, \diamondminus]$ over set of atomic of propositions P. Define $cl(\varphi)$, the *closure* of φ, to consist of all subformulas of φ (including φ) and their negations, where we identify $\neg\neg\psi$ with ψ. Furthermore, say that $\boldsymbol{s} \subseteq cl(\varphi)$ is a *subformula type* if for each formula $\psi \in cl(\varphi)$ precisely one of ψ and $\neg\psi$ is a member of $\boldsymbol{s}$, $\psi \in \boldsymbol{s}$ implies $\Diamond\psi, \diamondminus\psi \in \boldsymbol{s}$, and $\psi_1 \wedge \psi_2 \in \boldsymbol{s}$ iff $\psi_1 \in \boldsymbol{s}$ and $\psi_2 \in \boldsymbol{s}$. Given subformula types $\boldsymbol{s}$ and $\boldsymbol{t}$, write $\boldsymbol{s} \sim \boldsymbol{t}$ if $\boldsymbol{s}$ and $\boldsymbol{t}$ agree on all formulas whose outermost connective is a temporal operator, i.e., for all formulas ψ we have $\Diamond\psi \in \boldsymbol{s}$ iff $\Diamond\psi \in \boldsymbol{t}$, and $\diamondminus\psi \in \boldsymbol{s}$ iff $\diamondminus\psi \in \boldsymbol{t}$.

Fix an alphabet $\Sigma \subseteq 2^P$ and write tp_φ^Σ for the set of subformula types $\boldsymbol{s} \subseteq cl(\varphi)$ with $\boldsymbol{s} \cap P \in \Sigma$. (In subsequent applications Σ will arise as the set of propositional labels in a structure to be model checked.) Following [Wol01] we define a generalised Büchi automaton $A_\varphi^\Sigma = (\Sigma, S, S_0, \Delta, \ell, \mathcal{F})$ such that $L(A_\varphi^\Sigma) = \{w \in \Sigma^\omega : (w, 0) \models \varphi\}$. The set of states is $S = tp_\varphi^\Sigma$, with the set S_0 of initial states comprising those $\boldsymbol{s} \in tp_\varphi^\Sigma$ such that $\varphi \in \boldsymbol{s}$ and $\diamondminus\psi \in \boldsymbol{s}$ only if $\psi \in \boldsymbol{s}$ for any formula ψ. The state labelling function $\ell : S \to \Sigma$ is defined by $\ell(\boldsymbol{s}) = \boldsymbol{s} \cap P$. The transition relation Δ consists of those pairs $(\boldsymbol{s}, \boldsymbol{t})$ such that (i) $\diamondminus\psi \in \boldsymbol{t}$ iff either $\psi \in \boldsymbol{t}$ or $\diamondminus\psi \in \boldsymbol{s}$; (ii) $\Diamond\psi \in \boldsymbol{s}$ and $\psi \notin \boldsymbol{s}$ implies $\Diamond\psi \in \boldsymbol{t}$; (iii) $\neg\Diamond\psi \in \boldsymbol{s}$ implies $\neg\Diamond\psi \in \boldsymbol{t}$. The collection of accepting sets is $\mathcal{F} = \{F_{\Diamond\psi} : \Diamond\psi \in cl(\varphi)\}$, where $F_{\Diamond\psi} = \{\boldsymbol{s} : \psi \in \boldsymbol{s} \text{ or } \Diamond\psi \notin \boldsymbol{s}\}$.

A run of A_φ^Σ on a word $u \in \Sigma^\omega$ yields a function $f : \mathbb{N} \to 2^{cl(\varphi)}$. Moreover if the run is accepting it can be shown that for all formulas $\psi \in cl(\varphi)$, $\psi \in f(i) \Rightarrow (u, i) \models \psi$ [Wol01, Lemma 2]. But since $f(i)$ contains each formula or its negation, we have $\psi \in f(i) \Leftrightarrow (u, i) \models \psi$ for all $\psi \in cl(\varphi)$. We conclude that A_φ^Σ is unambiguous and accepts the language $L(\varphi)$.

Consider the automaton A_φ^Σ as a directed graph with set of vertices S and set of edges Δ. Then it is easy to check that: (i) states $\boldsymbol{s}$ and $\boldsymbol{t}$ are in the same strongly connected component iff $\boldsymbol{s} \sim \boldsymbol{t}$; (ii) each strongly connected component has size at most $|\Sigma|$; (iii) the dag of strongly connected components has depth at most $|\varphi|$ and outdegree at most $2^{|\varphi|}$; (iv) A_φ^Σ is deterministic within each strongly connected component, i.e., given transitions $(\boldsymbol{s}, \boldsymbol{t})$ and $(\boldsymbol{s}, \boldsymbol{u})$ with $\boldsymbol{s}, \boldsymbol{t}$ and $\boldsymbol{u}$ in the same strongly connected component, we have $\boldsymbol{t} = \boldsymbol{u}$ or $\ell(\boldsymbol{t}) \neq \ell(\boldsymbol{u})$.

Theorem 5. *Let φ be a UTL formula over set of propositions P with operator depth n with respect to $\bigcirc$ and $\ominus$. Given an alphabet $\Sigma \subseteq 2^P$, there is a family of at most $2^{|\varphi|^2}$ automata $\{A_i\}_{i \in I}$ such that (i) $\{w \in \Sigma^\omega : w \models \varphi\}$ is the disjoint union of the languages $L(A_i)$; (ii) A_i has at most $O(|\varphi||\Sigma|^{n+1})$ states; (iii) A_i is unambiguous and deterministic in the limit; (iv) there is a polynomial-time procedure that outputs A_i given input φ and index $i \in I$.*

Proof. We first treat the case $n = 0$, i.e., φ does not mention $\bigcirc$ or $\ominus$.

Let $A_\varphi^\Sigma = (\Sigma, S, S_0, \Delta, \mathcal{F})$ be the automaton corresponding to φ, as defined above. For each path $\pi = C_0, C_1, \ldots, C_k$ of SCC's in the SCC dag of A_φ^Σ we define a sub-automaton A_π as follows. A_π has set of states $S_\pi = C_0 \cup C_1 \cup \cdots \cup C_k$; its set of initial states is $S_0 \cap S_\pi$; its transition relation is $\Delta_\pi = \Delta \cap (S_\pi \times S_\pi)$, i.e., the transition relation of A_φ^Σ restricted to S_π; its collection of accepting states is $\mathcal{F}_\pi = \{F \cap C_k : F \in \mathcal{F}\}$.

It follows from observations (ii) and (iii) preceding Theorem 5 that A_π has at most $|\varphi||\Sigma|$ states and from observation (iii) that there are at most $2^{|\varphi|^2}$ such automata. Since A_φ^Σ is unambiguous, each accepting run of A_φ^Σ yields an accepting run of A_π for a unique path π, and so the $L(A_\pi)$ partition $L(A_\varphi^\Sigma)$.

Finally A_π is deterministic in the limit since all accepting states lie in a bottom strongly connected component, and it follows from observation (iv) that all states in such a component are deterministic. Therefore we can use a standard transformation of generalised Büchi automata to regular Büchi automata, which preserves both unambiguity and being deterministic in the limit. The transformation touches only the bottom strongly connected component of A_π, which size will become at most quadratic. This completes the proof in case $n = 0$. The general case can be handled using a similar technique to the proof of Theorem 3, that is, by regarding a UTL formula φ of operator depth n as a TL[$\Diamond, \diamondminus$] formula φ' over an extended set of propositions $\{\bigcirc^i p, \ominus^i p : 0 \leq i \leq n, p \in P\}$.

4 Verifying Non-Deterministic Systems

Model checking for traditional Kripke Structures is fairly well-understood. All of our logics subsume propositional logic, and the model checking problems we deal with generalize propositional satisfiability – hence they are all NP-hard. Temporal logic without Next and Previously (TL[$\Diamond, \diamondminus$]) is known to be NP-complete, UTL and LTL are PSPACE-complete [SC82], and—as shown by Etessami, Vardi, and Wilke in [EVW02]—FO^2 model-checking is complete for NEXPTIME.

Below we extend these results to give a comparison of the complexity of model checking for recursive state machines and two-player games, applying the translations in the previous section.

Recursive State Machines. We show that FO^2 model checking for RSMs can be done as efficiently as for ordinary Kripke structures.

Theorem 6. FO^2 *model checking of RSMs can be done in NEXPTIME.*

Proof. We describe a NEXPTIME algorithm that checks satisfiability of an FO^2 sentence φ on the language of RSM A. We can convert φ to an exponential-sized formula φ' in UTL whose operator depth equals the quantifier rank of φ. By Theorem 5 it suffices to check that one of the automata A_i is satisfied on a word accepted by A. We can thus guess such an A_i (by guessing a path in the component DAG) and can then check intersection of A_i with A in polynomial time in A_i and A, by forming the product and checking that we can reach an accepting bottom strongly connected component of the product using [ABE$^+$05].

Two-player games with FO^2 winning condition. Two-player games are known to be in 2EXPTIME for LTL [PR89]. We now show that the same is true for FO^2, making use of the first translation in the previous section. We also utilise the fact that a parity game with n vertices, m edges and d priorities can be solved in time $O(dmn^d)$ [Jur00].

From this we easily conclude the 2EXPTIME upper bound:

Theorem 7. *Two-player games with* FO^2 *winning conditions are solvable in 2EXPTIME.*

Proof. Using Theorem 4, we construct in 2EXPTIME a deterministic parity automaton for the FO^2 formula φ with doubly exponentially many states and at most exponentially many priorities. By taking the product of this automaton with the graph of the game, we get a parity game with doubly exponentially many states but only exponentially many priorities. (In fact if we define the automaton over an alphabet $\Sigma \subseteq 2^P$ containing only sets of propositions that occur as labels of states in the game, then polynomially many priorities suffice.) We can then determine the winner in double exponential time.

Combining this with the result by Alur, La Torre, and Madhusudan, who showed that two-player games are 2EXPTIME-hard [ATM03] already for the simplest $\mathrm{TL}[\Diamond, \diamondminus]$, along with the fact that we can convert UTL formula to FO^2 formula in polynomial time, we get 2EXPTIME-completeness:

Corollary 1. *Deciding two-player games with* FO^2 *winning conditions is 2EXPTIME-complete.*

The following table summarises both the known results and the results from this paper (in bold) concerning non-deterministic systems.

	$\mathrm{TL}[\Diamond, \diamondminus]$	UTL	FO^2	LTL
Kripke structure	NP	PSPACE	NEXP	PSPACE
HSM	NP	PSPACE	**NEXP**	PSPACE
RSM	NP	EXPTIME	**NEXP**	EXPTIME
Two-player games	2EXP	2EXP	**2EXP**	2EXP

The PSPACE bound for model checking LTL on HSMs follows by expanding the HSMs to 'flat' Kripke structures and recalling that model checking LTL on Kripke structures can be done in space polynomial in the logarithm of the model size. Additionally, the complexity of model checking UTL and LTL on RSMs is EXPTIME-complete [BEM97] and model checking $\mathrm{TL}[\Diamond, \diamondminus]$ on RSMs is NP-complete [LTP07].

5 Verifying Probabilistic Systems

Markov chains. We begin with model checking the most basic probabilistic system, Markov chains. Courcoubetis and Yannakakis [CY95] showed that one can determine if an LTL formula holds with non-zero probability in a Markov chain in PSPACE. This gives a PSPACE upper bound for $\mathrm{TL}[\Diamond, \diamondminus]$ and an EXPSPACE upper bound for FO^2. We will show how to get better bounds, even for approximating the probability of a formula holding, using the second translation from Section 3.

Our improved complexity bounds involve the *counting classes* #P and #EXP. #P is the class of functions f for which there is a non-deterministic polynomial-time Turing Machine M such that $f(x)$ is the number of accepting computation paths of M on input x. A complete problem for #P is #SAT, the problem of counting the number of satisfying assignments of a given boolean formula. The class of functions #EXP is defined analogously, except with M a non-deterministic exponential-time machine. A decision version of #EXP was previously considered in [BFT98].

Some care is needed in characterising the complexity of probabilistic model checking problems in terms of counting classes. Such problems typically involve computing fractional values, whereas counting classes, by definition, involve integer-valued functions. The approach we take is to consider the *approximation problem* for model checking a formula φ of temporal logic or first-order logic on a Markov chain M. The approximation problem takes an integer *accuracy parameter* k as input, in addition to M and φ. The output of the approximation problem is the unique integer n such that $n/2^k \leq P_M(L(\varphi)) < (n+1)/2^k$.

Following [CSS03] we note the following property of unambiguous automata:

Lemma 2. *Given a Markov chain $M = (\Sigma, X, V, E, P, p_0)$ and a generalised Büchi automaton $A = (\Sigma, S, S_0, \ell, \Delta, \mathcal{F})$ that is unambiguous, $P_M(L(A))$ can be computed in time polynomial in M and A.*

Proof. We define a directed graph $M \otimes A$ representing the synchronised product of M and A. The vertices of $M \otimes A$ are pairs $(x, s) \in X \times S$ with the same propositional labels, i.e., such that $V(x) = \ell(s)$; the set of directed edges is $\{((x,s),(y,t)) : (x,y) \in E \text{ and } (s,t) \in \Delta\}$. We say that a bottom strongly connected component (BSCC) of $M \otimes A$ is *accepting* if for each set of accepting states $F \in \mathcal{F}$ it contains a pair (x, s) with $s \in F$.

Let $L(A, s)$ denote the set of words accepted by A starting in state s. For each vertex (x, s) of $M \otimes A$ we have a variable $\xi_{x,s}$ representing the probability $P_{M,x}(L(A, s))$ of all runs of M starting in state x that are in $L(A, s)$. These probabilities can be computed as the unique solution of the following linear system of equations:

$$\begin{aligned}
\xi_{x,s} &= 1 && (x,s) \text{ in an accepting BSCC} \\
\xi_{x,s} &= 0 && (x,s) \text{ in a non-accepting BSCC} \\
\xi_{x,s} &= \sum_{(s,t)\in\Delta} \; \sum_{y : V(y) = \ell(t)} P_{xy} \cdot \xi_{y,t} && \text{otherwise.}
\end{aligned}$$

The correctness of the third equation follows from the following calculation:

$$\begin{aligned}
P_{M,x}(L(A,s)) &= P_{M,x}(\bigcup_{(s,t)\in\Delta} \ell(s)\cdot L(A,t)) \\
&= \sum_{(s,t)\in\Delta} P_{M,x}(\ell(s)\cdot L(A,t)) \quad \text{(since } A \text{ is unambiguous)} \\
&= \sum_{(s,t)\in\Delta} \sum_{y:V(y)=\ell(t)} P_{xy}\cdot P_{M,y}(L(A,y))
\end{aligned}$$

We show how this helps with model checking a very minimal temporal language:

Corollary 2. *The approximation problem for model checking a* TL[$\Diamond, \ominus$] *formula φ on a Markov chain M is in #P if the accuracy parameter k is given in unary.*

Proof. Let $\Sigma \subseteq 2^P$ be the set of propositional labels appearing in M. Using Theorem 5, we have that for formula φ there is a family $\{A_i\}$ comprising at most $2^{|\varphi|^2}$ unambiguous generalised Büchi automata whose languages partition $\{w \in \Sigma^\omega : w \models \varphi\}$. Moreover, each A_i has at most $|\varphi||\Sigma|$ states and can be generated in polynomial time from φ and index i. By Lemma 2 we can further compute the probability p_i of M satisfying A_i in polynomial time in the sizes of M and A_i.

We can approximate each p_i a by dyadic rational $a_i/2^b$, where b is large enough so that $\sum_i a_i/2^b$ is within 2^{-k} of $\sum_i p_i$. Since each a_i is computable in polynomial time we can compute $\sum_i a_i$ in $\#P$.

The same technique applies to FO^2 by first translating FO^2 formulas to equivalent UTL formulas using Theorem 1. In fact the extra expressiveness of #EXP allows us to give the accuracy parameter in binary.

Theorem 8. *The approximation problem for model checking an* FO^2 *formula φ on a Markov chain M is in #EXP if the accuracy parameter is given in binary.*

We can get corresponding tight lower bounds.

Theorem 9. *The approximation problem for model checking* TL[$\Diamond, \ominus$] *on Markov chains is #P-hard.*

Theorem 10. *The approximation problem for model checking* FO^2 *on Markov chains is #EXP-hard.*

Proof. $\#P$-hardness is proven by reduction from #SAT. Given a propositional formula φ one has a Markov chain generate a uniform distribution over strings that code possible truth assignments for φ. The temporal logic formula simply checks whether φ is satisfied by the assignment encoded in a given string.

#EXP-hardness is by reduction from the problem of counting the number of accepting paths of a NEXPTIME Turing machine M. The Markov chain generates a uniform distribution over strings of the appropriate length and the formula checks whether a given string encodes an accepting computation of M. The ability of FO^2 to check validity of such a string was already exploited in the NEXPTIME-hardness proof for FO^2 satisfiability in [EVW02].

Hierarchical and Recursive Markov Chains. For an RMC A, we can compute reachability probabilities $q_{(u,ex)}$ of exiting a component A_i starting at state $u \in V_i$ going to exit $ex \in Ex_i$. Etessami and Yannakakis [EY05] show that these probabilities are the unique solution of a system of non-linear equations which can be found in polynomial space using a decision procedure for the existential theory of the reals. Following [EY05] for every vertex $u \in V_i$ we let $ne(u) = 1 - \sum_{ex \in Ex_i} q_{(u,ex)}$ be the probability that a trajectory beginning from node u never exits the component A_i of u. Etessami and Yannakakis [YE05] also show that one can check properties specified by deterministic Büchi automata in PSPACE, while for non-deterministic Büchi automata they give a bound of EXPSPACE. Thus the prior results would give a bound of EXPSPACE for UTL and 2EXPSPACE for FO^2. We will improve upon both these bounds. We observe that the technique of [YE05] can be used to check properties specified by non-deterministic Büchi automata that are unambiguous in the same complexity as deterministic ones. This will then allow us to apply our second translation from Section 3 to both UTL and FO^2.

Theorem 11. *Given an unambiguous Büchi automaton B and a RMC A, we can compute the probability that B accepts a trajectory of A in PSPACE.*

Proof. Let B be an unambiguous Büchi automaton with set of states Q, transition function Δ and labelling function ℓ. Let A be an RMC with valuation V. We define a product RMC $A \otimes B$ with component and call structure coming from A whose states are pairs (x, s), with x a state of A and s a state of B such that $V(x) = \ell(s)$ (i.e., x and s have the same label). Such a pair (x, s) is accepting if s is an accepting state of B. A run through the product chain is accepting if at least one of the accepting states is visited infinitely often. Note that a path through A may expand to several runs in $A \otimes B$ since B is non-deterministic.

For each i, for each vertex $x \in V_i$, exit $ex \in Ex_i$ and states $s, t \in Q$ we define $p(x, s \to ex, t)$ to be the probability that a trajectory in RMC A that begins from a configuration with state x and some non-empty context (i.e. not at top-level) expands to an accepting run in $A \otimes B$ from (x, s) to (ex, t).

Just as in the case of deterministic automata, we can compute $p(x, s \to ex, t)$ as the solution of the following system of non-linear equations:

If $x \in V_i$ is not entrance of the box we have:

$$p(x, s \to ex, t) = \sum_{x' : (x, P_{xx'}, x') \in \delta_i} P_{xx'} \sum_{s' : (s, s') \in \Delta \wedge \ell(s') = V(x')} p(x', s' \to ex, t)$$

If $x \in V_i$ is entrance of the box $b \in B_i$ then we include the equations:

$$p(x, s \to ex, t) = \sum_{j, s' \in Q} p((b, en), s \to (b, ex_j), s') p((b, ex_j), s' \to ex, t)$$

where $p((b, en), s \to (b, ex_j), s') = p(en_{Y_i(b)}, s \to ex_j, s')$ and $ex_j \in Ex_{Y_i(b)}$.

The justification for these equations is as follows. Since B is unambiguous, each trajectory of A expands to at most one accepting run of $A \otimes B$. Thus in summing over automaton states s' in the two equations above we are summing probabilities over disjoint events which correctly gives us the probability of the union of these events.

We now explain how these probabilities can be used to compute the probability of acceptance. We assume without loss of generality that the transition function of B is total.

We construct a finite-state *summary chain* for the product $A \otimes B$ exactly as in the case of deterministic automata [YE05]. For each component A_i of A, vertex x of A_i, exit $ex \in Ex_i$ and for each pair of states s, t of B the probability to transition from (x, s) to (ex, t) in the summary chain is calculated from $p(x, s \to ex, t)$ after adjusting for probability $ne(x)$ that A never exits A_i starting at vertex x. Note that since automaton B is non-blocking, the probability of never exiting the current component of $A \otimes B$ starting at (x, s) is the same as $ne(x)$ (the probability of never exiting the current component from vertex x in the RMC A alone).

To summarise, we first compute reachability probabilities $q_{(u,ex)}$ and probabilities $ne(u)$ for the RMC A. Then we consider the product $A \otimes B$ and solve a system of non-linear equations to compute the probabilities of summary transitions $p(x, s \to ex, t)$. From these data we build the summary chain, identify accepting SCCs and compute the resulting probabilities in the same way as in [YE05]. All these steps can be expressed as a formula and its truth value can be decided using existential theory of the reals in PSPACE.

Using the translation from Theorem 5 and Theorem 11 we immediately obtain upper bounds for FO^2 and for $\mathrm{TL}[\Diamond, \diamondminus]$:

Theorem 12. *The probability of an* FO^2 *formula holding on an RMC can be computed in EXPSPACE.*

Theorem 13. *The probability of a* $\mathrm{TL}[\Diamond, \diamondminus]$ *formula holding on an RMC can be computed in PSPACE.*

Proof. By Theorem 5 we can convert a $\mathrm{TL}[\Diamond, \diamondminus]$ formula φ into an equivalent disjoint union of $2^{|\varphi|^2}$ unambiguous automata of polynomial size in $|\varphi|$ and the RMC. Using polynomial space we can therefore generate each automaton, calculate the probability that the RMC generates an accepting trajectory, and sum these probabilities for each automaton.

For an ordinary Markov chain, calculating the probability of an LTL formula can be done in PSPACE [Yan10], while we have seen previously that we can calculate the probability of an FO^2 formula in #EXP. One can achieve the same bounds for LTL and FO^2 on hierarchical Markov chains. In each case we expand the HMC into an ordinary Markov chain and then use the model checking algorithm for a Markov chain. This does not impact the complexity, since the space complexity is only polylog in the size of the machine for LTL and the time complexity is only polynomial in the machine size for FO^2. We thus get:

Theorem 14. *The probability of a* FO^2 *formula holding on a HMC can be computed in #EXP, while for an LTL formula it can be computed in PSPACE.*

Markov Decision Processes. Courcoubetis and Yannakakis [CY95] have shown that the maximal probability with which a scheduler can achieve an UTL objective on an MDP can be computed in 2EXPTIME. It follows from results of [ATM03] that even the qualitative problem of determining whether every scheduler achieves probability 1 is 2EXPTIME-hard. Combining the 2EXPTIME upper bound with the exponential translation from FO^2 to UTL [EVW02] yields a 3EXPTIME bound for FO^2. Below we see that using our FO^2-to-automaton construction we are able to do "exponentially better".

We begin with universal formulation of qualitative model checking MDPs. Here we can apply the second translation to get bounds:

Theorem 15. *Determining whether for all schedulers a* $\mathrm{TL}[\Diamond, \diamondminus]$*-formula* φ *holds on a Markov decision process* M *with probability* 1 *is in co-NP.*

Proof. The corresponding complement problem asks whether there exists a scheduler σ such that the probability is greater than 0. For this problem, there is an NP algorithm. We can just guess a particular A_i from Theorem 5 corresponding to one of the automata for φ.

It is easy to see that the bound is tight, since qualitative model checking for MDPs generalizes validity for $\mathrm{TL}[\Diamond, \diamondminus]$ formulas, which is co-NP hard.

Theorem 16. *Determining whether for all schedulers a UTL-formula* φ *holds on a Markov decision process* M *with probability* 1 *is in EXPTIME. For* FO^2 *the problem is in co-NEXPTIME.*

Proof. In [CY95], there is a polynomial algorithm for qualitative model checking deterministic Büchi automata on MDPs. As noted there, the algorithm applies to automata that are deterministic in the limit as well. Applying Theorem 5, we can apply the algorithm to each A_i, giving a single exponential algorithm.

The result for FO^2 follows along the lines of the proof of Theorem 15.

Note that here the FO^2 problem is *easier* than the corresponding LTL problem, which is known to be 2EXPTIME-complete.

Turning to lower bounds, note that co-NEXPTIME-hardness for FO^2 is inherited from the lower bound for Markov chains. On the other hand, we can show that the EXPTIME bound for UTL is tight:

Theorem 17. *Determining whether for all schedulers a UTL-formula* φ *holds on a Markov decision process* M *with probability* 1 *is EXPTIME-hard.*

Proof. The argument is based on the idea of Courcoubetis and Yannakakis for lower bounds in the LTL case. We reduce from the problem of determining whether an alternating PSPACE machine M accepts, to the problem of existence of a scheduler that enforces that a UTL formula holds with non-zero probability — clearly the latter is equivalent to qualitative model checking. The probabilistic environment will play the role of one player in the alternating PSPACE computation, proposing moves of one player with equal probability. The scheduler will play the role of the other player, proposing responses non-deterministically. The main difference between the MDP game structure and the configuration graph of M is that when the configuration reaches an acceptance state, it moves back to the initial state — this ensures that the game is played repeatedly, which can be used to amplify any positive probability of non-acceptance. We can give a UTL formula that will check that the full run is winning for the scheduler. If the alternating machine accepts, the winning strategy for the computation results in a scheduler for the game that can enforce the formula with probability 1. If the alternating machine does not accept, then any scheduler will have probability 0 of enforcing the probability.

For the existential case of the qualitative model-checking problem, an upper bound of 2EXPTIME for all of our languages will follow from the quantitative case below. On the other hand the arguments from [ATM03] can be adapted to get a 2EXPTIME lower bound even for qualitative model-checking $\mathrm{TL}[\Diamond, \diamondminus]$ in the existential case. Hence we have:

Theorem 18. *Determining if there is a scheduler that enforces a formula with probability one is 2EXPTIME-complete for each of* $\mathrm{TL}[\Diamond, \diamondminus], \mathrm{UTL}, \mathrm{LTL}, \mathrm{FO}^2$.

We now turn to the quantitative case. We apply the translation from FO^2 to deterministic parity automata from Section 3, along with the result that the value of a Markov decision process with parity winning objective can be computed in polynomial time [CH11]. Using Theorem 3 we immediately get bounds for FO^2 which match the known bounds for LTL:

Theorem 19. *We can compute the maximum probability of an* FO^2 *formula* φ *over all schedulers on a Markov decision processes* M *in 2EXPTIME.*

The table to follow summarizes the known results and the results from this paper (in bold) on probabilistic systems. An asterisk indicates bounds that are not known to be tight.

	$\mathrm{TL}[\Diamond, \diamondminus]$	UTL	FO^2	LTL
Markov chain	**#P**	PSPACE	**#EXP**	PSPACE
HMC	PSPACE*	PSPACE	**#EXP**	PSPACE
RMC	**PSPACE***	EXPSPACE*	**EXPSPACE***	EXPSPACE*
MDP (∀)	**co-NP**	**EXP**	**co-NEXP**	2EXP
MDP (∃)	2EXP	2EXP	**2EXP**	2EXP
MDP (quant)	2EXP	2EXP	**2EXP**	2EXP

References

[ABE$^+$05] Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. 27, 786–818 (2005)

[ATM03] Alur, R., La Torre, S., Madhusudan, P.: Playing games with boxes and diamonds. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 128–143. Springer, Heidelberg (2003)

[BEM97] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

[BFT98] Buhrman, H., Fortnow, L., Thierauf, T.: Nonrelativizing separations. In: COCO (1998)

[CH11] Chatterjee, K., Henzinger, T.A.: A survey of stochastic ω-regular games. J. Comput. Syst. Sci (2011)

[CSS03] Couvreur, J.-M., Saheb, N., Sutre, G.: An optimal automata approach to LTL model checking of probabilistic systems. In: Logic for Programming, A.I., and Reasoning, pp. 361–375 (2003)

[CY95] Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)

[Eme90] Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics, pp. 995–1072. Elsevier, Amsterdam (1990)

[EVW02] Etessami, K., Vardi, M.Y., Wilke, T.: First-order logic with two variables and unary temporal logic. Inf. and Comp. 179(2), 279–295 (2002)

[EY05] Etessami, K., Yannakakis, M.: Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 340–352. Springer, Heidelberg (2005)

[Jur00] Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)

[Kam68] Kamp, H.W.: Tense Logic and the Theory of Linear Order. PhD thesis, UCLA (1968)

[LTP07] La Torre, S., Parlato, G.: On the complexity of LTL model-checking of recursive state machines. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 937–948. Springer, Heidelberg (2007)

[PR89] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)

[SC82] Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. In: STOC, pp. 159–168 (1982)

[Sto74] Larry, J.: Stockmeyer. The Complexity of Decision Problems in Automata Theory and Logic. PhD thesis, MIT, Cambridge, Massasuchets, USA (1974)

[WI09] Weis, P., Immerman, N.: Structure theorem and strict alternation hierarchy for FO^2 on words. In: LMCS, vol. 5(3) (2009)

[Wol01] Wolper, P.: Constructing automata from temporal logic formulas: A tutorial. In: Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.) EEF School 2000 and FMPA 2000. LNCS, vol. 2090, pp. 261–277. Springer, Heidelberg (2001)

[Yan10] Yannakakis, M.: Personal communication (2010)

[YE05] Yannakakis, M., Etessami, K.: Checking LTL properties of Recursive Markov Chains. In: QEST, pp. 155–165 (2005)

A Compositional Framework for Controller Synthesis

Christel Baier, Joachim Klein, and Sascha Klüppelholz[⋆]

Technische Universität Dresden, Germany

Abstract. Given a system $\mathcal{A}$ and objective Φ, the general task of controller synthesis is to design a decision making policy that ensures Φ to be satisfied. This paper deals with LTS-like system models and controllers that make their decisions based on the observables of the actions performed so far. Our main contribution is a compositional framework for treating multiple linear-time objectives inductively. For this purpose, we introduce a novel notion of strategies that serve as generators for observation-based decision functions. Our compositional approach will rely on most general (i.e., most permissive) strategies generating *all* decision functions that guarantee the objective under consideration. Finally we show that for safety and co-safety objectives Φ, most general strategies are realizable by finite-state controllers that exogenously enforce Φ.

1 Introduction

The starting point of the classical controller synthesis problem is a formal model $\mathcal{A}$ for an open system (often called plant) and an objective Φ that formalizes the desired system properties and is typically given as a temporal formula. The model describes the possible interactions of the plant with its environment and typically relies on some nondeterministic automata model such as labeled transition systems (LTS). The task is then to design a controller that restricts the possible behaviors of $\mathcal{A}$ (i.e., discards certain nondeterministic alternatives) such that the controlled system meets the specification. Several instances of the controller synthesis problem have been studied in the literature that differ in the type of system models and objectives, the assumptions on what is visible to the controller and the way how the environment and controller interact with $\mathcal{A}$. For example, the problem of realizability of specifications, i.e., the synthesis of a stand-alone controller directly from a specification, has been treated in [1,12,15,8]. The synthesis problem has been studied for various types of systems including discrete event systems [16], hybrid systems [2], and timed systems [3]. We address here the case where the system $\mathcal{A}$ is described by some nondeterministic LTS-like automaton that models the parallel composition of controllable components and their (partly observable, but uncontrollable) environment. The controller $\mathfrak{C}$ is again an automaton representing a component that decides on the "legal" nondeterministic options in an exogenous manner, i.e., controls $\mathcal{A}$

[⋆] The authors are supported by the DFG-project Syanco.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 512–527, 2011.

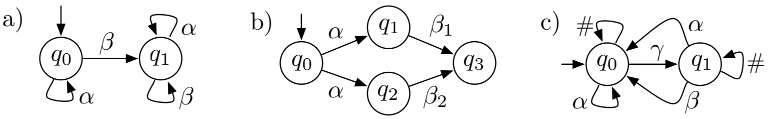


Fig. 1. Illustrating examples

from outside by running in parallel to $\mathcal{A}$ with synchronization over the scheduled actions. Given $\mathcal{A}$ and an objective Φ, soundness of the controller means that $\mathfrak{C} \bowtie \mathcal{A} \models \Phi$ where $\bowtie$ denotes parallel composition. Similar scenarios have been studied in the literature. For instance, in [10] the synthesis problem is studied for open systems with complete information in reactive environments and CTL* objectives. In distributed controller synthesis [13,11], the focus is on local controllers with no global knowledge for the individual components making up the system, working together to enforce the objective.

Our focus is to find a *compositional* approach for constructing a controller for a single component within a *partially observable* environment and *partial control.* Compositionality means that our approach is suitable to treat cascades of linear-time objectives $\Phi_1, \Phi_2, \ldots, \Phi_k$ in an online manner, i.e., first construct a controller $\mathfrak{C}_1$ for system $\mathcal{A}$ enforcing Φ_1, then a controller $\mathfrak{C}_2$ for system $\mathfrak{C}_1 \bowtie \mathcal{A}$ enforcing objective Φ_2, and so on, such that

$$\mathfrak{C}_k \bowtie \ldots \bowtie \mathfrak{C}_2 \bowtie \mathfrak{C}_1 \bowtie \mathcal{A} \models \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_k$$

if the conjunction can be enforced for $\mathcal{A}$. It is crucial for compositionality that each controller $\mathfrak{C}_i$ enforces Φ_i in a *most general* manner, i.e., being as permissive as possible and thus not ruling out ways to enforce Φ_i that may be needed to enforce subsequent objectives. Kuiper and van de Pol [9] consider most general strategies in a partial observation setting, but their approach is limited to safety objectives. Bernet et al. [4] introduce the notion of permissive strategies in a complete information setting and show that such permissive strategies only exist for safety objectives. Bouyer et al. [6] consider a different notion of permissiveness in a quantitative setting where a penalty value for non-permissiveness is minimized, calculated from weights on actions.

Safety objectives can be straightforwardly enforced in a most general manner by offering all the "legal actions" that do not lead to the violation of the objective. Fig. 1a illustrates by way of a well-known example why the standard notion of strategies relying on such a concept of "legal actions" are not sufficient to deal with objectives beyond safety in a most general manner. Suppose that actions α, β are controllable and α and β agree with their observables. Clearly, first scheduling α for a finite amount of time and then scheduling β enforces the reachability objective $\Diamond q_1$ ("eventually q_1"). Thus, both α and β are "legal" in state q_0 for the objective $\Diamond q_1$. However, offering both α and β in state q_0 does not guarantee $\Diamond q_1$ to hold as it does not avoid the computation that always takes α and stays forever in state q_0.

Our main contribution is a compositional framework for objectives beyond safety in a partial information setting, relying on a novel notion of strategies as generators of decision functions. Each decision function represents one particular way in which an objective can be enforced in an observation-based manner. Most general strategies are then those strategies that generate all decision functions that enforce a given objective, and controllers are realizations of strategies with finite memory. The strategies and controllers are equipped with a fairness condition and generate the decision function instances in an observation-based manner by choosing from all the sets of "legal actions" in a fair way. We are not aware of any other paper where a similar definition of most general strategies as generators for decision functions has been studied. We show the compositionality of most general strategies and their controllers for arbitrary objectives that can be enforced by controllers realizing most general strategies. Furthermore, we show the existence of most general strategies and the compositional construction of controllers realizing them for conjunctions $\varPhi = \varPhi_1 \wedge \ldots \wedge \varPhi_k$ of safety and co-safety objectives if $\varPhi$ is enforceable.

Reconsider the automaton in Fig. 1a. The decision functions enforcing $\Diamond q_1$ are those that schedule the singleton $\{\beta\}$ for at least one of the observations in α^*. A most general strategy that generates exactly these decision functions will offer the alternatives $\{\alpha\}$, $\{\beta\}$ and $\{\alpha, \beta\}$ at all times and imposes the fairness constraint that eventually the singleton $\{\beta\}$ will be scheduled. As there is no global bound $k \in \mathbb{N}$ for the number of observed α before a decision function enforcing $\Diamond q_1$ has to schedule $\{\beta\}$, strategies relying on a counting mechanism are not sufficient. In the context of realizability of specifications in Linear Temporal Logic (LTL), Filiot et al. [8] rely on such a counting mechanism to obtain safety objectives for all parts of the specification, using a bound depending on the whole specification. Our approach utilizes strategies and controllers that are truly *most general* and are thus independent of the additional objectives considered subsequently.

One of the challenges in any compositional approach is the treatment of termination, as different components may block each other. For the most general handling of only safety objectives, such finite behavior can be handled implicitly, as all prefixes of safe executions are safe themselves. For objectives beyond safety as treated in this paper, we have to take termination into account to ensure compositionality. The strategies and controllers enforcing subsequent objectives have to ensure that they do not cause termination when previously applied controllers require that no termination occurs to enforce their objectives, e.g., for reachability objectives. For this purpose, we suppose that the controllers and the controllable component may synchronize over special *suspend actions* that push the controllable component into a sleep mode, deactivating the controllable actions. The controller still observes visible actions by the environment and has the option to reactivate the controllable component. A suitable notion of admissibility then ensures that a controller must not cause termination in non-terminal states, unless it is in sleep mode.

Outline. Section 2 introduces the formal notions of decision functions, strategies and controllers used in our compositional framework. Section 3 then shows the adequacy for compositional reasoning. Section 4 outlines the game-based approach for constructing controllers for safety and co-safety objectives.

2 Decision Functions, Strategies and Controllers

Let Σ be an alphabet. Then $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ denotes the set of finite and infinite words over Σ. $f(x) = \bot$ denotes that function f is undefined for argument x.

Automata. An *automaton* is a tuple $\mathcal{A} = (Q, Act, \longrightarrow, Q_0)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, Act is a finite set of actions, and $\longrightarrow \subseteq Q \times Act \times Q$ the transition relation. Action α is *enabled* in state $q \in Q$ iff $q \xrightarrow{\alpha} q'$ for some $q' \in Q$. $Act(q)$ denotes the set of enabled actions in state q. An *execution* in $\mathcal{A}$ is a finite or infinite sequence built by consecutive transitions $\pi = q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} q_2 \xrightarrow{\alpha_3} \ldots$. If π is finite then $last(\pi)$ denotes its last state and $|\pi|$ the total number of transitions. Execution π is called *initial* if $q_0 \in Q_0$.

Throughout the paper, $\mathcal{A} = (Q, Act, \longrightarrow, Q_0)$ denotes an automaton representing the behavior of the controllable component and the environment. The actions in Act are classified as being either *controllable* or *uncontrollable* and as either *visible* or *invisible* for the controllable component. Let $Act_{\mathrm{vis}} \subseteq Act$ be the set of visible actions and let $Act_{\mathrm{ctr}} \subseteq Act_{\mathrm{vis}}$ be the set of controllable actions, i.e., we assume that all controllable actions are visible. The subset $Act_{\#} \subseteq Act_{\mathrm{ctr}}$ of the controllable actions contains the actions that signal the suspension of the controllable component, which deactivates the controllable actions until an uncontrollable, visible action occurs and the controllable component resumes.

Observables, observations. The visible actions can be observed via their *observables*. Let Obs be a finite set of observables and let $obs : Act_{\mathrm{vis}} \to Obs$ be a function assigning an observable to each visible action $\alpha \in Act_{\mathrm{vis}}$ such that there are sets $Obs_{\#} \subseteq Obs_{\mathrm{ctr}} \subseteq Obs$ with $obs(\alpha) \in Obs_{\mathrm{ctr}}$ iff $\alpha \in Act_{\mathrm{ctr}}$ and $obs(\alpha) \in Obs_{\#}$ iff $\alpha \in Act_{\#}$. We naturally extend obs for action sequences by concatenating the observables, i.e., $obs : Act^\infty \to Obs^\infty$ with $obs(\alpha_1\alpha_2\alpha_3\ldots) = obs(\alpha_1)obs(\alpha_2)obs(\alpha_3)\ldots$, where $obs(\alpha)$ is the empty word ε for invisible actions α. We refer to the elements of Obs^∞ as *observations*. The observation $obs(\pi)$ of a finite or infinite execution π in $\mathcal{A}$ is the observation $obs(\alpha_1\alpha_2\alpha_3\ldots)$ of its action sequence. An observation $\sigma \in Obs^\infty$ is called $\mathcal{A}$-*schedulable*, briefly *schedulable*, if there exists an initial execution π in $\mathcal{A}$ with $obs(\pi) = \sigma$.

Annotated observables, observations. Later, in the definitions of strategies and controllers, we label the observables in Obs with annotations from a finite, non-empty set $\mathfrak{Ann}$. Let $\mathfrak{Obs} = Obs \times \mathfrak{Ann}$ denote the set of annotated observables. An *annotation function* represents a policy to decorate observables with annotations in a history-dependent manner. Formally, an annotation function is a function $\mathfrak{ann} : Obs^* \times Obs \to \mathfrak{Ann}$. It induces an inductively defined transformation $\mathfrak{ann}^* : Obs^* \to \mathfrak{Obs}^*$ of observations into their annotated versions: $\mathfrak{ann}^*(\varepsilon) = \varepsilon$ and $\mathfrak{ann}^*(\sigma\beta) = \mathfrak{ann}^*(\sigma)\langle\beta, \mathfrak{ann}(\sigma, \beta)\rangle$ for all $\sigma \in Obs^*$ and $\beta \in Obs$. To strip the annotations, we write $\langle\beta, \mathfrak{a}\rangle|_{Obs} = \beta$, $\sigma'|_{Obs} = \beta_1\beta_2\ldots$ for $\sigma' = \langle\beta_1, \mathfrak{a}_1\rangle\langle\beta_2, \mathfrak{a}_2\rangle\ldots \in \mathfrak{Obs}^\infty$ and $\mathfrak{O}|_{Obs} = \{\beta'|_{Obs} : \beta' \in \mathfrak{O}\}$ for $\mathfrak{O} \in 2^{\mathfrak{Obs}}$.

***O*-compliance.** For $O \in 2^{Obs}$, action α is called O-compliant if either α is invisible or it is visible with an observable in O. The set of O-compliant actions in state q is $Act(q, O) = \big\{\alpha \in Act(q) : obs(\alpha) \in O \text{ or } \alpha \notin Act_{\mathrm{vis}}\big\}$.

Non-blocking, passive. To ensure that the controllable component can not refuse actions in $Act_{\text{vis}} \setminus Act_{\text{ctr}}$, it must always offer observables for these actions. This condition can be understood as "input enabledness" for the visible, but uncontrollable actions. For this purpose, we use a partition $\mathcal{U}_{Obs} = \{U_1, \ldots, U_k\}$ of $Obs \setminus Obs_{\text{ctr}}$ that groups the observations of the visible, but uncontrollable actions and require that decision functions must offer at least one observable of each group. Intuitively, each group U_i stands for some visible, but uncontrollable action that might be annotated by the controller in different ways, yielding different observables for the same action. The latter will be crucial for compositionality. Initially, there are no annotations and each visible, but uncontrollable action constitutes its own group, i.e., $\mathcal{U}_{Obs} = \big\{\{obs(\alpha)\} : \alpha \in Act_{\text{vis}} \setminus Act_{\text{ctr}}\big\}$.

A set $O \in 2^{Obs}$ is *non-blocking* if $U \cap O \neq \varnothing$ for each $U \in \mathcal{U}_{Obs}$. We say a set $O \in 2^{Obs}$ is *passive* if O is non-blocking and $O \cap Obs_{\text{ctr}} = \varnothing$.

Decision functions describe observation-based scheduling policies for the controllable component to "offer" sets of interactions. A *decision function* is a function $\mathfrak{d} : Obs^* \to 2^{Obs}$ such that $\mathfrak{d}(\sigma)$ is non-blocking for all $\sigma \in Obs^*$ and $\mathfrak{d}(\sigma)$ is passive for all $\sigma \in Obs^* Obs_{\#}$. The first condition ensures that the controllable component cannot avoid visible but uncontrollable actions to be taken. The second condition ensures that after being suspended, the controllable component becomes passive. As soon as the next visible and uncontrollable action is performed, it switches back to the normal mode.

To illustrate why we have to deal with decision functions that schedule sets of observables rather than single observables, consider the automaton in Fig. 1b. Suppose that α, β_1, β_2 are controllable actions and that they agree with their observables. The reachability objective $\Diamond q_3$ ("eventually q_3") can be enforced by the decision function that first offers observable α and then offers both observables β_1 and β_2, while there is no decision function enforcing $\Diamond q_3$ that offers single observables only.

Executions and paths. The notion of a path captures "complete" behavior, relying on a maximal progress assumption for the environment and the controllable component, with special treatment for the suspend mode. An execution π in $\mathcal{A}$ is called a path in $\mathcal{A}$ if one of the following three conditions (1), (2) or (3) holds: (1) π is infinite, or (2) π is finite and ends in a terminal state, i.e., $Act(last(\pi)) = \varnothing$, or (3) π is finite and the last observable is a suspend signal (i.e., $obs(\pi) \in Obs^* Obs_{\#}$) and $Act(last(\pi)) \setminus Act_{\text{ctr}} = \varnothing$. Later, in Section 3, we will augment the automata with a suitable fairness condition and will update the definition of paths to take this fairness into account.

Given a decision function $\mathfrak{d}$, an execution $\pi = q_0 \xrightarrow{\alpha_1}_{\mathcal{A}} q_1 \xrightarrow{\alpha_2}_{\mathcal{A}} q_2 \xrightarrow{\alpha_3}_{\mathcal{A}} \ldots$ in $\mathcal{A}$ is a $\mathfrak{d}$-*execution* if, for all $i < |\pi|$, $\alpha_{i+1} \in Act(q_i, \mathfrak{d}(obs(\alpha_1 \ldots \alpha_i)))$, i.e., that α_{i+1} is a $\mathfrak{d}(obs(\alpha_1 \ldots \alpha_i))$-compliant action in state q_i. A $\mathfrak{d}$-*path* denotes either an infinite $\mathfrak{d}$-execution or a finite $\mathfrak{d}$-execution as above with $Act(last(\pi), \mathfrak{d}(obs(\pi))) = \varnothing$. An observation $\sigma \in Obs^\infty$ is said to be $\mathfrak{d}$-*schedulable* if there exists an initial $\mathfrak{d}$-execution π such that $\sigma = obs(\pi)$. An annotated observation $\sigma' =$

$\beta'_1 \ldots \beta'_n \in \mathfrak{Obs}^*$ with $\mathfrak{Obs} = \mathit{Obs} \times \mathfrak{Ann}$ is called $\mathfrak{d}'$-schedulable for a function $\mathfrak{d}' : \mathfrak{Obs}^* \to 2^{\mathfrak{Obs}}$ if $\sigma'|_{\mathit{Obs}}$ is $\mathcal{A}$-schedulable and $\beta'_{i+1} \in \mathfrak{d}'(\beta'_1 \ldots \beta'_i)$ for $i < n$.

Admissibility. Clearly, each $\mathfrak{d}$-execution is an execution in $\mathcal{A}$. However, the corresponding statement for paths does not hold in general. If the last observation is not a suspend signal, then a finite $\mathfrak{d}$-path might not be a path in $\mathcal{A}$. Decision function $\mathfrak{d}$ is called *admissible* if there is no finite, initial $\mathfrak{d}$-path $\pi = q_0 \xrightarrow{\alpha_1}_{\mathcal{A}} \ldots \xrightarrow{\alpha_n}_{\mathcal{A}} q_n$, with $\mathit{obs}(\alpha_1 \ldots \alpha_n) \notin \mathit{Obs}^* \mathit{Obs}_{\#}$ and $\mathit{Act}(q_n) \neq \varnothing$. Obviously, $\mathfrak{d}$ is admissible iff each $\mathfrak{d}$-path is a path in $\mathcal{A}$. We will later adapt the notion of admissibility (and the notions that rely on admissibility) to take fairness conditions of $\mathcal{A}$ into account.

We introduce strategies as generators of decision functions. The notion of a most general strategy for some objective then corresponds to the inclusion of all decision functions enforcing the objective in the strategy. In our approach, strategies and controllers may label the observables in Obs with annotations from a set $\mathfrak{Ann}$, yielding the set of annotated observables $\mathfrak{Obs} = \mathit{Obs} \times \mathfrak{Ann}$. The annotations can be used to switch between phases. E.g., a most general strategy for an objective $\Diamond\Box\Phi$ (eventually always Φ) might operate in two phases. In the first phase, it ensures that it remains possible to eventually ensure $\Box\Phi$, while the second phase actually enforces $\Box\Phi$. The switch from the first to the second phase is realized using fairness assumptions on the annotations, see Example 3.

Definition 1 (Strategy). *A strategy is a tuple* $\mathfrak{S} = (\mathfrak{D}, \mathfrak{Fair}, \mathfrak{Ann})$ *where*

- $\mathfrak{D} : \mathfrak{Obs}^* \to 2^{2^{\mathfrak{Obs}}}$ *is a decision function template such that the following conditions (i)-(iv) hold for all annotated observations* $\sigma' \in \mathfrak{Obs}^*$*:*
  (i) $\mathfrak{D}(\sigma') \neq \varnothing$,
  (ii) $\langle \beta, \mathfrak{a}_1 \rangle, \langle \beta, \mathfrak{a}_2 \rangle \in \mathfrak{O}$ *implies* $\mathfrak{a}_1 = \mathfrak{a}_2$*, for all* $\mathfrak{O} \in \mathfrak{D}(\sigma')$ *and all* $\beta \in \mathit{Obs}$,
  (iii) $\mathfrak{O}|_{\mathit{Obs}}$ *is non-blocking for all* $\mathfrak{O} \in \mathfrak{D}(\sigma')$ *and*
  (iv) $\mathfrak{O}|_{\mathit{Obs}}$ *is passive for all* $\mathfrak{O} \in \mathfrak{D}(\sigma')$ *if* $\sigma'|_{\mathit{Obs}} \in \mathit{Obs}^* \mathit{Obs}_{\#}$.
- $\mathfrak{Fair} = \{\mathfrak{F}_1, \ldots, \mathfrak{F}_\ell\}$ *is a fairness condition consisting of finitely many subsets* $\mathfrak{F}_j$ *of* $\mathfrak{Obs}^* \times 2^{\mathfrak{Obs}}$ *such that* $\mathfrak{O} \in \mathfrak{D}(\sigma')$ *for all* $(\sigma', \mathfrak{O}) \in \mathfrak{F}_1 \cup \ldots \cup \mathfrak{F}_\ell$.
- $\mathfrak{Ann}$ *is a finite set of annotations.*

For $\mathfrak{F} \in \mathfrak{Fair}$ *and* $\sigma' \in \mathfrak{Obs}^*$*, we write* $\mathfrak{F}(\sigma')$ *for the set* $\{\mathfrak{O} \in 2^{\mathfrak{Obs}} : (\sigma', \mathfrak{O}) \in \mathfrak{F}\}$. *A function* $\mathfrak{d}' : \mathfrak{Obs}^* \to 2^{\mathfrak{Obs}}$ *is called an* annotated instance *of* $\mathfrak{S}$ *if* $\mathfrak{d}'$ *can be generated by the template* $\mathfrak{D}$ *in a fair way, i.e., if conditions (I1) and (I2) hold:*

(I1) $\mathfrak{d}'(\sigma') \in \mathfrak{D}(\sigma')$ *for all* $\mathfrak{d}'$*-schedulable annotated observations* $\sigma' \in \mathfrak{Obs}^*$
(I2) $\mathfrak{d}'$ *respects* $\mathfrak{Fair}$*, i.e., for each infinite* $\mathfrak{d}'$*-schedulable annotated observation* $\sigma' = \beta'_1 \beta'_2 \beta'_3 \ldots \in \mathfrak{Obs}^\omega$ *and for each* $\mathfrak{F} \in \mathfrak{Fair}$*, either (I2.1) or (I2.2) holds:*
  (I2.1) *There are only finitely many positions* $i \geqslant 1$ *where* $\mathfrak{F}(\beta'_1 \ldots \beta'_i) \neq \varnothing$.
  (I2.2) *There are infinitely many* $i \geqslant 1$ *with* $\mathfrak{d}'(\beta'_1 \ldots \beta'_i) \in \mathfrak{F}(\beta'_1 \ldots \beta'_i)$.

A decision function $\hat{\mathfrak{d}} : \mathit{Obs}^* \to 2^{\mathit{Obs}}$ *is called a* plain instance *of a strategy* $\mathfrak{S}$ *if there exists an annotated instance* $\mathfrak{d}' : \mathfrak{Obs}^* \to 2^{\mathfrak{Obs}}$ *of* $\mathfrak{S}$ *and an annotation function* $\mathfrak{ann}$ *such that for all* $\sigma \in \mathit{Obs}^*$

$$\mathfrak{d}'(\mathfrak{ann}^*(\sigma)) = \{\langle \beta, \mathfrak{ann}(\sigma, \beta) \rangle \in \mathfrak{Obs} : \beta \in \hat{\mathfrak{d}}(\sigma)\}$$

A decision function $\mathfrak{d} : Obs^* \to 2^{Obs}$ *is said to be an* $\mathcal{A}$*-instance, or briefly* instance*, of* $\mathfrak{S}$*, if there exists a plain instance* $\hat{\mathfrak{d}} : Obs^* \to 2^{Obs}$ *of* $\mathfrak{S}$ *such that for each* $\mathfrak{d}$*-schedulable observation* $\sigma \in Obs^*$ *and each observable* $\beta \in Obs$*:*

$$\textit{If } \sigma\beta \textit{ is } \mathcal{A}\textit{-schedulable then } \beta \in \mathfrak{d}(\sigma) \textit{ iff } \beta \in \hat{\mathfrak{d}}(\sigma).$$

An execution in $\mathcal{A}$ is called an $\mathfrak{S}$-*execution* if it is a $\mathfrak{d}$-execution for some instance $\mathfrak{d}$ of $\mathfrak{S}$. $\mathfrak{S}$-paths are defined accordingly. $\mathfrak{S}$ is called *admissible* if all $\mathfrak{S}$-paths are paths in $\mathcal{A}$, i.e., if all instances of $\mathfrak{S}$ are admissible. A decision function $\mathfrak{d}$ *enforces* objective $\Phi \subseteq Q \times (Act \times Q)^\infty$ if $\mathfrak{d}$ is admissible and all $\mathfrak{d}$-paths π satisfy Φ, i.e., $\pi \in \Phi$. A strategy $\mathfrak{S}$ enforces Φ if all instances $\mathfrak{d}$ of $\mathfrak{S}$ enforce Φ. An objective Φ is called *enforceable* if there is a decision function/strategy that enforces Φ.

Example 1. Reconsider the automaton $\mathcal{A}$ in Fig. 1a, with visible and controllable (non-suspend) actions α and β and objective $\Diamond q_1$ ("eventually q_1"). We identify the visible actions and their observables. A strategy that enforces $\Diamond q_1$ is $\mathfrak{S}_1 = (\mathfrak{D}, \mathfrak{Fair})$[1], with $\mathfrak{D}(\sigma) = \big\{\{\beta\}\big\}$ for all $\sigma \in Obs^*$ and $\mathfrak{Fair} = \varnothing$. The instances $\mathfrak{d}$ of $\mathfrak{S}_1$ are the decision functions $\mathfrak{d}$ with $\mathfrak{d}(\beta^*) = \{\beta\}$. Observations σ that are not of the form β^* are not $\mathfrak{S}_1$-schedulable and thus irrelevant, the instances can offer any subset of *Obs* for those σ, i.e., $\mathfrak{d}(\sigma) \subseteq \{\alpha, \beta\}$ for $\sigma \notin \beta^*$.
Another strategy enforcing $\Diamond q_1$ is $\mathfrak{S}_2 = (\mathfrak{D}, \mathfrak{Fair})$ with $\mathfrak{D}(\sigma) = \big\{\{\alpha\}, \{\beta\}, \{\alpha, \beta\}\big\}$ for all observations $\sigma \in Obs^*$ and $\mathfrak{Fair} = \{\mathfrak{F}\}$ with $\mathfrak{F} = \{(\alpha^n, \{\beta\}) : n \geqslant 0\}$. Note that $\varnothing \notin \mathfrak{D}(\sigma)$ as this would violate admissibility. Due to $\mathfrak{Fair}$, all instances of $\mathfrak{S}$ have to eventually offer $\{\beta\}$, leading to q_1. The instances of $\mathfrak{S}_2$ are the decision functions $\mathfrak{d}$ such that $\mathfrak{d}(\sigma) \in \big\{\{\alpha\}, \{\beta\}, \{\alpha, \beta\}\big\}$ for all $\mathfrak{d}$-schedulable observations σ and $\mathfrak{d}(\alpha^k) = \{\beta\}$ for some $k \in \mathbb{N}$. Again, for the irrelevant non-schedulable observations, the instances may choose any subset of *Obs*. Clearly, all instances of $\mathfrak{S}_1$ are instances of $\mathfrak{S}_2$. In fact, *all* the decision functions that enforce $\Diamond q_1$ are instances of $\mathfrak{S}_2$, making it a *most general* strategy enforcing $\Diamond q_1$:

Definition 2 (Most general strategy). *A strategy* $\mathfrak{S}$ *is called a* most general strategy *enforcing objective* Φ *iff* $\mathfrak{S}$ *enforces* Φ *and each decision function* $\mathfrak{d}$ *that enforces* Φ *is an instance of* $\mathfrak{S}$*.*

A strategy $\mathfrak{S}$ enforcing Φ is a most general strategy enforcing Φ iff for each strategy $\mathfrak{S}'$ that also enforces Φ, all instances of $\mathfrak{S}'$ are instances of $\mathfrak{S}$. A most general strategy enforcing Φ generates all decisions functions that enforce Φ, in particular all decision functions that enforce $\Phi \wedge \Psi$ for an arbitrary objective Ψ.

Controllers are finite-state machines that realize finite-memory strategies:

Definition 3 (Controller).
A controller is a tuple $\mathfrak{C} = (\mathfrak{M}, \mathfrak{m}_0, \Delta, \mu, \mathfrak{fair}, \mathfrak{Ann})$ *consisting of*

- *a finite set* $\mathfrak{M}$ *of modes and an initial mode* $\mathfrak{m}_0 \in \mathfrak{M}$*,*
- *a decision function template* $\Delta : \mathfrak{M} \to 2^{2^{\mathfrak{Obs}}}$ *and*

[1] We simply write $(\mathfrak{D}, \mathfrak{Fair})$ if no annotations are needed. I.e., $(\mathfrak{D}, \mathfrak{Fair})$ stands for the triple $(\mathfrak{D}, \mathfrak{Fair}, \mathfrak{Ann})$ where $\mathfrak{Ann}$ is a singleton $\{\mathfrak{a}\}$ and each annotated observation $\beta' = \langle \beta, \mathfrak{a} \rangle$ is identified with β. Controllers can be treated in a similar fashion.

- *a partial next-mode function* $\mu : \mathfrak{M} \times \mathfrak{Obs} \to \mathfrak{M}$,
- *a fairness condition* $\mathfrak{fair} = \{\mathfrak{F}_1, \dots, \mathfrak{F}_\ell\}$ *consisting of finitely many subsets* $\mathfrak{F}_j$ *of* $\mathfrak{M} \times 2^{\mathfrak{Obs}}$,
- *and a finite set of annotations* $\mathfrak{Ann}$.

We extend μ *to the partial function* $\mu^* : \mathfrak{M} \times \mathfrak{Obs}^* \to \mathfrak{M}$ *by* $\mu^*(\mathfrak{m}, \varepsilon) = \mathfrak{m}$ *and* $\mu^*(\mathfrak{m}, \beta'_1\beta'_2 \dots \beta'_n) = \mu^*\big(\mu(\mathfrak{m}, \beta'_1), \beta'_2 \dots \beta'_n\big)$. *For all modes* $\mathfrak{m} \in \mathfrak{M}$, *we require that all of the following conditions (i)-(v) hold:*

(i) $\Delta(\mathfrak{m}) \neq \varnothing$ *and* $\mathfrak{O}|_{Obs}$ *is non-blocking for all* $\mathfrak{O} \in \Delta(\mathfrak{m})$,
(ii) $\langle \beta, \mathfrak{a}_1 \rangle, \langle \beta, \mathfrak{a}_2 \rangle \in \mathfrak{O}$ *implies* $\mathfrak{a}_1 = \mathfrak{a}_2$, *for all* $\mathfrak{O} \in \Delta(\mathfrak{m})$ *and all* $\beta \in Obs$,
(iii) $\mu(\mathfrak{m}, \beta') \neq \bot$ *if* $\beta' \in \mathfrak{O}$ *for some* $\mathfrak{O} \in \Delta(\mathfrak{m})$,
(iv) $\mathfrak{O}|_{Obs}$ *is passive for all* $\mathfrak{O} \in \Delta(\mu(\mathfrak{m}, \beta'))$ *if* $\beta'|_{Obs} \in Obs_{\#}$ *and* $\beta' \in \mathfrak{O}$ *for some* $\mathfrak{O} \in \Delta(\mathfrak{m})$,
(v) $\mathfrak{O} \in \Delta(\mathfrak{m})$ *for all* $(\mathfrak{m}, \mathfrak{O}) \in \mathfrak{F}_1 \cup \ldots \cup \mathfrak{F}_\ell$.

Formally, the strategy realized by controller $\mathfrak{C} = (\mathfrak{M}, \mathfrak{m}_0, \Delta, \mu, \mathfrak{fair}, \mathfrak{Ann})$ with $\mathfrak{fair} = \{\mathfrak{F}_1, \dots, \mathfrak{F}_\ell\}$ is $\mathfrak{S}_{\mathfrak{C}} = (\mathfrak{D}, \mathfrak{Fair}, \mathfrak{Ann})$ where

$$\mathfrak{D}(\sigma') = \Delta(\mu^*(\mathfrak{m}_0, \sigma')) \text{ and } \mathfrak{Fair} = \{\mathfrak{F}'_1, \dots, \mathfrak{F}'_\ell\}$$

with $\mathfrak{F}'_j = \{(\sigma', \mathfrak{O}) \subseteq \mathfrak{Obs}^* \times 2^{\mathfrak{Obs}} : \mu^*(\mathfrak{m}_0, \sigma') = \mathfrak{m} \text{ and } (\mathfrak{m}, \mathfrak{O}) \in \mathfrak{F}_j\}$ for $1 \leq j \leq \ell$. The definition of $\mathfrak{D}(\sigma')$ supposes $\mu^*(\mathfrak{m}_0, \sigma') \in \mathfrak{M}$. If $\mu^*(\mathfrak{m}_0, \sigma') = \bot$ then σ' is not $\mathfrak{D}$-schedulable and the value of $\mathfrak{D}(\sigma')$ is irrelevant.

A controller $\mathfrak{C}$ induces an automaton $\mathcal{A}_{\mathfrak{C}} = (\mathfrak{M}, Obs \times \mathfrak{Ann}, \longrightarrow_{\mathfrak{C}}, \{\mathfrak{m}_0\})$ in a natural way, with $\longrightarrow_{\mathfrak{C}}$ derived as follows:

$$\mathfrak{m} \xrightarrow{\beta'}_{\mathfrak{C}} \mathfrak{m}' \text{ iff } \beta' \in \mathfrak{O} \text{ for some } \mathfrak{O} \in \Delta(\mathfrak{m}) \text{ and } \mathfrak{m}' = \mu(\mathfrak{m}, \beta').$$

In the sequel, we identify controller $\mathfrak{C}$ with its induced automaton $\mathcal{A}_{\mathfrak{C}}$.

Example 2. Consider the most general strategy $\mathfrak{S}_2$ enforcing $\Diamond q_1$ from Example 1. This strategy can be realized by a controller with modes $\mathfrak{m}_0$ and $\mathfrak{m}_1$ and the induced automaton shown in Fig. 2a with $\mathfrak{fair} = \{\mathfrak{F}\}$ and $\mathfrak{F} = \{(\mathfrak{m}_0, \{\beta\})\}$.
As another example, consider the system given by the automaton in Fig. 1c and the safety objective "whenever γ, immediately afterwards α", where actions α, β are controllable, γ is visible but uncontrollable and suspend action #. We again identify the actions and their observables. A most general strategy $\mathfrak{S} = (\mathfrak{D}, \mathfrak{Fair})$ for this objective has $\mathfrak{D}$ given by

$$\begin{aligned}&\mathfrak{D}(\varepsilon) = \mathfrak{D}(\sigma\alpha) = \mathfrak{D}(\sigma\beta) = \{O : \{\gamma\} \subseteq O \subseteq \{\#, \alpha, \beta, \gamma\}\},\\ &\mathfrak{D}(\sigma\gamma) = \{\{\alpha, \gamma\}\} \text{ and } \mathfrak{D}(\sigma\#) = \{\{\gamma\}\} \text{ for } \sigma \in Obs^*\end{aligned}$$

and $\mathfrak{Fair} = \varnothing$. The uncontrollable action γ is always offered. When γ occurs, the strategy disallows β and #, forcing α to occur. After #, observation of γ wakes the suspended controllable component. $\mathfrak{S}$ is realized by the controller with induced automaton shown in Fig. 2b and $\mathfrak{fair} = \varnothing$. ■

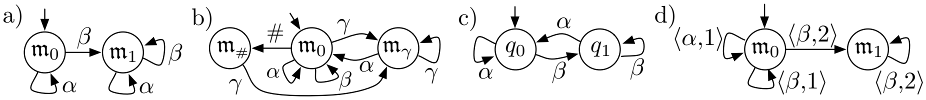


Fig. 2. Examples for most general strategies and controllers

Example 3. To illustrate the use of annotations, consider the system automaton $\mathcal{A}$ in Fig. 2c with controllable (non-suspend) actions α, β and the objective $\varPhi = \Diamond\Box q_1$ (eventually always q_1), i.e., $\varPhi = (Q \times \mathit{Obs})^* \times (\{q_1\} \times \mathit{Obs})^\omega$. A controller realizing a most general strategy enforcing $\varPhi$ uses annotations $\mathfrak{Ann} = \{1,2\}$. In a first phase, $\varDelta(\mathfrak{m}_0) = \big\{\{\langle\alpha,1\rangle\},\{\langle\beta,1\rangle\},\{\langle\beta,2\rangle\},\{\langle\alpha,1\rangle,\langle\beta,1\rangle\}\big\}$, in a second phase $\varDelta(\mathfrak{m}_1) = \big\{\langle\beta,2\rangle\big\}$. The induced automaton is shown in Fig. 2d. The fairness condition $\mathfrak{fair} = \{\mathfrak{F}\}$ with $\mathfrak{F} = \{(\mathfrak{m}_0, \{\langle\beta,2\rangle\})\}$ ensures that eventually $\{\langle\beta,2\rangle\}$ is scheduled and the switch from the first to the second phase occurs. From that point on, only β is allowed and only q_1 is visited. ■

The concept of annotations is crucial to capture most general strategies:

Lemma 1. *There is no most general strategy $\mathfrak{S} = (\mathfrak{D}, \mathfrak{Fair}, \mathfrak{Ann})$ for $\mathcal{A}$ from Example 3 that does not utilize annotations, i.e., where $\mathfrak{Ann}$ is a singleton set, and that enforces $\varPhi = \Diamond\Box q_1$.*

3 Compositionality

To allow the compositional treatment of cascades of objectives $\varPhi_1, \varPhi_2, \ldots, \varPhi_k$ for a system $\mathcal{A}$ in an online manner, we first construct a controller $\mathfrak{C}_1$ for system $\mathcal{A}$ realizing a most general strategy enforcing $\varPhi_1$, i.e., such that the composition of $\mathfrak{C}_1$ and $\mathcal{A}$ satisfies $\varPhi_1$, $\mathfrak{C}_1 \bowtie \mathcal{A} \models \varPhi_1$. We then construct a controller $\mathfrak{C}_2$ realizing a most general strategy enforcing objective $\varPhi_2$ for the system $\mathfrak{C}_1 \bowtie \mathcal{A}$, a controller enforcing $\varPhi_3$ for the system $\mathfrak{C}_2 \bowtie (\mathfrak{C}_1 \bowtie \mathcal{A})$, and so on, such that

$$\mathfrak{C}_k \bowtie \ldots \bowtie \mathfrak{C}_2 \bowtie \mathfrak{C}_1 \bowtie \mathcal{A} \quad \models \quad \varPhi_1 \wedge \varPhi_2 \wedge \ldots \wedge \varPhi_k$$

if $\varPhi$ is enforceable in $\mathcal{A}$. For this approach, it is crucial that the strategies realized by the controllers $\mathfrak{C}_i$ are most general, as they must not prematurely rule out any of the decision functions that enforce $\varPhi_i$ which may be necessary to enforce the conjunction $\varPhi_i \wedge \varPhi_{i+1}$ or with other subsequent objectives.

The controlled system, i.e., the composition $\mathfrak{C} \bowtie \mathcal{A}$ of the system $\mathcal{A}$ and the controller automaton $\mathfrak{C}$, is obtained by a product construction where the two automata are synchronized via the observables. As noted in the previous section, a controller $\mathfrak{C} = (\mathfrak{M}, \mathfrak{m}_0, \varDelta, \mu, \mathfrak{fair}, \mathfrak{Ann})$ induces an automaton $\mathcal{A}_\mathfrak{C} = (\mathfrak{M}, \mathit{Obs} \times \mathfrak{Ann}, \longrightarrow_\mathfrak{C}, \mathfrak{m}_0)$ which we identify with $\mathfrak{C}$, with an action alphabet consisting of the annotated observables. Let $\mathcal{A} = (Q, \mathit{Act}, \longrightarrow_\mathcal{A}, Q_0)$ be the automaton for the system.

The composed system is defined as $\mathfrak{C} \bowtie \mathcal{A} = (\mathfrak{M} \times Q, \mathit{Act}', \longrightarrow_\bowtie, \{\mathfrak{m}_0\} \times Q_0)$, with action alphabet Act' obtained from Act by annotating the visible actions, i.e., $Act' = Act'_{\mathrm{vis}} \cup (Act \setminus Act_{\mathrm{vis}})$ and $Act'_{\mathrm{vis}} = Act_{\mathrm{vis}} \times \mathfrak{Ann}$. The controllable and suspend actions of the composed system are the corresponding annotated actions, i.e., $Act'_{\mathrm{ctr}} = Act_{\mathrm{ctr}} \times \mathfrak{Ann}$ and $Act'_{\#} = Act_{\#} \times \mathfrak{Ann}$. The transition relation $\longrightarrow_{\bowtie}$ is given by

$$\frac{q \xrightarrow{\alpha}_{\mathcal{A}} q' \ \wedge \ \alpha \in Act_{\mathrm{vis}} \ \wedge \ \mathfrak{m} \xrightarrow{\langle \beta, \mathfrak{a} \rangle}_{\mathfrak{C}} \mathfrak{m}' \ \wedge \ obs(\alpha) = \beta}{\langle \mathfrak{m}, q \rangle \xrightarrow{\langle \alpha, \mathfrak{a} \rangle}_{\bowtie} \langle \mathfrak{m}', q' \rangle} \qquad \frac{q \xrightarrow{\alpha}_{\mathcal{A}} q' \ \wedge \ \alpha \notin Act_{\mathrm{vis}}}{\langle \mathfrak{m}, q \rangle \xrightarrow{\alpha} \langle \mathfrak{m}, q' \rangle}$$

The rule on the left synchronizes a visible action of $\mathcal{A}$ with the corresponding observable allowed by the controller $\mathfrak{C}$. Note that the controller can not inhibit uncontrollable actions, as it is required to be non-blocking (condition (i) in Def. 3), while the rule on the right ensures that invisible actions occur independent of the controller. The observables of $\mathfrak{C} \bowtie \mathcal{A}$ then consist of the annotated observables $\mathfrak{Obs} = Obs \times \mathfrak{Ann}$ of $\mathcal{A}$, with $obs' : Act'_{\mathrm{vis}} \to \mathfrak{Obs}$ given by $obs'(\langle \alpha, \mathfrak{a} \rangle) = \langle obs(\alpha), \mathfrak{a} \rangle$. The partition $\mathcal{U}_{\mathfrak{Obs}} = \big\{\{\langle obs(\alpha), \mathfrak{a} \rangle : \mathfrak{a} \in \mathfrak{Ann}\} : \alpha \in Act_{\mathrm{vis}} \setminus Act_{\mathrm{ctr}}\big\}$ groups the annotated observables of the uncontrollable actions such that $\mathfrak{O} \in 2^{\mathfrak{Obs}}$ is non-blocking if there is at least one annotated observable $\beta' \in \mathfrak{O}$ for each uncontrollable action.

For an execution $\pi_{\mathfrak{C}}$ in $\mathfrak{C} \bowtie \mathcal{A}$, let $\pi_{\mathfrak{C}}|_{\mathcal{A}}$ be the corresponding execution in $\mathcal{A}$ that is obtained by stripping the controller modes in all states and the annotations in the actions. $\pi_{\mathfrak{C}}$ is said to satisfy objective $\Phi \subseteq Q \times (Act \times Q)^{\infty}$, denoted $\pi_{\mathfrak{C}} \models \Phi$, if $\pi_{\mathfrak{C}}|_{\mathcal{A}} \in \Phi$.

Fairness. To be able to capture the fairness imposed by the controller on the controlled system at the automaton level, we augment our concept of automata with a suitable fairness condition, syntactically similar to the fairness condition used for strategies: A fairness condition $\mathfrak{Fair}[\mathcal{A}]$ for an automaton $\mathcal{A}$ is a finite set $\mathfrak{Fair}[\mathcal{A}] = \{\mathfrak{F}_1, \ldots, \mathfrak{F}_\ell\}$ consisting of subsets $\mathfrak{F}$ of $Obs^* \times 2^{Obs}$ such that O is non-blocking for all $\mathfrak{F} \in \mathfrak{Fair}[\mathcal{A}]$ and $(\sigma, O) \in \mathfrak{F}$. We require $\#$-*admissibility* of $\mathfrak{Fair}[\mathcal{A}]$, i.e., for all observations $\sigma \in Obs^*$ and $\mathfrak{F} \in \mathfrak{Fair}[\mathcal{A}]$:

(1) For all $\beta \in Obs_{\#}$ and $O \in \mathfrak{F}(\sigma\beta)$, O is passive.
(2) If $\sigma \notin Obs^* Obs_{\#}$ and π an initial execution in $\mathcal{A}$ with $obs(\pi) = \sigma$ such that $last(\pi)$ is non-terminal then $Act_{\mathcal{A}}(last(\pi), O) \neq \varnothing$ for all $O \in \mathfrak{F}(\sigma)$.

As before, $\mathfrak{F}(\sigma)$ denotes $\{O : (\sigma, O) \in \mathfrak{F}\}$. An observation $\sigma = \beta_1\beta_2 \ldots \in Obs^{\infty}$ is called $\mathfrak{Fair}[\mathcal{A}]$-schedulable if σ is finite or σ is infinite and, for each $\mathfrak{F} \in \mathfrak{Fair}[\mathcal{A}]$, either the number of positions $i \geqslant 1$ with $\mathfrak{F}(\beta_1\beta_2 \ldots \beta_i) \neq \varnothing$ is finite or there are infinitely many positions i such that $\beta_{i+1} \in O$ for some $O \in \mathfrak{F}(\beta_1\beta_2 \ldots \beta_i)$.

We adapt the concepts of paths and admissibility to take fairness into account: An execution π is a path in $\mathcal{A}$ with $\mathfrak{Fair}[\mathcal{A}]$ if it is a path in $\mathcal{A}$ and if $obs(\pi)$ is $\mathfrak{Fair}[\mathcal{A}]$-schedulable. A decision function $\mathfrak{d}$ is admissible for $\mathcal{A}$ with $\mathfrak{Fair}[\mathcal{A}]$ if it is admissible for $\mathcal{A}$ and if all $\mathfrak{d}$-schedulable observations are $\mathfrak{Fair}[\mathcal{A}]$-schedulable, i.e., all $\mathfrak{d}$-paths in $\mathcal{A}$ are fair according to $\mathfrak{Fair}[\mathcal{A}]$. The notion of a decision function or strategy enforcing an objective Ψ for $\mathcal{A}$ with $\mathfrak{Fair}[\mathcal{A}]$ requires admissibility and thus also takes fairness into account.

Let $\mathcal{A}$ be a system automaton with fairness condition $\mathfrak{Fair}[\mathcal{A}]$ (with observables Obs), let $\mathfrak{C}$ be a controller realizing an admissible strategy and let $\mathfrak{Fair}$

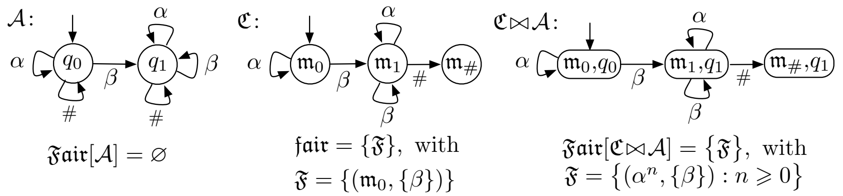

$\mathcal{A}$:
q_0
q_1
α
β
$\#$
$\mathfrak{Fair}[\mathcal{A}] = \varnothing$
$\mathfrak{C}$:
$\mathfrak{m}_0$
$\mathfrak{m}_1$
$\mathfrak{m}_\#$
$\mathfrak{fair} = \{\mathfrak{F}\}$, with
$\mathfrak{F} = \{(\mathfrak{m}_0, \{\beta\})\}$
$\mathfrak{C} \bowtie \mathcal{A}$:
$\mathfrak{m}_0, q_0$
$\mathfrak{m}_1, q_1$
$\mathfrak{m}_\#, q_1$
$\mathfrak{Fair}[\mathfrak{C} \bowtie \mathcal{A}] = \{\mathfrak{F}\}$, with
$\mathfrak{F} = \{(\alpha^n, \{\beta\}) : n \geqslant 0\}$

Fig. 3. Automaton $\mathcal{A}$, controller $\mathfrak{C}$ for objective $\Diamond q_1$ and product automaton $\mathfrak{C} \bowtie \mathcal{A}$

(with observables $\mathfrak{Obs}$) be the fairness condition of the strategy $\mathfrak{S}_{\mathfrak{C}}$ realized by the controller. The fairness condition $\mathfrak{Fair}[\mathfrak{C} \bowtie \mathcal{A}] = \mathfrak{Fair} \cup \mathfrak{Fair}[\mathcal{A}]$ for the controlled system is then the conjunction of the fairness conditions, with $\mathfrak{Fair}[\mathcal{A}]$ appropriately lifted from *Obs* to $\mathfrak{Obs}$. Note that, as $\mathfrak{C}$ is admissible and thus takes $\mathfrak{Fair}[\mathcal{A}]$ into account, the fairness $\mathfrak{Fair}$ of the controller subsumes $\mathfrak{Fair}[\mathcal{A}]$ and it is sufficient to consider $\mathfrak{Fair}[\mathfrak{C} \bowtie \mathcal{A}] = \mathfrak{Fair}$ only.

Lemma 2 (Soundness of $\mathfrak{C} \bowtie \mathcal{A}$). *Let $\mathfrak{C}$ be a controller such that its induced strategy $\mathfrak{S}_{\mathfrak{C}}$ is admissible. Then, for every initial $\mathfrak{S}_{\mathfrak{C}}$-path π in $\mathcal{A}$, there exists an initial path $\pi_{\mathfrak{C}}$ in $\mathfrak{C} \bowtie \mathcal{A}$ such that $\pi_{\mathfrak{C}}|_{\mathcal{A}} = \pi$, and vice versa.*

Note that decision functions $\mathfrak{d}'$ for $\mathfrak{C} \bowtie \mathcal{A}$ can be regarded as strategies for $\mathcal{A}$ with annotations $\mathfrak{Ann}$, where $\mathfrak{Ann}$ is the set of annotations of $\mathfrak{C}$. The notion of controllers realizing most general strategies, capturing all the decision functions that enforce a given objective, allows the compositional treatment of conjunctive objectives.

Theorem 1 (Compositionality). *Let Φ and Ψ be arbitrary objectives and let $\mathfrak{C}$ be a controller such that its induced strategy $\mathfrak{S}_\Phi = \mathfrak{S}_{\mathfrak{C}}$ is most general enforcing Φ for $\mathcal{A}$.*

(1) For every decision function $\mathfrak{d}$ enforcing $\Phi \wedge \Psi$ for $\mathcal{A}$, there is a decision function $\mathfrak{d}'$ for $\mathfrak{C} \bowtie \mathcal{A}$ that enforces Ψ for $\mathfrak{C} \bowtie \mathcal{A}$ such that $\mathfrak{d}$ is an instance of $\mathfrak{d}'$, when $\mathfrak{d}'$ is viewed as a strategy.
(2) Let $\mathfrak{S}_\Psi$ be a most general strategy enforcing Ψ for $\mathfrak{C} \bowtie \mathcal{A}$. Then there exists a strategy $\mathfrak{S}_{\Phi \wedge \Psi}$ for $\mathcal{A}$ that is most general enforcing $\Phi \wedge \Psi$.
(3) Let $\mathfrak{C}_\Psi$ be a controller such that its induced strategy $\mathfrak{S}_\Psi = \mathfrak{S}_{\mathfrak{C}_\Psi}$ is most general enforcing Ψ for $\mathfrak{C} \bowtie \mathcal{A}$. Then there exists a controller $\mathfrak{C}_{\Phi \wedge \Psi}$ for $\mathcal{A}$ such that its induced strategy is most general enforcing $\Phi \wedge \Psi$.

As a direct consequence of (1) in Theorem 1, if $\Phi \wedge \Psi$ is enforceable in $\mathcal{A}$, then Ψ is enforceable in $\mathfrak{C} \bowtie \mathcal{A}$.

Example 4. Consider the automaton $\mathcal{A}$ in Fig. 3, with controllable actions α and β and suspend action $\#$. We identify actions and their observables. Controller $\mathfrak{C}$ in Fig. 3 with $\Delta(\mathfrak{m}_0) = \big\{\{\alpha\}, \{\beta\}, \{\alpha, \beta\}\big\}$, $\varnothing \neq \Delta(\mathfrak{m}_1) \subseteq Obs$ and $\Delta(\mathfrak{m}_\#) = \{\varnothing\}$ realizes a most general strategy for the reachability objective $\Phi = \Diamond q_1$. The composition $\mathfrak{C} \bowtie \mathcal{A}$ depicted in Fig. 3 with the fairness condition derived from $\mathfrak{C}$ then serves as the system automaton for a second objective Ψ, to be enforced in conjunction with $\varPhi$. For an objective like $\varPsi = \Box\neg q_1$ ("never q_1"), where $\varPsi$ is enforceable for $\mathcal{A}$ on its own, but $\varPsi \wedge \varPhi$ is not enforceable, $\varPsi$ is not enforceable for $\mathfrak{C} \bowtie \mathcal{A}$: A decision function enforcing $\Box\neg q_1$ for $\mathfrak{C} \bowtie \mathcal{A}$ would have to either force termination before β is scheduled, which would violate admissibility as the suspend action is not available in state $\langle \mathfrak{m}_0, q_0 \rangle$, or schedule $\{\alpha\}$ continuously, which likewise violates admissibility as the fairness condition $\mathfrak{Fair}[\mathfrak{C} \bowtie \mathcal{A}]$ would be violated. Objectives that can be enforced in conjunction with $\varPhi$ can be enforced in $\mathfrak{C} \bowtie \mathcal{A}$. E.g., to enforce the objective that all executions have to be finite, a decision function could first schedule $\{\beta\}$ and then force admissible termination by scheduling $\{\#\}$. ■

4 Game-Based Controller Synthesis

We now provide a game-based characterization of the controller synthesis problem and adapt known algorithms for observation-based games [14,7] with reachability and invariance objectives using a powerset construction and fixed point computations to our setting. The required modifications are non-trivial since we have to extract controllers realizing most general strategies rather than just computing the winning regions. Among other technical differences to previous approaches we have to deal with fairness assumptions and ensure admissibility. In the sequel, we provide an overview of the game based construction of controllers realizing most general strategies for a given objective.

Objectives. We treat here the case where the automaton $\mathcal{A}$ arises from the product of an automaton $\mathcal{A}_{\text{ctr}}$ for the controllable component and an automaton $\mathcal{A}_{\text{env}}$ for the environment and the automata for previously applied controllers. We can project from the states Q of $\mathcal{A}$ to the set of (local) states Q_{ctr} of $\mathcal{A}_{\text{ctr}}$ and assume that the local state of $\mathcal{A}_{\text{ctr}}$ does only change in $\mathcal{A}$ on visible actions. We treat here linear-time objectives $\varPhi \subseteq Q_{\text{ctr}} \times (\mathit{Obs} \times Q_{\text{ctr}})^{\infty}$ that just refer to the observables and (local) states of $\mathcal{A}_{\text{ctr}}$. A path π in $\mathcal{A}$ satisfies $\varPhi$ iff the sequence obtained by removing invisible actions, replacing the visible actions by their observables and projecting the states from Q to Q_{ctr} is contained in $\varPhi$.

Observation-based game. In essence, we use the automaton $\mathcal{A}$, slightly modified to reflect the effect of the suspend signal, as the game arena of a turn-based two-player game where the controllable component is viewed as one player (P_1) and the environment as its opponent (P_2). The states of $\mathcal{A}$ serve as game configurations. The initial game configuration of a play is chosen by P_2 from Q_0. Each round of a play consists of two steps. First, P_1 chooses some non-blocking $O \in 2^{\mathit{Obs}}$. In case the previous observable was the suspend signal, it has to choose a passive O. Second, P_2 selects a transition from the current game configuration q to some state q' with an O-compliant action $\alpha \in \mathit{Act}(q, O)$. State q' becomes the game configuration for the next round. The play terminates if $\mathit{Act}(q, O) = \varnothing$. The game is viewed to be a partial-information game for P_1, i.e., it has to choose its $O \in 2^{\mathit{Obs}}$ on the basis of the observation generated by the actions that have been chosen in the previous rounds. Each play generates an initial execution π in $\mathcal{A}$. Objective $\varPhi$ is viewed as winning criterion for P_1, winning a play if the execution π is a path in $\mathcal{A}$ and $\pi \models \varPhi$.

Complete-information game. We adapt the well-known powerset construction [14] to turn this partial-information game into a turn-based two-player game with complete information for both players. By abstracting from the invisible actions and only regarding the observables, the vertices belonging to the controllable component in the game graph then consist of the set of all states consistent with the observation of the history. Special vertices deal with the possibility of termination, divergence (the environment chooses to only schedule invisible actions) and failure to ensure admissibility in some game configuration q. The initial vertex of the game, $[Q_0]_*$ is the set of initial states closed under invisible moves.

Reachability objectives without fairness. We first deal with the case of $\mathcal{A}$ without a fairness condition and consider reachability objectives $\varPhi = \Diamond F$ where with $F \subseteq Q_{\mathrm{ctr}}$. The goal is to design a controller realizing a most general strategy $\mathfrak{S}$ enforcing $\varPhi$, i.e., that ensures that at some point the local state of $\mathcal{A}_{\mathrm{ctr}}$ will be in F along all $\mathfrak{S}$-paths. We add a gadget to $\mathcal{A}$ to track whether a game configuration has been reached via an execution π with $\pi \models \varPhi$. The goal vertices $\mathcal{F}$ are then those sets of game configurations where each one has been reached via some $\pi \models \varPhi$, as well as the special vertices for termination and divergence that occur in a state that has been reached via an execution that satisfies $\varPhi$. We obtain the set of winning vertices $\mathrm{Win}(\Diamond F)$ using the standard fixed point characterization of reachability games with goal vertices $\mathcal{F}$, as well as winning regions $\mathcal{W}^{(i)}$ containing the vertices where the controllable component can enforce reaching $\mathcal{F}$ in at most i observations. If the initial vertex $[Q_0]_* \in \mathrm{Win}(\Diamond F)$, we conclude that $\Diamond F$ is enforceable for $\mathcal{A}$ and construct from these sets a controller enforcing $\Diamond F$ in a most general manner. The modes of the controller consist of the vertices in $\mathrm{Win}(\Diamond F)$, with initial mode $[Q_0]_*$. The decision function template allows all choices for O that ensure staying in $\mathrm{Win}(\Diamond F)$. The next-mode function tracks the successor mode for the observables. Crucially, the fairness condition of the controller ensures eventual progress from the modes in $\mathcal{W}^{(i+1)}$ to the modes in $\mathcal{W}^{(i)}$, by requiring fairness for those O in the decision function template that lead to modes that are "closer" to the goal.

Invariance objectives without fairness of the form $\varPhi = \Box I$, with $I \subseteq Q_{\mathrm{ctr}}$ are handled similarly. The set of safe vertices $\mathcal{I}$ are those vertices where the local state in $\mathcal{A}_{\mathrm{ctr}}$ of all contained game configurations is in I. We apply the standard fixed-point characterization for invariances and obtain the winning region $\mathrm{Win}(\Box I)$. If $[Q_0]_* \in \mathrm{Win}(\Box I)$, we extract a controller realizing a most general strategy enforcing $\Box I$ the same way as for reachability, except that no fairness condition is used.

Reachability, invariance with fairness. If $\mathcal{A}$ has a fairness condition $\mathfrak{Fair}[\mathcal{A}]$, we have to adapt our techniques. We require that each $\mathfrak{F} \in \mathfrak{Fair}[\mathcal{A}]$ satisfies the following conditions:

(F1) If $\mathfrak{F}(\sigma) \neq \varnothing$ then $\mathfrak{F}'(\sigma) = \varnothing$ for all $\sigma \in \mathit{Obs}^*$ and $\mathfrak{F}' \in \mathfrak{Fair}[\mathcal{A}] \setminus \{\mathfrak{F}\}$.

(F2) If $\pi_1, \pi_1', \pi_2, \pi_2'$ are initial executions in $\mathcal{A}$ such that $\mathit{last}(\pi_1) = \mathit{last}(\pi_1')$, $\mathit{last}(\pi_2) = \mathit{last}(\pi_2')$ and $\mathit{obs}(\pi_1') = \mathit{obs}(\pi_2')$ then $\mathfrak{F}(\mathit{obs}(\pi_1)) = \mathfrak{F}(\mathit{obs}(\pi_2))$.

(F3) For each $\mathfrak{Fair}[\mathcal{A}]$-schedulable observation $\mathit{obs}(\pi) = \beta_1\beta_2\ldots$ there exists a positive integer i_0 such that $\mathfrak{F}(\beta_1 \ldots \beta_i) = \varnothing$ for all $i \geqslant i_0$.

(F2) holds for the fairness assumptions for any controller since they have been defined mode-wise and the controller behaves deterministically for a given observation. (F1) and (F2) allow us to relate game vertices to the relevant fairness conditions. (F3) asserts that the fairness condition is "finitary". For all the controllers constructed in this section, the fairness condition $\mathfrak{Fair}$ induced by the controller enjoys (F1)-(F3) and, as the fairness condition $\mathfrak{Fair}$ of the controller subsumes the fairness $\mathfrak{Fair}[\mathcal{A}]$, it is thus sufficient to consider $\mathfrak{Fair}[\mathfrak{C} \bowtie \mathcal{A}] = \mathfrak{Fair}$, which then again satisfies (F1)-(F3).

For $\Phi = \Diamond F$ and automaton $\mathcal{A}$ with $\mathfrak{Fair}[\mathcal{A}]$, we construct a controller as before for $\mathcal{A}$ and Φ, which is then adapted to ensure $\mathfrak{Fair}[\mathcal{A}]$-fairness after $\mathcal{F}$ has been reached. For $\Phi = \Box I$, we first calculate $\mathrm{Win}(\Box I)$ as before, ignoring the fairness condition $\mathfrak{Fair}[\mathcal{A}]$. We then compute the set $\mathcal{T}$ of vertices in $\mathrm{Win}(\Box I)$ that have no associated fairness condition. Again we apply the techniques for invariances without fairness to obtain the set $\mathcal{X} = \mathrm{Win}(\Box \mathcal{T})$ of vertices where it is possible to enforce $\Box I$ while staying in $\mathcal{T}$. Finally, we calculate the set $\mathrm{Win}(I \ \mathrm{U}\ \mathcal{X})$ (where U denotes the standard until operator), by slightly modifying the fixpoint calculation for reachability. If $[Q_0]_* \in \mathrm{Win}(I \ \mathrm{U}\ \mathcal{X})$ then we construct a controller realizing a most general strategy enforcing $\Box I$ for $\mathcal{A}$ with $\mathfrak{Fair}[\mathcal{A}]$ operating in two phases marked by annotations. First, the controller ensures that the plays stay in $\mathrm{Win}(I \ \mathrm{U}\ \mathcal{X})$ but eventually, via a fairness condition on the annotations, switches to the second phase, where it ensures that the plays eventually reach $\mathcal{X}$ and forever stay in $\mathcal{X}$.

Regular safety and co-safety objectives, i.e., $\Phi = [[L]]I$ and $\Phi = \langle\langle L\rangle\rangle F$, where L is a regular language over the observables, $\langle\langle L\rangle\rangle F$ requires that F is reached via some observation in L and for $[[L]]I$ all states reachable with observations in L are in I, are then handled using a product automata approach and the controller construction for invariance and reachability.

Conjunctions of objectives. Combining the compositional approach to treating conjunctions of objectives (Theorem 1) with the game-based construction of controllers realizing most general strategies for regular safety and co-safety objectives in this section yields the following theorem:

Theorem 2. *Let $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_k$ be a conjunction of regular safety and co-safety objectives Φ_i. If Φ is enforceable for $\mathcal{A}$ then there exists a controller $\mathfrak{C}$ that realizes a most general strategy enforcing Φ for $\mathcal{A}$.*

As our algorithmic approach relies on a powerset construction to obtain a complete-information game, there can be an exponential blow-up in the size of the automaton under consideration. This is to be expected, as the problem of deciding whether there is a decision function enforcing an objective in such a partial-information setting is known to be EXPTIME-complete for reachability objectives [14,7] as well as for safety objectives [5]. The controller $\mathfrak{C}$ of Theorem 2 that realizes a most general strategy enforcing $\Phi_1 \wedge \ldots \wedge \Phi_k$ for reachability and invariance objectives Φ_i has at most $4^k \cdot 2^{|Q|}$ reachable modes, where $|Q|$ is the number of states of $\mathcal{A}$. The factor 4 per objective arises from a factor 2 for correctly treating the suspend actions and a factor 2 for the tracking of reachability or the two phases for invariance objectives with fairness. For regular (co-)safety objectives, the sizes of the DFA recognizing the regular languages L are additional factors. Due to the idempotence of the powerset construction and the fact that the controllers are deterministic and synchronize over the observables, the factor $2^{|Q|}$ is only incurred once.

5 Conclusion

Our main contribution is the presentation of a novel framework for strategies under partial observability that is adequate for compositional reasoning. For this purpose, we introduced the notion of most general strategies that generate all decision functions enforcing a given linear-time objective. Furthermore, we showed how to adapt the standard powerset construction to obtain finite-memory controllers realizing most general strategies for safety and co-safety objectives. We currently work on extending our results to generate controllers realizing most general strategies for the full ω-regular objectives.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. IEEE Special Issue on Hybrid Systems 88, 1011–1025 (2000)
3. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995)
4. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: From parity games to safety games. ITA 36(3), 261–275 (2002)
5. Berwanger, D., Doyen, L.: On the power of imperfect information. In: FSTTCS 2008. LIPIcs, vol. 2, pp. 73–82. Schloss Dagstuhl (2008)
6. Bouyer, P., Duflot, M., Markey, N., Renault, G.: Measuring permissivity in finite games. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 196–210. Springer, Heidelberg (2009)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.F.: Algorithms for omega-regular games with imperfect information. Logical Methods in Computer Science 3(3) (2007)
8. Filiot, E., Jin, N., Raskin, J.F.: Compositional Algorithms for LTL Synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010)

9. Kuijper, W., van de Pol, J.: Compositional control synthesis for partially observable systems. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 431–447. Springer, Heidelberg (2009)
10. Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open systems in reactive environments: Control and synthesis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 92–107. Springer, Heidelberg (2000)
11. Mohalik, S., Walukiewicz, I.: Distributed games. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 338–351. Springer, Heidelberg (2003)
12. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190. ACM, New York (1989)
13. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: LICS 1990, vol. 2, pp. 746–757 (1990)
14. Reif, J.H.: The complexity of two-player games of incomplete information. Journal of Computer and System Sciences 29(2), 274–301 (1984)
15. Vardi, M.Y.: An automata-theoretic approach to fair realizability and synthesis. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 267–278. Springer, Heidelberg (1995)
16. Wonham, W.M.: On the control of discrete-event systems. In: Three Decades of Mathematical System Theory. LNCIS, vol. 135, pp. 542–562. Springer, Heidelberg (1989)

Decidability of Branching Bisimulation on Normed Commutative Context-Free Processes*

Wojciech Czerwiński, Piotr Hofman, and Sławomir Lasota

Institute of Informatics, University of Warsaw
{wczerwin,ph209519,sl}@mimuw.edu.pl

Abstract. We investigate normed commutative context-free processes (Basic Parallel Processes). We show that branching bisimilarity admits the *small response property*: in the Bisimulation Game, Duplicator always has a response leading to a process of size linearly bounded with respect to the Spoiler's process. The linear bound is effective, which leads to decidability of branching bisimilarity. For weak bisimilarity, we are able merely to show existence of some linear bound, which is not sufficient for decidability. We conjecture however that the same effective bound holds for weak bisimilarity as well. We believe that further elaboration of novel techniques developed in this paper may be sufficient to demonstrate decidability.

1 Introduction

Bisimulation equivalence (bisimilarity) is a fundamental notion of equivalence of processes, with many natural connections to logic, games and verification [10,13]. This paper is a continuation of the active line of research focusing on decidability and complexity of decision problems for bisimulation equivalence on various classes of infinite systems [12].

We investigate the class of commutative context-free processes, known also under name Basic Parallel Processes (BPP) [1]. By this we mean the labeled graphs induced by context-free grammars in Greibach normal form, with a proviso that non-terminals appearing on the right-hand side of a productions are assumed to be commutative. For instance, the production $X \longrightarrow aYZ$, written

$$X \xrightarrow{a} YZ,$$

says that X performs an action a and then executes Y and Z in parallel. Formally, the right-hand side is a multiset rather than a sequence.

Over this class of graphs, we focus on bisimulation equivalence as the primary type of semantic equality of processes. It is known that *strong* bisimulation equivalence is decidable [2] and PSPACE-complete [11,8]; and is polynomial

* The first author acknowledges a partial support by the Polish MNiSW grant N N206 568640. The remaining two authors acknowledge a partial support by the Polish MNiSW grant N N206 567840.

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 528–542, 2011.

for *normed* processes [6]. Dramatically less is known about weak bisimulation equivalence, that abstracts from the silent ε-transitions: we only know that it is semi-decidable [3] and that it is decidable in polynomial space over a very restricted class of *totally normed* processes [4]. The same applies to branching bisimulation equivalence, a variant of weak bisimulation that respects faithfully branching of equivalent processes. The only non-trivial decidability result known by now for weak bisimulation equivalence is [14], it applies however to a very restricted subclass.

During last two decades decidability of weak bisimulation over context-free processes became an established long-standing open problem. This paper is a significant step towards solving this problem in confirmative.

It is well known that bisimulation equivalences have an alternative formulation, in terms of Bisimulation Game played between Spoiler (aiming at showing non-equivalence) and Duplicator (aiming at showing equivalence) [13]. One of the main obstacles in proving decidability of weak (or branching) bisimulation equivalence is that Duplicator may do arbitrarily many silent transitions in a single move, and thus the size of the resulting process is hard to bound.

In this paper we investigate branching bisimilarity over normed commutative context-free processes. Our main technical result is the proof of the following *small response property*, formulated as Theorem 1 in Section 3: if Duplicator has a response, then he also has a response that leads to a process of size linearly bounded with respect to the other (Spoiler's) process. Importantly, we obtain an effective bound on the linear coefficient, which enables us to prove (Theorem 2) decidability of branching bisimulation equivalence. The proof of Theorem 1 is quite complex and involves a lot of subtle investigations of combinatorics of BPP transitions, the main purpose being elimination of unnecessary silent transitions.

A major part of the proof works for weak bisimulation equally well (and, as we believe, also for any reasonable equivalence that lies between the two equivalences). However, for weak bisimulation we can merely show *existence* of the linear coefficient witnessing the small response property, while we are not able to obtain any effective bound. Nevertheless we strongly believe (and conjecture) that a further elaboration of our approach will enable proving decidability of weak bisimulation. We plan to pursue this as a future work. In particular, we actually reprove decidability in the subclass investigated in [14].

2 Preliminaries

The commutative context-free processes (known also as Basic Parallel Processes) are determined by the following ingredients (called a *process definition*): a finite set $V = \{X_1, \ldots, X_n\}$ of variables, a finite set A of letters, and a finite set T of transition rules, each of the form $X \xrightarrow{\zeta} \alpha$ where X is a variable, $\zeta \in A \cup \{\varepsilon\}$ and α is a finite multiset of variables.

A *process*, is any finite multiset of variables, thus of the form $X_1^{a_1} \ldots X_n^{a_n}$, and may be understood as the parallel composition of a_1 copies of X_1, ... , and a_n copies of X_n. In particular the *empty process*, denoted ε, when $a_1 = \cdots = a_n = 0$.

For any $W \subseteq V$ we denote by $W^{\otimes}$ the set of all processes where only variables from W occur, that is, $W^{\otimes}$ is the set of all finite multisets over W.

By $\alpha\beta$ we mean the composition of processes α and β, understood as the multiset union. The behavior, i.e., the *transition relation*, is defined by the following extension rule:

$$\text{if } X \xrightarrow{\zeta} \alpha \in T \text{ then } X\beta \xrightarrow{\zeta} \alpha\beta, \text{ for any } \beta \in V^{\otimes}.$$

Remark 1. Commutative context-free processes are precisely labeled communication free Petri nets, where the places are variables and transitions $X \xrightarrow{\zeta} \alpha$ are firing rules. A process $X_1^{a_1} \ldots X_n^{a_n}$ represents the marking with a_i tokens on the place X_i.

The transition relation $\xrightarrow{\varepsilon}$ models silent steps and will be written $\longrightarrow$. We write $\alpha \Longrightarrow \beta$ if a process β can be reached from α by a sequence of $\xrightarrow{\varepsilon}$ transitions. To simplify definitions, we assume that $\alpha \longrightarrow \alpha$ for any α.

Definition 1. *A binary symmetric relation B over processes is a* branching bisimulation *iff for every pair α B β and $\zeta \in A \cup \{\varepsilon\}$ satisfies: if $\alpha \xrightarrow{\zeta} \alpha'$ then $\beta \Longrightarrow \beta'' \xrightarrow{\zeta} \beta'$ such that α B β'' and α' B β'.*

We say that two processes α and β are branching bisimilar, denoted $\alpha \approx \beta$, if there exists a branching bisimulation B such that α B β.

In the proofs we will use the characterization of bisimilarity in terms of Bisimulation Game [10,13]. The game is played by two players, Spoiler and Duplicator, over an arena consisting of all pairs of processes, and proceeds in rounds. Each round starts with a Spoiler's move followed by a Duplicator's response. In position (α, β), Spoiler chooses one of processes, say α, and one transition $\alpha \xrightarrow{\zeta} \alpha'$. As a response, Duplicator has to do a sequence of transitions of the form $\beta \Longrightarrow \beta'' \xrightarrow{\zeta} \beta'$, and then Spoiler chooses whether the play continues from (α, β'') or (α', β').

If one of players gets stuck, the other wins. Otherwise the play is infinite and in this case it is Duplicator who wins. A well-known fact is that two processes are branching bisimilar iff Duplicator has a winning strategy in the game that starts in these two processes.

For the rest of this paper we assume that each variable X has a sequence of transitions $X \xrightarrow{\zeta_1} \ldots \xrightarrow{\zeta_m} \varepsilon$ leading to the empty process. A process definition that fulfills this requirement is usually called *normed*. By the *norm* of X, denoted norm(X), we mean the smallest possible number of visible transitions that appears in some sequence as above. Formally speaking, the norm of X is the length of the shortest word $a_1 \ldots a_n \in A^*$ such that

$$X \Longrightarrow\xrightarrow{a_1}\Longrightarrow \ldots \Longrightarrow\xrightarrow{a_n}\Longrightarrow \varepsilon.$$

We additively enhance the definition of norm to processes and write norm(α) for any $\alpha \in V^{\otimes}$. Note that the norm is *weak* in the sense that silent transitions do not count.

3 Decidability via Small Response Property

It was known before that branching bisimilarity is semi-decidable [3]. A main obstacle for a semi-decision procedure for inequivalence is that commutative context-free processes are not image finite with respect to branching bisimilarity: a priori Duplicator has infinitely many possible responses to a Spoiler's move. The main insight of this paper is that commutative context-free processes *are* essentially image-finite, in the following sense. Define the size of a process as its multiset cardinality: $\mathrm{size}(X_1^{a_1} \dots X_n^{a_n}) = a_1 + \cdots + a_n$. Then Duplicator has always a response of size bounded linearly with respect to a Spoiler's process (cf. Theorem 1 below).

Definition 2. *Let $c \in \mathbb{N}$. By a c-branching bisimulation we mean a relation B defined as in Definition 1 with the additional requirement*

$$size(\beta'), size(\beta'') \leq c \cdot size(\alpha'). \tag{1}$$

Let the size d of a process definition be the sum of lengths of all production rules. Our main technical result is an efficient estimation of c, with respect to d:

Theorem 1 (small response property). *For each normed process definition of size d with n variables, branching bisimilarity $\approx$ is a $(2d^{n-1} + d)$-branching bisimulation.*

The proof of Theorem 1 is deferred to Sections 4–6.

In consequence, a Spoiler's winning strategy, seen as a tree, becomes finitely branching. This observation leads directly to decidability:

Theorem 2. *Branching bisimilarity $\approx$ is decidable over normed commutative context-free processes.*

Proof. We sketch two semi-decision procedures (along the lines of [9]): one for branching bisimilarity and the other for $(2d^{n-1} + d)$-branching bisimilarity.

For the positive side we use a standard semi-linear representation, knowing that each congruence, including $\approx$, is semi-linear [5,7]. The algorithm guesses a base-period representation of a semi-linear set and then checks validity of a Presburger formula that says that this set is a branching bisimulation containing the input pair of processes.

For the negative side, we observe that due to Theorem 1 Duplicator has only finitely many possible answers to each Spoiler's move. Thus, if Spoiler wins then its winning strategy may be represented by a finitely-branching tree. Furthermore, by König Lemma this tree is finite. The algorithm thus simply guesses a finite Spoiler's strategy. This can be done effectively: for given β, β', β'' and ζ it is decidable if $\beta \Longrightarrow_0 \beta'' \xrightarrow{\zeta} \beta'$, as the $\Longrightarrow_0$ relation is effectively semilinear [3]. □

Proof strategy. The rest of this paper is devoted to the proof of Theorem 1. Consider a fixed normed process definition from now on. In Section 4 we define a notion of normal form $\mathrm{nf}(\alpha)$ for a process α and provide linear lower and upper bounds on its size:

$$\mathrm{size}(\alpha) \leq \mathrm{size}(\mathrm{nf}(\alpha)) \leq c \cdot \mathrm{size}(\alpha) \tag{2}$$

(the lower bound holds assumed that α is minimal wrt. multiset inclusion in its bisimulation class). However, the linear coefficient c is not bounded effectively. The computable estimation of the coefficient is derived in Section 5. Finally, in Section 6 we show how the bounds (2) are used to prove Theorem 1. Due to space limitations we omit some proofs in Sections 4–6.

As observed e.g. in [14], a crucial obstacle in proving decidability is so called *generating* transitions of the form $X \longrightarrow XY$, as they may be used by Duplicator to reach silently XY^m for arbitrarily large m. A great part of our proofs is an analysis of combinatorial complexity of generating transitions and, roughly speaking, elimination of 'unnecessary' generations.

Weak bisimilarity. Branching bisimilarity is more discriminating than the well known weak bisimilarity. The whole development of Section 4 is still valid if weak bisimilarity is considered in place of branching bisimilarity. Furthermore, except one single case, the entire proof of estimation of the coefficient in Section 5 remains valid too. Interestingly, this single case is obvious under assumptions of [14], thus our proof remains valid for weak bisimilarity over the subclass studied there. We conjecture that the single missing case is provable for weak bisimilarity and thus Theorem 1 holds just as well. This would imply decidability.

4 Normal Form by Squeezing

In the sequel we often implicitly use the well-known fact that branching bisimilarity is substitutive, i.e., $\alpha \approx \beta$ implies $\alpha\gamma \approx \beta\gamma$.

In this section we develop a framework useful for the proof of Theorem 1 in the following sections. We define a normal form $\mathrm{nf}(\alpha)$ of a process α that identifies the bisimulation class of α uniquely. Moreover, we provide estimations of the size of $\mathrm{nf}(\alpha)$ relative to the size of α, from both sides, in Corollary 1 and Lemma 11, which culminate this section.

A transition $\alpha \xrightarrow{\zeta} \beta$ is *norm preserving* if $|\alpha| = |\beta|$ and *norm reducing* if $|\alpha| = |\beta| + 1$. In the sequel we will pay special attention to norm preserving ε-transitions. Therefore we write $\alpha \longrightarrow_0 \beta$, respectively $\alpha \Longrightarrow_0 \beta$, to emphasize that the transitions are norm preserving.

Lemma 1. *If $\alpha \Longrightarrow_0 \beta \Longrightarrow_0 \alpha'$ and $\alpha \approx \alpha'$ then $\beta \approx \alpha$.*

We call the transition $\alpha \xrightarrow{\zeta} \beta$ *decreasing* if either $\zeta \in A$ and the transition is norm-reducing; or $\zeta = \varepsilon$ and the transition is norm preserving. Note that every variable has a sequence of decreasing transitions leading to the empty process ε.

Lemma 2 (decreasing response). *Whenever $\alpha \approx \beta$ and $\alpha \xrightarrow{\zeta} \alpha'$ is decreasing then any Duplicator's matching sequence of transitions from β contains exclusively decreasing transitions.*

Due to Lemma 1, instantiated to single variables, we may assume wlog. that there are no two distinct variables X, Y with $X \Longrightarrow_0 Y \Longrightarrow_0 X$. Indeed, since reachability via the $\Longrightarrow_0$ transitions is decidable [3], in a preprocessing one may eliminate such pairs X, Y. Relying on this assumption, we may define a partial order induced by decreasing transitions.

Definition 3. *Let $X >_0 Y$ if there is a sequence of decreasing transitions leading from X to Y. Let $>$ denote an arbitrary fixed total order extending $>_0$.*

In the sequel we assume that there are n variables, ordered $X_1 > X_2 > \ldots > X_n$. Directly from the definition of $>$ we deduce:

Lemma 3 (decreasing transition). *If a decreasing transition $X_1^{a_1} \ldots X_n^{a_n} \xrightarrow{\zeta} X_1^{b_1} \ldots X_n^{b_n}$ is performed by X_k, say, then $b_1 = a_1, \ldots, b_{k-1} = a_{k-1}$.*

Consider a norm preserving silent transition $X \longrightarrow_0 \delta$. If X appears in δ, i.e. $\delta = X\bar{\delta}$, we call the transition *generating*. We use the name *generating* also for a general transition $\alpha \longrightarrow_0 \beta$ as a single transition is always performed by a single variable.

Lemma 4 (decreasing transition cont.). *If a decreasing transition as in Lemma 3 is not generating then $b_k = a_k - 1$.*

Following [14], we say that X *generates* Y if $X \Longrightarrow_0 XY$. Thus if $X \longrightarrow X\bar{\delta}$ then X generates every variable that appears in $\bar{\delta}$. In particular, X may generate itself. Note that each generated variable is of norm 0. More generally, we say that α generates β if $\alpha \Longrightarrow_0 \alpha\beta$. This is the case precisely iff every variable occurring in β is generated by some variable occurring in α. As a direct corollary of Lemma 1 we get ($\sqsubseteq$ stands for the multiset inclusion of processes):

Lemma 5. *If α generates β then $\alpha \approx \alpha\bar{\beta}$ for any $\bar{\beta} \sqsubseteq \beta$.*

Lemma 5 will be especially useful in the sequel, as a tool for eliminating unnecessary transitions and thus decreasing the size of a resulting process.

A process $X_1^{a_1} \ldots X_n^{a_n}$ may be equivalently presented as a sequence of exponents $(a_1 \ldots a_n) \in \mathbb{N}^n$. In this perspective, $\sqsubseteq$ is the point-wise order. The sequence presentation $(a_1 \ldots a_n) \in \mathbb{N}^n$ induces additionally the lexicographic order on processes, denoted $\preceq$. We will exploit the fact that this order is total, and thus each bisimulation class exhibits the least element. (A *bisimulation class* of a process α is the set of all processes β with $\beta \approx \alpha$.)

The sequence presentation allows us to speak naturally of *prefixes* of a process: the *k-prefix* of $X_1^{a_1} \ldots X_n^{a_n}$ is the process $X_1^{a_1} \ldots X_k^{a_k}$, for $k = 0 \ldots n$.

We now go to one of the crucial notions used in the proof: *unambiguous processes* and their *greatest extensions.*

Definition 4 (unambiguous processes). *A process $X_1^{a_1} \dots X_n^{a_n}$, is called* k-unambiguous *if for every $1 \leq i \leq k$, $\alpha, \beta \in \{X_{i+1}, \dots, X_n\}^{\otimes}$ and $b, c \in \mathbb{N}$ such that*

$$X_1^{a_1} X_2^{a_2} \dots X_{i-1}^{a_{i-1}} X_i^b \alpha \approx X_1^{a_1} X_2^{a_2} \dots X_{i-1}^{a_{i-1}} X_i^c \beta$$

we have either $b, c \geq a_i$ or $b = c$. When $k = n$ we write simply unambiguous.

Note that being k-unambiguous is a property of the k-prefix: a process is k-unambiguous iff its k-prefix is so.

Example 1. Consider following process definition:

$$\begin{array}{lll} X_1 \xrightarrow{a} X_1 & \quad X_2 \xrightarrow{b} X_3 & \quad X_3 \xrightarrow{b} \varepsilon \\ X_1 \longrightarrow \ \varepsilon & \quad X_2 \longrightarrow X_3 & \quad X_3 \longrightarrow \varepsilon \end{array}$$

and an order $X_1 > X_2 > X_3$ on variables. We observe that $X_1^2 \approx X_1$, therefore the process X_1^2 is not (1-)unambiguous. On the other hand $X_1 \not\approx \alpha$ for any $\alpha \in \{X_2, X_3\}^{\otimes}$ (because neither X_2 nor X_3 can perform an a transition), so X_1 is unambiguous. Furthermore $X_1 X_2 \approx X_1 X_3^2$, hence $X_1 X_2$ is not (2-)unambiguous. Finally we observe that $X_1 X_3^2 \not\approx X_1 X_3$. Therefore $X_1 X_3^2$ is unambiguous, but also $X_1 X_3$ is so. □

Note that a prefix of a k-unambiguous process is k-unambiguous as well. Moreover, k-unambiguous processes are downward closed wrt. $\sqsubseteq$: whenever $\alpha \sqsubseteq \beta$ and β is k-unambiguous, then α is k-unambiguous as well.

Directly by Definition 4, if $\gamma = X_1^{a_1} \dots X_{k-1}^{a_{k-1}}$ is $(k-1)$-unambiguous then it is automatically k-unambiguous (in fact j-unambiguous for any $j \geq k$). According to the sequence presentation, this corresponds to extending the process $(a_1 \dots a_{k-1})$ with $a_k = 0$. We will be especially interested in the greatest value of a_k possible, as formalized in the definition below.

Definition 5 (the greatest extension). *The* greatest k-extension *of a $(k-1)$-unambiguous process $\gamma = X_1^{a_1} \dots X_{k-1}^{a_{k-1}} \in \{X_1 \dots X_{k-1}\}^{\otimes}$ is that process among k-unambiguous processes γX_k^a that maximizes a.*

Clearly the greatest extension does not need exist in general, as illustrated below.

Example 2. Consider the processes from Example 1. The process X_1 is the greatest 1-extension of the empty process as X_1^2 is not 1-unambiguous. X_1 is also its own greatest 2-extension. Furthermore, X_1 does not have the greatest 3-extension. Indeed, $X_1 X_3^a$ is not bisimilar to $X_1 X_3^b$, for $a \neq b$, therefore $X_1 X_3^a$ is 3-unambiguous for any a. □

Definition 6 (unambiguous prefix). *By an* unambiguous prefix *of a process $X_1^{a_1} \dots X_n^{a_n}$ we mean any k-prefix $X_1^{a_1} \dots X_k^{a_k}$ that is k-unambiguous, for $k = 0 \dots n$. The* maximal unambiguous prefix *is the one that maximizes k.*

Example 3. For the process definition from Example 1, the maximal unambiguous prefix of $X_1 X_2^2$ is X_1, and the maximal unambiguous prefix of $X_1^2 X_2$ is the empty process. □

The following lemma is a crucial observation underlying our subsequent development.

Lemma 6. *Let $\gamma \in \{X_1 \dots X_{k-1}\}^{\otimes}$ be $(k-1)$-unambiguous and assume that γX_k^a is its greatest k-extension. Let $b > a$ and let $\alpha, \beta \in \{X_{k+1}, \dots, X_n\}^{\otimes}$ be arbitrary processes such that*

$$\gamma\, X_k^b\, \beta \approx \gamma\, X_k^a\, \alpha.$$

Then for any decreasing transition $X_k^b\, \beta \xrightarrow{\zeta} X_k^{b'}\, \beta'$, that gives rise to a Spoiler's move

$$\gamma\, X_k^b\, \beta \xrightarrow{\zeta} \gamma\, X_k^{b'}\, \beta'$$

there are some $\alpha', \alpha'' \in \{X_{k+1}, \dots, X_n\}^{\otimes}$ and a sequence $\alpha \Longrightarrow_0 \alpha'' \xrightarrow{\zeta} \alpha'$ of transitions that gives rise to a Duplicator's response

$$\gamma\, X_k^a\, \alpha \Longrightarrow_0 \gamma\, X_k^a\, \alpha'' \xrightarrow{\zeta} \gamma\, X_k^a\, \alpha',$$

as required by Definition 1.

Note 1. According to the assumptions, γX_k^a is an unambiguous prefix of $\gamma X_k^a \alpha$. The crucial consequence of the lemma is that Duplicator has a response that preserves γX_k^a being a prefix, as only α is engaged in the response.

Proof. Consider a Duplicator's response (all transition are necessarily decreasing by Lemma 2):

$$\gamma\, X_k^a\, \alpha \Longrightarrow_0 \gamma''\, X_k^{a''}\, \alpha'' \xrightarrow{\zeta} \gamma'\, X_k^{a'}\, \alpha' \tag{3}$$

where $\gamma', \gamma'' \in \{X_1 \dots X_{k-1}\}^{\otimes}$ and $\alpha', \alpha'' \in \{X_{k+1}, \dots, X_n\}^{\otimes}$. Wlog. we may assume that

$$\gamma''\, X_k^{a''}\, \alpha'' \not\approx \gamma'\, X_k^{a'}\, \alpha' \tag{4}$$

as otherwise lemma holds trivially. A fast observation is that

$$\gamma\, X_k^a \preceq \gamma'\, X_k^{a'}. \tag{5}$$

Indeed, suppose $\gamma'\, X_k^{a'} \prec \gamma\, X_k^a$. Knowing $\gamma\, X_k^{b'} \beta' \approx \gamma'\, X_k^{a'} \alpha'$ and $b' \geq a$ we get to a contradiction with the fact that $\gamma\, X_k^a$ is k-unambiguous.

Our aim is to demonstrate that Duplicator has a matching response (3) that uses only transition rules of variables $X_{k+1} \dots X_n$; in particular, by Lemma 3 this will imply $\gamma'\, X_k^{a'} = \gamma\, X_k^a$. We will describe below a transformation of the Duplicator's response to the required form.

Assume that some of variables $X_1 \dots X_k$ was engaged in (3) and let X_i be the greatest of them wrt. $>$. By (5) and by Lemma 4 we learn that at least one of transitions performed by some X_i must be generating, say

$$X_i \longrightarrow X_i \delta. \tag{6}$$

We will show how to remove one of these transitions from (3) but still preserve the bisimulation class of processes appearing along (3), and thus keep satisfying the requirements of Definition 1.

All variables that appear in δ are necessarily of norm 0, and thus they may participate later in the sequence (3) only with further norm preserving ε-transitions. Informally speaking, we consider the tree of norm preserving ε-transitions initiated by (6), that are performed along (3), say:

$$X_i \Longrightarrow_0 X_i^j \delta', \tag{7}$$

for some $j \geq 0$ and $\delta' \in \{X_{i+1} \dots X_n\}^{\otimes}$.

Formally, the sequence (7) is defined by the following coloring argument. As a process may contain many occurrences of the same variable we consider variable occurrences as independent entities. Assume that every variable occurrence in $\gamma\, X_k^a$ has been initially colored by a unique color. Assume further that colors are inherited via transitions: every transition in (3) is colored with the color of the occurrence of its left-hand side variable that is engaged; and likewise are colored all the right-hand side variables occurrences. The sequence (7) contains all transitions colored with the color of (6).

The sequence (7) forms a subsequence of (3). There can be many such sequences, but at least one witnesses $j > 0$, by (5) and by the choice of X_i as the greatest wrt. $>$. Let us focus on removing this particular subsequence from (3).

As δ' is generated by X_i, by Lemma 5 we obtain $X_i \approx X_i^j \delta'$. By our assumption (4) we deduce that the sequence (7) can not contain the last transition of (3). Thus, by substitutivity of $\approx$, the sequence (3), after removing transitions (7), yields a process bisimulation equivalent to that yielded by (3). By continuing in the same manner we arrive finally at the Duplicator's response that does not engage variables $X_1 \dots X_k$ at all. This completes the proof. □

Lemma 7 (squeezing out). *Let $\gamma \in \{X_1 \dots X_{k-1}\}^{\otimes}$ be $(k-1)$-unambiguous and assume that γX_k^a is its greatest k-extension. Then for some $\delta \in \{X_{k+1} \dots X_n\}^{\otimes}$ it holds:*

$$\gamma\, X_k^{a+1} \approx \gamma\, X_k^a\, \delta. \tag{8}$$

Definition 7. *If a $(k-1)$-unambiguous process $\gamma \in \{X_1 \dots X_{k-1}\}^{\otimes}$ has the greatest k-extension, say γX_k^a, then the variable X_k is called γ-*squeezable *and any $\delta \in \{X_{k+1} \dots X_n\}^{\otimes}$ satisfying (8) is called a γ-*squeeze *of X_k.*

By the very definition, X_k has a γ-squeeze only if it is γ-squeezable. Lemma 7 shows the opposite: a γ-squeezable X_k has a γ-squeeze, that may depend in general on γ and k. The squeeze is however not uniquely determined and in fact X_k may admit many different γ-squeezes. In the sequel assume that for each $(k-1)$-unambiguous $\gamma \in \{X_1 \dots X_{k-1}\}^{\otimes}$ and X_k, some γ-squeeze of X_k is chosen; this squeeze will be denoted by $\delta_{k,\gamma}$.

Definition 8 (squeezing step). *For a given process α, assuming it is not n-unambiguous, let γ be its maximal unambiguous prefix. Thus there is $k \leq n$ such that*

$$\alpha = \gamma\, X_k^a\, \delta,$$

$\gamma \in \{X_1 \ldots X_{k-1}\}^{\otimes}$, $\delta \in \{X_{k+1} \ldots X_n\}^{\otimes}$, and $\gamma\, X_k^a$ is not k-unambiguous. Note that a is surely greater than 0. We define squeeze(α) by

$$squeeze(\alpha) = \gamma\, X_k^{a-1}\, \delta_{k,\gamma}\, \delta.$$

Otherwise, i.e. when α is n-unambiguous, for convenience put squeeze$(\alpha) = \alpha$.

By Lemma 7 and by substitutivity of $\approx$ we conclude that $\alpha \approx \text{squeeze}(\alpha)$ and if α is not unambiguous then $\text{squeeze}(\alpha) \prec \alpha$.

We have the following characterization of unambiguous processes:

Lemma 8. *A process α is n-unambiguous if and only if it is the least one in its bisimulation class wrt. $\preceq$.*

Lemma 7, applied in a systematic manner sufficiently many times on a process α, yields a kind of normal form, as stated in Lemma 9 below. A process α we call shortly $\sqsubseteq$-minimal if there is no $\beta \sqsubset \alpha$ with $\beta \approx \alpha$.

Definition 9 (normal form). *For any process α let nf(α) denote the unambiguous process obtained by consecutive alternating applications of the following two steps:*

- *the squeezing step: replace α by squeeze(α),*
- *the $\sqsubseteq$-minimization step: replace α by any $\sqsubseteq$-minimal $\bar{\alpha} \sqsubseteq \alpha$ with $\bar{\alpha} \approx \alpha$.*

As $\alpha \approx \text{squeeze}(\alpha)$ then $\alpha \approx \text{nf}(\alpha)$ and thus using Lemma 8 we conclude that bisimulation equivalence is characterized by syntactic equality of normal forms:

Lemma 9. *$\alpha \approx \beta$ if and only if nf$(\alpha) =$ nf(β).*

Finally we are able to formulate lower and upper bounds on the size of $\text{nf}(\alpha)$, with respect to the size of α, that will be crucial for the proof of Theorem 1. The first one applies uniquely to $\sqsubseteq$-minimal processes.

Lemma 10. *If α is $\sqsubseteq$-minimal then size$(\alpha) \leq$ size$(\bar{\alpha})$, for any $\bar{\alpha} \sqsubseteq$ squeeze(α) such that $\bar{\alpha} \approx$ squeeze(α).*

Corollary 1 (lower bound). *If α is $\sqsubseteq$-minimal then size(nf$(\alpha)) \geq$ size(α).*

Lemma 11 (upper bound). *There is a constant c, depending only on the process definition, such that size(nf$(\alpha)) \leq c \cdot$ size(α) for any process α.*

Concerning the upper bound, in the following section we demonstrate a sharper result, with the constant c estimated effectively.

5 Small Normal Form

Denote the size of the process definition by d.

Lemma 12 (upper bound). *For any α, size(nf$(\alpha)) \leq d^{n-1} \cdot$ size(α).*

Lemma 12 follows immediately from Lemma 13 that says that squeezing does not increase a weighted measure of size, defined as:

$$d\text{-size}(X_1^{a_1} \ldots X_n^{a_n}) = a_1 \cdot d^{n-1} + a_2 \cdot d^{n-2} + \ldots + a_{n-1} \cdot d + a_n.$$

Lemma 13. *For every k and $(k-1)$-unambiguous $\gamma \in \{X_1 \ldots X_{k-1}\}^{\otimes}$, if X_k is γ-squeezable then it has a γ-squeeze δ with d-size$(\delta) \leq d$-size(X_k).*

Indeed, Lemma 13 implies d-size(nf(α)) $\leq d$-size(α) and then Lemma 12 follows:

$$\text{size}(\text{nf}(\alpha)) \leq d\text{-size}(\text{nf}(\alpha)) \leq d\text{-size}(\alpha) \leq d^{n-1} \cdot \text{size}(\alpha).$$

Before embarking on the proof of Lemma 13, we formulate a slight generalization of Lemma 6 from Section 4. For two processes $\alpha, \beta \in \{X_1 \ldots X_l\}^{\otimes}$ we say that α *is l-dominating* β if α is bisimilar to some $\alpha' \sqsupseteq \beta$.

Lemma 14. *Let α be an arbitrary process, $\beta_1 \in \{X_1 \ldots X_l\}^{\otimes}$ be m-unambiguous and $\beta_2 \in \{X_{l+1} \ldots X_n\}^{\otimes}$ such that $\alpha \approx \beta_1\beta_2$. Let $\alpha \xrightarrow{\zeta} \alpha'$ be an arbitrary decreasing transition such that the l-prefix of α' is l-dominating β_1. Then there is a sequence of transitions $\beta_2 \Longrightarrow_0 \beta_2'' \xrightarrow{\zeta} \beta_2'$ that gives rise to a Duplicator's response*

$$\beta_1\beta_2 \Longrightarrow_0 \beta_1\beta_2'' \xrightarrow{\zeta} \beta_1\beta_2',$$

as required by Definition 1.

Lemma 14 is proved in exactly the same way as Lemma 6. Recalling Lemma 6 observe that it is indeed a special case of Lemma 14: $\gamma X_k^{b'}$ is surely k-dominating γX_k^a as $b' \geq b - 1 \geq a$.

Now we return to the proof of Lemma 13, by induction on k. For $k = n$ it trivially holds. Fix $k < n$ and assume the lemma for all greater values of k. Fix a $(k-1)$-unambiguous $\gamma \in \{X_1 \ldots X_{k-1}\}^{\otimes}$ and consider its greatest k-extension γX_k^a. The proof is split into three cases:

- $a > 0$,
- $a = 0$ and X_k has a γ-squeeze δ such that $X_k \Longrightarrow_0 \delta$,
- $a = 0$ and X_k has no γ-squeeze δ such that $X_k \Longrightarrow_0 \delta$.

In the rest of this section we prove the last case only. The other cases are omitted due to space limitations.

Simplifying assumption. Variables $X_{k+1} \ldots X_n$ may be split into those generated by X_k, an those not generated by X_k. A simple but crucial observation is that the order $>$ on variables $X_{k+1} \ldots X_n$ may be rearranged, without losing generality, so that all variables generated by X_k are smaller than all variables not generated by X_k. Clearly, if we provide a γ-squeeze of X_k for the rearranged order, it is automatically a γ-squeeze of X_k for the initial order.

Thus for some $l \geq k$ we know that variables $X_{l+1} \ldots X_n$ are all generated by X_k, and all the remaining variables $X_{k+1} \ldots X_l$ are not generated by X_k. To emphasize this we will write $[\alpha \cdot \beta]$ for the composition of α and β, instead of $\alpha\beta$, whenever we know that $\alpha \in \{X_{k+1} \ldots X_l\}^{\otimes}$ and $\beta \in \{X_{l+1} \ldots X_n\}^{\otimes}$.

Lemma 15. *No γ-squeeze of X_k contains a variable generated by X_k.*

Proof. Assume the contrary, that is,

$$\gamma\, X_k \approx \gamma\, \delta'\, Y, \tag{9}$$

with $\delta'\, Y \in \{X_{k+1} \ldots X_n\}^{\otimes}$ and Y generated by X_k. Consider the Bisimulation Game for $\gamma X_k \approx \gamma\, \delta'\, Y$ and an arbitrary sequence of $\longrightarrow_0$ transitions $Y \Longrightarrow_0 \varepsilon$ from Y to the empty process ε, giving rise to the sequence of Spoiler's moves

$$\gamma\, \delta'\, Y \Longrightarrow_0 \gamma\, \delta'.$$

By Lemma 14 we know that there is a Duplicator's response that does not engage γ at all:

$$\gamma\, X_k \Longrightarrow_0 \gamma\, \omega,$$

i.e. $X_k \Longrightarrow_0 \omega$. Now substituting $\gamma\omega$ in place of $\gamma\delta'$ in (9) we obtain a γ-squeeze of X_k

$$\gamma\, X_k \approx \gamma\, \omega\, Y,$$

such that $X_k \longrightarrow_0 X_k Y \Longrightarrow_0 X_k\, \omega\, Y$. This is in contradiction with the assumption that no γ-squeeze is reachable from X_k by $\Longrightarrow_0$. Thus the claim is proved. □

Using Lemma 15 we deduce that the normal form $\text{nf}(\gamma\, X_k) = \gamma\, \delta$ contains no variable generated by X_k, i.e., $\text{nf}(\gamma\, X_k) = \gamma\, [\delta \cdot \varepsilon]$. We will show that the weighted size of δ satisfies the required bound.

Consider the Bisimulation Game for $\gamma\, X_k \approx \gamma\, \delta$ and the Spoiler's move from the smallest variable occurring in δ wrt. $>$, say X_m. Process δ contains no variable generated by X_k, hence $m \leq l$. Thus $\delta = \delta'\, X_m$, and let the Spoiler's move be induced by a decreasing non-generating transition $X_m \xrightarrow{\zeta} \omega$:

$$\gamma\, \delta'\, X_m \xrightarrow{\zeta} \gamma\, \delta'\, \omega.$$

By Lemma 14 we know that there is a Duplicator's response that does not engage γ. As no γ-squeeze of X_k is reachable from X_k by $\Longrightarrow_0$, the response has necessarily the following form

$$\gamma\, X_k \Longrightarrow_0 \gamma\, X_k\, \eta \xrightarrow{\zeta} \gamma\, \sigma\, \eta,$$

where η is generated by X_k:

$$X_k \Longrightarrow_0 X_k\, \eta \quad \text{and} \quad X_k \xrightarrow{\zeta} \sigma,$$

as otherwise at some point in the $\Longrightarrow_0$ sequence a γ-squeeze would appear. We obtain $\gamma\, \sigma\, \eta \approx \gamma\, \delta'\, \omega$ and thus

$$\text{nf}(\gamma\, \sigma\, \eta) \;=\; \text{nf}(\gamma\, \delta'\, \omega). \tag{10}$$

From the last equality we will deduce how the sizes of $\mathrm{nf}(\gamma\,\sigma)$ and $\mathrm{nf}(\gamma\,\delta')$ are related, in order to conclude that the weighted size of δ is as required.

Let's inspect the l-prefix of the left processes in (10). Process η can not contribute to that prefix of the normal form, thus if we restrict to the l-prefixes we have the equality

$$l\text{-prefix}(\mathrm{nf}(\gamma\,\sigma\,\eta)) \;=\; l\text{-prefix}(\mathrm{nf}(\gamma\,\sigma)). \tag{11}$$

Similarly, let's inspect the m-prefix of the right process in (10). Again, ω can not contribute to that prefix of the normal form, thus if we restrict to the m-prefixes we have the equality

$$m\text{-prefix}(\mathrm{nf}(\gamma\,\delta'\,\omega)) \;=\; m\text{-prefix}(\mathrm{nf}(\gamma\,\delta')).$$

As $\gamma\,\delta$ is the normal form, the process $\gamma\,\delta'$ is unambiguous and thus clearly $\mathrm{nf}(\gamma\,\delta') = \gamma\,\delta'$. Substitute this to the last equality above:

$$m\text{-prefix}(\mathrm{nf}(\gamma\,\delta'\,\omega)) \;=\; m\text{-prefix}(\gamma\,\delta') \;=\; \gamma\,\delta'. \tag{12}$$

Using induction assumption we obtain $d\text{-size}(\mathrm{nf}(\gamma\,\sigma)) \leq d\text{-size}(\gamma\,\sigma)$. As $m \leq l$, by (10), (11) and (12) we conclude that

$$d\text{-size}(\gamma\,\delta') \leq d\text{-size}(\mathrm{nf}(\gamma\,\sigma)) \leq d\text{-size}(\gamma\,\sigma)$$

and thus $d\text{-size}(\delta') \leq d\text{-size}(\sigma)$. By the last inequality together with $\mathrm{size}(\sigma) \leq d-1$ and $\sigma \in \{X_{k+1}\dots X_n\}^{\otimes}$ we get the required bound on weighted size of δ:

$$\begin{aligned} d\text{-size}(\delta) = d\text{-size}(\delta') + d\text{-size}(X_m) \leq d\text{-size}(\sigma) + d^{n-m} \leq \\ (d-1)\,d^{n-k-1} + d^{n-m} \leq d^{n-k} = d\text{-size}(X_k). \end{aligned}$$

6 Proof of the Small Response Property

Now we show how Theorem 1 follows from the estimations given in Corollary 1 and Lemma 12. We will need a definition and two lemmas.

We write $\alpha \overset{\approx}{\Longrightarrow}_0 \beta$ if $\alpha \Longrightarrow_0 \beta$ and $\alpha \approx \beta$. A process α is called $\overset{\approx}{\Longrightarrow}_0$-*minimal* if there is no $\beta \prec \alpha$ with $\alpha \overset{\approx}{\Longrightarrow}_0 \beta$.

Lemma 16. *For any α there is a $\overset{\approx}{\Longrightarrow}_0$-minimal process $\bar{\alpha}$ with $\alpha \overset{\approx}{\Longrightarrow}_0 \bar{\alpha}$ of size bounded by size($\bar{\alpha}$) $\leq$ size(nf(α)).*

Lemma 17. *If α is $\overset{\approx}{\Longrightarrow}_0$-minimal and $\alpha \overset{\approx}{\Longrightarrow}_0 \beta$ then $\alpha \sqsubseteq \beta$.*

Proof of Theorem 1. Consider $\alpha \approx \beta$, a Spoiler's move $\alpha \xrightarrow{\zeta} \alpha'$ and a Duplicator's response: $\beta \Longrightarrow_0 \beta_1 \xrightarrow{\zeta} \beta_2$, with $\alpha \approx \beta_1$ and $\alpha' \approx \beta_2$. The basic idea of the proof is essentially to eliminate some unnecessary generation done by transitions $\beta \Longrightarrow_0 \beta_1$.

As the first step we apply Lemma 16 to β, thus obtaining a sequence of transitions $\beta \Longrightarrow_0 \bar{\beta}$, for some $\overset{\approx}{\Longrightarrow}_0$-minimal process $\bar{\beta}$, in order to consider the pair $(\alpha, \bar{\beta})$ instead of (α, β). Knowing $\alpha \approx \bar{\beta}$ we obtain a Duplicator's response

$$\beta \Longrightarrow_0 \bar{\beta} \Longrightarrow_0 \beta'_1 \xrightarrow{\zeta} \beta'_2 \tag{13}$$

with $\alpha \approx \beta'_1$ and $\alpha' \approx \beta'_2$. Note that by Lemma 17 we know $\bar{\beta} \sqsubseteq \beta'_1$.

As the second step extend (13) with any sequence $\beta'_2 \overset{\approx}{\Longrightarrow}_0 \bar{\beta}'_2$ leading to a $\sqsubseteq$-minimal process $\bar{\beta}'_2 \sqsubseteq \beta'_2$. Our knowledge may be outlined with the following diagram (the subscript in $\Longrightarrow_0$ is omitted):

$$\begin{tikzcd}
\bar{\beta} \arrow[r, Rightarrow, "\approx", "\sqsubseteq"'] & \beta'_1 \arrow[d, "\zeta"] \\
\bar{\beta}'_2 & \beta'_2 \arrow[l, Rightarrow, "\approx"', "\sqsubseteq"]
\end{tikzcd}$$

Both left-most processes in the diagram are size bounded. Indeed, Corollary 1 applied to $\bar{\beta}$ and $\bar{\beta}'_2$ yields

$$\text{size}(\bar{\beta}) \leq \text{size}(\text{nf}(\alpha)) \quad \text{and} \quad \text{size}(\bar{\beta}'_2) \leq \text{size}(\text{nf}(\alpha')).$$

Then applying Lemma 12 to α and α' we obtain:

$$\text{size}(\bar{\beta}) \leq \text{size}(\alpha) \cdot d^{n-1} \quad \text{and} \quad \text{size}(\bar{\beta}'_2) \leq \text{size}(\alpha') \cdot d^{n-1}. \tag{14}$$

As the third and the last step of the proof, we claim that β'_1 and β'_2 may be replaced by processes of size bounded, roughly, by the sum of sizes of $\bar{\beta}$ and $\bar{\beta}'_2$.

Claim. There are some processes $\beta''_1 \approx \beta'_1$ and $\beta''_2 \approx \beta'_2$ such that

$$\bar{\beta} \Longrightarrow_0 \beta''_1 \xrightarrow{\zeta} \beta''_2 \tag{15}$$

and

$$\text{size}(\beta''_1), \text{size}(\beta''_2) \leq \text{size}(\bar{\beta}) + \text{size}(\bar{\beta}'_2) + d. \tag{16}$$

The claim is sufficient for Theorem 1 to hold, by inequalities (14). Thus to complete the proof we only need to demonstrate the claim. The idea underlying the proof of the claim is illustrated by the following diagram:

$$\begin{tikzcd}
\bar{\beta} \arrow[r, Rightarrow, "\approx", "\sqsubseteq"'] \arrow[dr, dotted, dash, "\sqsubseteq"'] & & \beta'_1 \arrow[d, "\zeta"] \arrow[dl, Rightarrow, dashed, "\approx"', "\sqsubseteq"] \\
& \beta''_1 \arrow[d, "\zeta"'] & \beta'_2 \arrow[dl, Rightarrow, "\approx"', "\sqsubseteq"] \\
\bar{\beta}'_2 & \beta''_2 \arrow[l, Rightarrow, "\approx"', "\sqsubseteq"] &
\end{tikzcd}$$

We use a coloring argument, similarly as in the proof of Lemma 6. Let us color uniquely every variable occurrence in β'_1 and let every transition preserve the color of the left-hand side variable. Obviously at most size($\bar{\beta}'_2$) of these colors will be still present in $\bar{\beta}'_2$, name them *surviving colors*. Let the $\beta'_1 \xrightarrow{\zeta} \beta'_2$ transition be performed due to a transition rule $X \xrightarrow{\zeta} \delta$, color this particular X, say, brown.

Let β_1'' consists of all variables which either belong to $\bar{\beta}$ or are colored surviving or brown color. Thus clearly $\bar{\beta} \sqsubseteq \beta_1'' \sqsubseteq \beta_1'$. One easily observes that after the brown transition $X \xrightarrow{\zeta} \delta$ from β_1'' we get β_2'' such that $\bar{\beta}_2' \sqsubseteq \beta_2'' \sqsubseteq \beta_2'$, because all surviving colored variables are still present. By Lemma 1 one has $\beta_1'' \approx \beta_1'$ and $\beta_2'' \approx \beta_2'$.

Finally we obtain the size estimation $\mathrm{size}(\beta_1'') \leq \mathrm{size}(\bar{\beta}) + \mathrm{size}(\bar{\beta}_2') + 1$ as in β_1'' there can be at most $\mathrm{size}(\bar{\beta}_2') + 1$ surviving and brown colored variables that do not belong to $\bar{\beta}$. This easily implies the estimation for $\mathrm{size}(\beta_2'')$. □

Acknowledgments. We are grateful to the reviewers for valuable comments, in particular for encouraging us to restrict the paper to branching bisimilarity.

References

1. Christensen, S.: Decidability and Decomposition in process algebras. PhD thesis, Dept. of Computer Science, University of Edinburgh, UK (1993)
2. Christensen, S., Hirshfeld, Y., Moller, F.: Bisimulation equivalence is decidable for Basic Parallel Processes. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 143–157. Springer, Heidelberg (1993)
3. Esparza, J.: Petri nets, commutative context-free grammars, and Basic Parallel Processes. Fundam. Inform. 31(1), 13–25 (1997)
4. Fröschle, S., Lasota, S.: Normed processes, unique decomposition, and complexity of bisimulation equivalences. Electr. Notes Theor. Comp. Sci. 239, 17–42 (2009)
5. Hirshfeld, Y.: Congruences in commutative semigroups. Technical report, University of Edinburgh, LFCS report ECS-LFCS-94-291 (1994)
6. Hirshfeld, Y., Jerrum, M., Moller, F.: A polynomial-time algorithm for deciding bisimulation equivalence of normed Basic Parallel Processes. Mathematical Structures in Computer Science 6(3), 251–259 (1996)
7. Jancar, P.: Decidability questions for bismilarity of Petri nets and some related problems. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 1994. LNCS, vol. 775, pp. 581–592. Springer, Heidelberg (1994)
8. Jancar, P.: Strong bisimilarity on Basic Parallel Processes is PSPACE-complete. In: LICS, pp. 218–227 (2003)
9. Lasota, S.: Decidability of performance equivalence for Basic Parallel Processes. Theoretical Computer Science 360, 172–192 (2006)
10. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1995)
11. Srba, J.: Strong Bisimilarity and Regularity of Basic Parallel Processes Is PSPACE-Hard. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 535–546. Springer, Heidelberg (2002)
12. Srba, J.: Roadmap of Infinite results. Formal Models and Semantics, vol. 2. World Scientific Publishing Co., Singapore (2004)
13. Stirling, C.: The joys of bisimulation. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 142–151. Springer, Heidelberg (1998)
14. Stirling, C.: Decidability of Weak Bisimilarity for a Subset of Basic Parallel Processes. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 379–393. Springer, Heidelberg (2001)

Refining the Process Rewrite Systems Hierarchy via Ground Tree Rewrite Systems

Stefan Göller[1] and Anthony Widjaja Lin[2]

[1] Universität Bremen, Institut für Informatik, Germany
[2] Oxford University Department of Computer Science, UK

Abstract. In his seminal paper, R. Mayr introduced the well-known Process Rewrite Systems (PRS) hierarchy, which contains many well-studied classes of infinite systems including pushdown systems, Petri nets and PA-processes. A seperate development in the term rewriting community introduced the notion of Ground Tree Rewrite Systems (GTRS), which is a model that strictly extends pushdown systems while still enjoying desirable decidable properties. There have been striking similarities between the verification problems that have been shown decidable (and undecidable) over GTRS and over models in the PRS hierarchy such as PA and PAD processes. It is open to what extent PRS and GTRS are connected in terms of their expressive power. In this paper we pinpoint the exact connection between GTRS and models in the PRS hierarchy in terms of their expressive power with respect to strong, weak, and branching bisimulation. Among others, this connection allows us to give new insights into the decidability results for subclasses of PRS, e.g., simpler proofs of known decidability results of verifications problems on PAD.

1 Introduction

The study of infinite-state verification has revealed that *unbounded recursions* and *unbounded parallelism* are two of the most important sources of infinity in the programs. Infinite-state models with unbounded recursions such as Basic Process Algebra (BPA), and Pushdown Systems (PDS) have been studied for a long time (e.g. [2,21]). The same can be said about infinite-state models with unbounded parallelism, which include Basic Parallel Processes (BPP) and Petri nets (PN), e.g. [10,14]. While these aforementioned models are either *purely sequential* or *purely parallel*, there are also models that simultaneously inherit both of these features. A well-known example are PA-processes [3], which are a common generalization of BPA and BPP. It is known that all of these models are not Turing-powerful in the sense that decision problems such as reachability is still decidable (e.g. see [9]), which makes them suitable for verification.

In his seminal paper [18], R. Mayr introduced the Process Rewrite Systems (PRS) hierarchy (see leftmost diagram in Figure 1) containing several models of infinite-state systems that generalize the aforementioned well-known models with unbounded recursions and/or unbounded parallelism. The idea is to treat models in the hierarchy as a form of term rewrite systems, and classify them according to which terms are permitted on the left/right hand sides of the rewrite rules. In addition to the aforementioned models of infinite systems, the PRS hierarchy contains three new models: (1) Process

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 543–558, 2011.

Rewrite Systems (PRS), which generalize PDS, PA-processes, and Petri nets, (2) PAD-processes, which unify PDS and PA-processes, and (3) PAN-processes, which unify both PA-processes and Petri nets. Mayr showed that the hierarchy is strict with respect to strong bisimulation. Despite of its expressive power PRS is not Turing-powerful since reachability is still decidable for this class. Before the PRS hierarchy was introduced, another class of infinite-state systems called Ground Tree/Term Rewrite Systems (GTRS) already emerged in the term rewriting community as a class with nice decidability properties. While extending the expressive power of PDS, GTRS still enjoys decidability of reachability (e.g. [8,11]), recurrent reachability [15], model checking first-order logic with reachability [12], and model checking the fragments LTL_{det} and $\text{LTL}(\mathbf{F}_{\text{s}}, \mathbf{G}_{\text{s}})$ of LTL [24,23]. Due to the tree structures that GTRS use in their rewrite rules, GTRS can be used to model concurrent systems with both unbounded parallelism (a new thread may be spawned at any given time) and unbounded recursions (each thread may behave as a pushdown system).

When comparing the definitions of PRS (and subclasses thereof) and GTRS, one cannot help but notice their similarity. Moreover, there is a striking similarity between the problems that are decidable (and undecidable) over subclasses of PRS like PA/PAD-processes and GTRS. For example, reachability, EF model checking, and $\text{LTL}(\mathbf{F}_{\text{s}}, \mathbf{G}_{\text{s}})$ and LTL_{det} model checking are decidable for both PAD-processes and GTRS [7,15,18,19,23,24]. Furthermore, model checking general LTL properties is undecidable for both PA-processes and GTRS [7,24]. Despite these, the precise connection between the PRS hierarchy and GTRS is currently still open.

Contributions: In this paper, we pinpoint the precise connection between the expressive powers of GTRS and models inside the PRS hierarchy with respect to strong, branching, and weak bisimulation. Bisimulations are well-known and important notions of semantic equivalences on transition systems. Among others, most properties of interests in verification (e.g. those expressible in standard modal/temporal logics) cannot distinguish two transition systems that are bisimilar. Strong/weak bisimulations are historically the most important notions of bisimulations on transition systems in verification [20]. Weak bisimulations extend strong bisimulations by distinguishing observable and non-observable (i.e. τ) actions, and only requiring the observable behavior of two systems to agree. In this sense, weak bisimulation is a coarser notion than strong bisimulation. Branching bisimulation [25] is a notion of semantic equivalence that is strictly coarser than strong bisimulation but is strictly finer than weak bisimulation. It refines weak bisimulation equivalence by preserving the branching structure of two processes even in the presence of unobservable τ-actions; it is required that all intermediate states that are passed through during τ-transitions are related.

Our results are summarized in the middle and right diagrams in Figure 1. Our first main result is that the expressive power of GTRS with respect to branching and weak bisimulation is strictly in between PAD and PRS but incomparable with PAN. This result allows us to transfer many decidability/complexity results of model checking problems over GTRS to PA and PAD-processes. In particular, it gives a simple proof of the decidability of model checking the logic EF over PAD [19], and decidability (with good complexity upper bounds) of model checking the common fragments LTL_{det} and $\text{LTL}(\mathbf{F}_{\text{s}}, \mathbf{G}_{\text{s}})$ of LTL over PAD (this decidability result was initially given in [7] without upper bounds). In fact, we also show that Regular Ground Tree Rewrite Systems (RGTRS) [15] — the extension of GTRS with possibly infinitely many GTRS rules compactly represented as tree automata — have the same expressive power as GTRS up to branching/weak bisimulation. Our proof technique also implies that PDS is equivalent to prefix-recognizable systems (e.g. see [9]), abbreviated as PREF, up to branching/weak bisimulation. On the other hand, when we investigate the expressive power of GTRS with respect to strong bisimulation, we found that PAD (even PA) is no longer subsumed in GTRS. Despite this, we can show that up to strong bisimulation GTRS is strictly more expressive than BPP and PDS, and is strictly subsumed in PRS. Finally, we mention that our results imply that Mayr's PRS hierarchy is also strict with respect to weak bisimulation equivalence.

Related work: Our work is inspired by the work of Lugiez and Schnoebelen [16] and Bouajjani and Touili [6], which study PRS (or subclasses thereof) by first distinguishing process terms that are "equivalent" in Mayr's sense [18]. This approach allows them to make use of techniques from classical theory of tree automata for solving interesting problems over PRS (or subclasses thereof). Our translation from PAD to GTRS is similar in spirit.

There are other models of multithreaded programs with unbounded recursions that have been studied in the literature. Specifically, we mention Dynamic Pushdown Networks (DPN) and extensions thereof (e.g. see [5]) since an extension of DPN given in [5] also extends PAD-processes. We leave it for future work to study the precise connections between these models and GTRS.

Organization: Preliminaries are given in Section 2. We provide the models of infinite systems (PRS, GTRS, etc.) in Section 3. Our containment results (e.g. PAD is subsumed in GTRS up to branching bisimulation) can be found in Section 4. Section 5 gives the separation results for the refined PRS hierarchies. Finally, we briefly discuss applications of our results in Section 6.

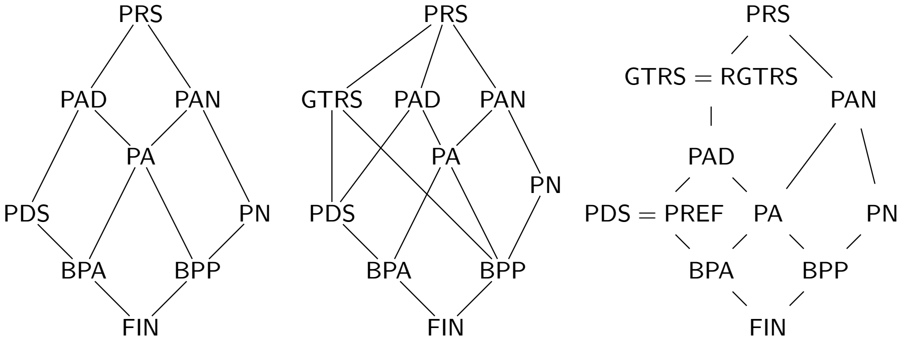


Fig. 1. Depictions of Mayr's PRS hierarchy and their refinements via GTRS as Hasse diagrams (the top being the most expressive). The leftmost diagram is the original (strict) PRS hierarchy where expressiveness is measured with respect to strong bisimulation. The middle (resp. right) diagram is a strict refinement via GTRS with respect to strong (resp. weak/branching) bisimulation.

2 Preliminaries

By $\mathbb{N} = \{0, 1, 2, \ldots\}$ we denote the non-negative integers. For each $i, j \in \mathbb{N}$ we define the interval $[i, j] = \{i, i+1, \ldots, j\}$.

Transition systems and weak/branching/strong bisimulation equivalence: Let us fix a countable set of action labels Act. A *transition system* is tuple $\mathcal{T} = (S, \mathbb{A}, \{\xrightarrow{a} \mid a \in \mathbb{A}\})$, where S is a set of *states*, $\mathbb{A} \subseteq \mathsf{Act}$ is a finite set of action labels, and where $\xrightarrow{a} \subseteq S \times S$ is a set of *transitions*. We write $s \xrightarrow{a} t$ to abbreviate $(s, t) \in \xrightarrow{a}$. We apply similar abbreviations for other binary relations over S. For each $R \subseteq S \times S$, we write sR to denote that there is some $t \in S$ with $(s, t) \in R$. For each $\Lambda \subseteq \mathbb{A}$, we define $\xrightarrow{\Lambda} = \bigcup_{a \in \Lambda} \xrightarrow{a}$ and we define $\longrightarrow = \xrightarrow{\mathbb{A}}$. Whenever $\mathcal{T}$ is clear from the context and $U \subseteq S$, we define $\mathsf{post}^*_\Lambda(U) = \{t \in S \mid \exists s \in U : s \xrightarrow{\Lambda}{}^{*} t\}$. In case $U = \{s\}$ is a singleton, we also write $\mathsf{post}^*_\Lambda(s)$ for $\mathsf{post}^*_\Lambda(U)$.

A *pointed transition system* is a pair $(\mathcal{T}, s)$, where $\mathcal{T}$ is a transition system and s is some state of $\mathcal{T}$. Let $\mathcal{T} = (S, \mathbb{A}, \{\xrightarrow{a} \mid a \in \mathbb{A}\})$ be a transition system. A relation $R \subseteq S \times S$ is a *strong bisimulation* if R is symmetric and for each $(s, t) \in R$ and for each $a \in \mathbb{A}$ we have that if $s \xrightarrow{a} s'$, then there is $t \xrightarrow{a} t'$ such that $(s', t') \in R$. We say that s is *strongly bisimilar* to t (abbreviated by $s \sim t$) whenever there is a strong bisimulation R such that $(s, t) \in R$.

Next, we define the notions of branching bisimulation and weak bisimulation. For this, let us fix a *silent action* $\tau \notin \mathbb{A}$ and let $\mathbb{A}_\tau = \mathbb{A} \cup \{\tau\}$. Moreover let $\mathcal{T} = (S, \mathbb{A}_\tau, \{\xrightarrow{a} \mid a \in \mathbb{A}_\tau\})$ be a transition system. We define the binary relations $\overset{\tau}{\Longrightarrow} = (\xrightarrow{\tau})^*$ and $\overset{a}{\Longrightarrow} = (\xrightarrow{\tau})^* \circ \xrightarrow{a} \circ (\xrightarrow{\tau})^*$ for each $a \in \mathbb{A}$.

A binary relation $R \subseteq S \times S$ is a *branching bisimulation* if R is symmetric and if for each $(s, t) \in R$ the following two conditions hold: (i) if $s \xrightarrow{\tau} s'$, then $(s', t) \in R$ and (ii) if $s \xrightarrow{a} s'$ for some $a \in \mathbb{A}$, then there is $t \overset{\tau}{\Longrightarrow} t' \xrightarrow{a} t'' \overset{\tau}{\Longrightarrow} t'''$ such that $(s, t'), (s', t''), (s', t''') \in R$. We say that s is *branching bisimilar* to t (abbreviated by $s \simeq t$) whenever there is a branching bisimulation R such that $(s, t) \in R$.

A binary relation $R \subseteq S \times S$ is a *weak bisimulation* if R is symmetric and for each $(s, t) \in R$ and for each $a \in \mathbb{A}_\tau$ we have that if $s \xrightarrow{a} s'$, then there is $t \overset{a}{\Longrightarrow} t'$ such that $(s', t') \in R$. We say that s is *weakly bisimilar* to t (abbreviated by $s \approx t$) whenever there is a weak bisimulation R such that $(s, t) \in R$.

Each of the three introduced bisimulation notions can be generalized between states s_1 and s_2 where s_1 (resp. s_2) is a state of some transition system $\mathcal{T}_1$ (resp. $\mathcal{T}_2$), by simply taking the disjoint union of $\mathcal{T}_1$ and $\mathcal{T}_2$.

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be classes of transition systems and let $\equiv \in \{\sim, \simeq, \approx\}$ be some notion of equivalence. We write $\mathcal{C}_1 \leq_\equiv \mathcal{C}_2$ if for every pointed transition system $(\mathcal{T}_1, s_1)$ with $\mathcal{T}_1 \in \mathcal{C}_1$ there exists some pointed transition system $(\mathcal{T}_2, s_2)$ with $\mathcal{T}_2 \in \mathcal{C}_2$ such that $s_1 \equiv s_2$. We write $\mathcal{C}_1 \equiv \mathcal{C}_2$ in case $\mathcal{C}_1 \leq_\equiv \mathcal{C}_2$ and $\mathcal{C}_2 \leq_\equiv \mathcal{C}_1$.

These above-mentioned equivalences can also be characterized by the standard Attacker-Defender game, see e.g. [13] and the references therein.

Ranked trees: Let $\preceq$ denote the prefix order on $\mathbb{N}^*$, i.e. $x \preceq y$ for $x, y \in \mathbb{N}^*$ if there is some $z \in \mathbb{N}^*$ such that $y = xz$, and $x \prec y$ if $x \preceq y$ and $x \neq y$. A *ranked alphabet* is a collection of finite and pairwise disjoint alphabets $A = (A_i)_{i \in [0,k]}$ for some $k \geq 0$. For simplicity we identify A with $\bigcup_{i \in [0,k]} A_i$. A *ranked tree* (over the ranked alphabet A) is a mapping $t : D_t \rightarrow A$, where $D_t \subseteq [1,k]^*$ satisfies the following: D_t is non-empty, finite and prefix-closed and for each $x \in D_t$ with $t(x) \in A_i$ we have $x1, \ldots, xi \in D_t$ and $xj \notin D_t$ for each $j > i$. We say that D_t is the *domain* of t – we call these elements *nodes*. A *leaf* is a node x with $t(x) \in A_0$. We also refer to $\varepsilon \in D_t$ as the *root* of t. By Trees_A we denote the set of all ranked trees over the ranked alphabet A. We also use the usual term representation of trees, e.g. if t is a tree with root a and left (resp. right) subtree t_1 (resp. t_2) we have $t = a(t_1, t_2)$.

Let t be a ranked tree and let x be a node of t. We define $xD_t = \{xy \in [1,k]^* \mid y \in D_t\}$ and $x^{-1}D_t = \{y \in [1,k]^* \mid xy \in D_t\}$. By $t^{\downarrow x}$ we denote the *subtree of t* with root x, i.e. the tree with domain $D_{t^{\downarrow x}} = x^{-1}D_t$ defined as $t^{\downarrow x}(y) = t(xy)$. Let $s, t \in \mathsf{Trees}_A$ and let x be a node of t. We define $t[x/s]$ to be the tree that is obtained by replacing $t^{\downarrow x}$ in t by s; more formally $D_{t[x/s]} = (D_t \setminus xD_{t^{\downarrow x}}) \cup xD_s$ with $t[x/s](y) = t(y)$ if $y \in D_t \setminus xD_{t^{\downarrow x}}$ and $t[x/s](y) = s(z)$ if $y = xz$ with $z \in D_s$.
Define $|t| = |D_t|$ as the number of nodes in a tree t.

Regular tree languages: A *nondeterministic tree automaton (NTA)* is a tuple $\mathcal{A} = (Q, F, A, \Delta)$, where Q is a finite set of *states*, $F \subseteq Q$ is a set of *final states*, $A = (A_i)_{i \in [0,k]}$ is a ranked alphabet, and $\Delta \subseteq \bigcup_{i \in [0,k]} Q^i \times A_i \times Q$ is the *transition relation*. A *run* of $\mathcal{A}$ on some tree $t \in \mathsf{Trees}_A$ is a mapping $\rho : D_t \rightarrow Q$ such that for each $x \in D_t$ with $t(x) \in A_i$ we have $(\rho(x1), \ldots, \rho(xi), t(x), \rho(x)) \in \Delta$. We say ρ is *accepting* if $\rho(\varepsilon) \in F$. By $L(\mathcal{A}) = \{t \in \mathsf{Trees}_A \mid$ there is an accepting run of $\mathcal{A}$ on $t\}$ we denote the *language* of $\mathcal{A}$. A set of trees $U \subseteq \mathsf{Trees}_A$ is *regular* if $U = L(\mathcal{A})$ for some NTA $\mathcal{A}$. The *size* of an NTA $\mathcal{A}$ is defined as $|\mathcal{A}| = |Q| + |A| + |\Delta|$.

3 The Models

3.1 Mayr's PRS Hierarchy

In the following, let us fix a countable set of process constants (a.k.a. process variables) $\mathbb{X} = \{A, B, C, D, \ldots\}$. The set of *process terms* is given by the following grammar, where X ranges over $\mathbb{X}$:

$$t, u \quad ::= \quad 0 \quad | \quad X \quad | \quad t.u \quad | \quad t \| u$$

The operator . is said to be *sequential composition*, while the operator $\|$ is referred to as *parallel composition*. In order to minimize clutters, we assume that both operators . and $\|$ are left-associative, e.g., $X_1.X_2.X_3.X_4$ stands for $((X_1.X_2).X_3).X_4$. The *size* $|t|$ of a term is defined as usual. Mayr distinguishes the following classes of process terms:

$\mathbb{1}$ Terms consisting of a single constant $X \in \mathbb{X}$.
$\mathbb{S}$ Process terms without any occurrence of parallel composition.
$\mathbb{P}$ Process terms without any occurrence of sequential composition.
$\mathbb{G}$ Arbitrary process terms possibly with sequential or parallel compositions.

By $\mathbb{1}(\Sigma)$, $\mathbb{S}(\Sigma)$, $\mathbb{P}(\Sigma)$, respectively $\mathbb{G}(\Sigma)$ we denote the set $\mathbb{1}$, $\mathbb{S}$, $\mathbb{P}$, respectively $\mathbb{G}$ restricted to process constants from Σ, for each finite subset $\Sigma \subseteq \mathbb{X}$.

A *process rewrite system (*PRS*)* is a tuple $\mathcal{P} = (\Sigma, \mathbb{A}, \Delta)$, where $\Sigma \subseteq \mathbb{X}$ is a finite set of process constants, $\mathbb{A} \subseteq \mathsf{Act}$ is a finite set of action labels, and Δ is a finite set of rewrite rules of the form $t_1 \mapsto_a t_2$, where $t_1 \in \mathbb{G}(\Sigma) \setminus \{0\}$, $t_2 \in \mathbb{G}(\Sigma)$ and $a \in \mathbb{A}$. Other models in PRS hierarchy are Finite Systems (FIN), Basic Process Algebra (BPA), Basic Parallel Processes (BPP), Pushdown Systems (PDS), Petri Nets (PN), PA-processes (PA), PAD-processes (PAD), and PAN-processes (PAN). They can be defined by restricting the terms that are allowed on the left/right hand side of the PRS rewrite rules as specified in the following tables.

Model	L.H.S.	R.H.S
FIN	$\mathbb{1}(\Sigma)$	$\mathbb{1}(\Sigma)$
BPA	$\mathbb{1}(\Sigma)$	$\mathbb{S}(\Sigma)$
BPP	$\mathbb{1}(\Sigma)$	$\mathbb{P}(\Sigma)$

Model	L.H.S.	R.H.S
PDS	$\mathbb{S}(\Sigma)$	$\mathbb{S}(\Sigma)$
PN	$\mathbb{P}(\Sigma)$	$\mathbb{P}(\Sigma)$

Model	L.H.S.	R.H.S
PAD	$\mathbb{S}(\Sigma)$	$\mathbb{G}(\Sigma)$
PAN	$\mathbb{P}(\Sigma)$	$\mathbb{G}(\Sigma)$

We follow the approach of [16,6] to define the semantics of PRS. While Mayr [18] directly works on the equivalence classes of terms (induced by some equivalence relation $\equiv$ defined by some axioms including associativity and commutativity of $\|$) to define the dynamics of PRS, we shall initially work on term level. More precisely, given a PRS $\mathcal{P} = (\Sigma, \mathbb{A}, \Delta)$, we write $\mathcal{T}_0(\mathcal{P})$ to denote the transition system $(\mathbb{G}(\Sigma), \mathbb{A}, \{\xrightarrow{a} \mid a \in \mathbb{A}\})$ where $\xrightarrow{a}$ is defined by the following rules:

$$\frac{t_1 \xrightarrow{a} t_1'}{t_1\|t_2 \xrightarrow{a} t_1'\|t_2} \qquad \frac{t_2 \xrightarrow{a} t_2'}{t_1\|t_2 \xrightarrow{a} t_1\|t_2'} \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2} \qquad \frac{}{u \xrightarrow{a} t}\ (u \mapsto_a t) \in \Delta$$

We now define Mayr's semantics of PRS in terms of $\mathcal{T}_0(\mathcal{P})$. First of all, let us define the equivalence relation $\equiv$ on terms using the following proof rules:

$$\frac{}{t.0 \equiv t}\ R0. \qquad \frac{}{t_1.(t_2.t_3) \equiv (t_1.t_2).t_3}\ A. \qquad \frac{t_1 \equiv u_1 \qquad t_2 \equiv u_2}{t_1.t_2 \equiv u_1.u_2}\ \text{Con.}$$

$$\frac{}{0.t \equiv t}\ L0. \qquad \frac{}{t_1\|(t_2\|t_3) \equiv (t_1\|t_2)\|t_3}\ A\| \qquad \frac{t_1 \equiv u_1 \qquad t_2 \equiv u_2}{t_1\|t_2 \equiv u_1\|u_2}\ \text{Con}\|$$

$$\frac{}{t\|0 \equiv t}\ R0\| \qquad \frac{}{t_1\|t_2 \equiv t_2\|t_1}\ C\| \qquad \frac{u \equiv u' \qquad u' \equiv u''}{u \equiv u''}\ \text{Trans}$$

$$\frac{}{0\|t \equiv t}\ L0\| \qquad \frac{}{u \equiv u}\ \text{Ref} \qquad \frac{t \equiv u}{u \equiv t}\ \text{Sym}$$

Here, u, t, t_i, u_i range over all terms in $\mathbb{G}$. Intuitively, the axioms defining $\equiv$ say that 0 is *identity*, while the operator . (resp. $\|$) is associative (resp. associative and commutative). The rules (Con.) and (Con$\|$) are standard *context rules* in process algebra saying that term equivalence is preserved under substitutions of equivalent subterms. Finally, Trans, Sym, and Ref state that $\equiv$ is an equivalence relation. In the sequel, we also use the symbol $\equiv_1$ to denote the equivalence relation on process terms that allows all the above axioms except for (A$\|$) and (C$\|$). Obviously, $\equiv_1 \subseteq \equiv$. Given a term $t \in \mathbb{G}$, we denote by $[t]_\equiv$ (resp. $[t]_{\equiv_1}$) the $\equiv$-class (resp. $\equiv_1$-class) containing t.

Mayr's semantics on a PRS $\mathcal{P} = (\Sigma, \mathbb{A}, \Delta)$ such that $\mathcal{T}_0(\mathcal{P}) = (\mathbb{G}(\Sigma), \mathbb{A}, \{\xrightarrow{a} \mid a \in \mathbb{A}\})$ is a transition system $\mathcal{T}(\mathcal{P}) = (S, \mathbb{A}, \{E_a \mid a \in \mathbb{A}\})$, where $S = \{[t]_\equiv \mid t \in \mathbb{G}(\Sigma)\}$ and where $(C, C') \in E_a$ iff there exist $t \in C$ and $t' \in C'$ such that $t \xrightarrow{a} t'$. An important result by Mayr [18] is that the PRS hierarchy is strict with respect to strong bisimulation.

3.2 (Regular) Ground Tree Rewrite Systems and Prefix-Recognizable Systems

A *regular ground tree rewrite system* (RGTRS) is a tuple $\mathcal{R} = (A, \mathbb{A}, R)$, where A is a ranked alphabet, $\mathbb{A} \subseteq \mathsf{Act}$ is a finite set of action labels and where R is finite set of rewrite rules $L \overset{a}{\hookrightarrow} L'$, where L and L' are regular tree languages over A given as NTA. The transition system defined by $\mathcal{R}$ is $\mathcal{T}(\mathcal{R}) = (\mathsf{Trees}_A, \mathbb{A}, \{\xrightarrow{a} \mid a \in \mathbb{A}\})$, where for each $a \in \mathbb{A}$, we have $t \xrightarrow{a} t'$ if and only if there is some $x \in D_t$ and some rule $L \overset{a}{\hookrightarrow} L' \in R$ such that $t^{\downarrow x} = s$ and $t' = t[x/s']$ for some $s \in L$ and some $s' \in L'$.

A *ground tree rewrite system* (GTRS) is an RGTRS $\mathcal{R} = (A, \mathbb{A}, R)$, where for each $L \overset{a}{\hookrightarrow} L' \in R$ we have that both $L = \{t\}$ and $L' = \{t'\}$ is a singleton; we also write $t \overset{a}{\hookrightarrow} t' \in R$ for this.

A *prefix-recognizable system* (PREF) is an RGTRS $\mathcal{R} = (A, \mathbb{A}, R)$, where only A_0 and A_1 may be non-empty. We note that analogously pushdown systems can be defined as GTRS $\mathcal{R} = (A, \mathbb{A}, R)$, where only A_0 and A_1 may be non-empty.

4 Containment Results

In this section, we prove the following containment results: $\mathsf{PAD} \leq_{\simeq} \mathsf{GTRS}$ (Section 4.1), $\mathsf{BPP} \leq_{\sim} \mathsf{GTRS}$ and $\mathsf{GTRS} \leq_{\sim} \mathsf{PRS}$, and finally $\mathsf{RGTRS} =_{\simeq} \mathsf{GTRS}$ (Section 4.2).

4.1 $\mathsf{PAD} \leq_{\simeq} \mathsf{GTRS}$

Theorem 1 ($\mathsf{PAD} \leq_{\simeq} \mathsf{GTRS}$)**.** *Given a* PAD $\mathcal{P} = (\Sigma, \mathbb{A}, \Delta)$ *and a term* $t_0 \in \mathbb{G}(\Sigma)$*, there exists a* GTRS $\mathcal{R} = (A, \mathbb{A}_\tau, R)$ *and a tree* $t_0' \in \mathsf{Trees}_A$ *such that* $(\mathcal{T}(\mathcal{P}), [t_0]_\equiv)$ *is branching bisimilar to* $(\mathcal{T}(\mathcal{R}), t_0')$*. Furthermore,* $\mathcal{R}$ *and* t_0' *may be computed in time polynomial in* $|\mathcal{P}| + |t_0|$*.*

Before proving this theorem, we shall first present the general proof strategy. The main difficulty of the proof is that the domain S' of $\mathcal{T}(\mathcal{P})$ consists of $\equiv$-classes of process terms, while the domain of $\mathcal{T}(\mathcal{R})$ consists of ranked trees. On the other hand, observe that the other semantics $\mathcal{T}_0(\mathcal{P})$ is more close to a GTRS since the domain S of $\mathcal{T}_0(\mathcal{P})$ consists of process terms (not equivalence classes thereof). Therefore, the first hurdle in the proof is to establish a connection between $\mathcal{T}(\mathcal{P})$ and $\mathcal{T}_0(\mathcal{P})$. To this end, we will require that t_0 and all process terms in $\mathcal{P}$ have a minimum number of zeros and have no right-associative occurrence of the sequential composition operator. We will then pick a small subset of the axioms of $\equiv$ as τ-transitions, which we will add to $\mathcal{T}_0(\mathcal{P})$. These axioms include those that reduce the occurrences of 0 from terms, and the rule that turns a right-associative occurrence of the sequential composition operator into a left-associative occurrence. The resulting pointed transition system $(\mathcal{T}_0(\mathcal{P}), t_0)$ will become branching bisimilar to $(\mathcal{T}(\mathcal{P}), [t_0]_\equiv)$. In fact, fixing t_0 as the initial configuration, we will see that further restrictions to the axioms for $\equiv$ (e.g. associativity of .) may be made resulting in a pointed transition system that can be easily captured in the framework of GTRS.

Adding the τ-transitions to $\mathcal{T}_0(\mathcal{P})$: We define the relation $\xrightarrow{\tau}$ on arbitrary process terms given by the following proof rules:

$$\begin{array}{|lll|}
\hline
\dfrac{}{0.t \xrightarrow{\tau} t} & \dfrac{}{t\|0 \xrightarrow{\tau} t} & \dfrac{t_1 \xrightarrow{\tau} t_1'}{t_1.t_2 \xrightarrow{\tau} t_1'.t_2} \\
\dfrac{}{t.0 \xrightarrow{\tau} t} & \dfrac{}{t_1.(t_2.t_3) \xrightarrow{\tau} (t_1.t_2).t_3} & \dfrac{t_2 \xrightarrow{\tau} t_2'}{t_1\|t_2 \xrightarrow{\tau} t_1\|t_2'} \\
\dfrac{}{0\|t \xrightarrow{\tau} t} & \dfrac{t_1 \xrightarrow{\tau} t_1'}{t_1\|t_2 \xrightarrow{\tau} t_1'\|t_2} & \dfrac{t_2 \xrightarrow{\tau} t_2'}{t_1.t_2 \xrightarrow{\tau} t_1.t_2'} \\
\hline
\end{array}$$

Here, t is allowed to be any process term. Observe that these τ-transitions remove redundant occurrences of 0 and turns a right-associative occurrence of the sequential composition into a left-associative one. Observe that we do not allow associativity/commutativity axioms for $\|$ in our definition of $\xrightarrow{\tau}$. It is easy to see that $\xrightarrow{\tau} \subseteq \equiv_1 \subseteq \equiv$. We now note a few simple facts about $\xrightarrow{\tau}$ in the following lemmas.

Lemma 2. *For all terms t, there exists a unique term $t_\downarrow$ such that $t \xrightarrow{\tau}^* t_\downarrow$ and $t_\downarrow \not\xrightarrow{\tau}$. Furthermore, all paths from t to $t_\downarrow$ are of length at most $O(|t|^2)$, and moreover $t_\downarrow$ is computable from t in polynomial time.*

Lemma 3. *The following statements hold: (1) If $t \equiv_1 t'$, then $t_\downarrow = t'_\downarrow$, (2) If $0 \equiv v$, then $v \xrightarrow{\tau}^* 0$, and (3) If $X_1.X_2 \ldots X_n \equiv v$, then $v \xrightarrow{\tau}^* X_1.X_2 \ldots X_n$.*

Lemma 2 is a basic property of a rewrite system commonly known as *confluence* and *termination* (e.g. see [1]). In fact, it does not take long to terminate. Lemma 3 gives the form of the unique "minimal" term with respect to $\xrightarrow{\tau}$ given various different initial starting points. The proofs of these lemmas are standard. For the rest of the proof of Theorem 1, we assume the following conventions:

Convention 4 *The term t_0 and all process terms in $\mathcal{P}$ are minimal with respect to $\xrightarrow{\tau}$. That is, each of such terms t satisfies $t = t_\downarrow$.*

We now add these τ-transitions into $\mathcal{T}_0(\mathcal{P})$. So, we will write $\mathcal{T}_0(\mathcal{P}) = (\mathbb{G}(\Sigma), \mathbb{A}_\tau, \{\xrightarrow{a}: a \in \mathbb{A}_\tau\})$. Our first technical result is that the equivalence relation $\equiv$ is indeed a branching bisimulation on $\mathcal{T}_0(\mathcal{P})$.

Lemma 5. *$\equiv$ is a branching bisimulation on $\mathcal{T}_0(\mathcal{P})$.*

The proof of this lemma is not difficult but tedious. As an immediate corollary, we obtain that $(\mathcal{T}_0(\mathcal{P}), t_0)$ is equivalent to $(\mathcal{T}(\mathcal{P}), [t_0]_\equiv)$ up to branching bisimulation.

Corollary 6. *The relation $R = \{(C, t) \subseteq S' \times S : t \in C\}$ is a branching bisimulation between $\mathcal{T}(\mathcal{P})$ and $\mathcal{T}_0(\mathcal{P})$.*

Removing complex τ-transitions: Corollary 6 implies that we may restrict ourselves to the transition system $\mathcal{T}_0(\mathcal{P})$. At this stage, our τ-transitions still contain some rules that cannot easily be captured in the framework of GTRS, e.g., left-associativity rule of the sequential composition. We will now show that fixing an initial configuration t_0 allows us to remove these τ-transitions from our systems.

Recall that our initial configuration t_0 satisfies $t_0 = (t_0)_\downarrow$. Denote by W the set of all subtrees (either of t_0 or of a left/right side of a rule in $\mathcal{P}$) rooted at a node that is a right child of a .-labeled node. It is easy to see that Convention 4 implies that each $t \in W$ satisfies $t = t_\downarrow$. Consequently, each $t \in W$ cannot be of the form $t_1.t_2$ or 0 since t is a right child of the sequential composition. Furthermore, $|W|$ is linear in the size of $\mathcal{P}$.

Lemma 7. *Fix a term $t \in post^*(t_0)$ with respect to $\mathcal{T}_0(\mathcal{P})$. Then, any subtree of t which is a right child of a .-labeled node is in W.*

This lemma can be easily proved by induction on the length of the witnessing path that $t \in post^*(t_0)$ and that this invariant is always satisfied. This lemma implies that some of the rules for defining $\xrightarrow{\tau}$ may be restricted when only considering $post^*(t_0)$ as the domain of our system, resulting in the following simplified definition:

$$\begin{array}{lll}
\dfrac{}{0.t \xrightarrow{\tau} t}\; t \in W & \dfrac{}{t\|0 \xrightarrow{\tau} t} & \dfrac{t_1 \xrightarrow{\tau} t_1'}{t_1.t_2 \xrightarrow{\tau} t_1'.t_2}\; t_2 \in W \\[2ex]
\dfrac{}{0\|t \xrightarrow{\tau} t} & \dfrac{t_2 \xrightarrow{\tau} t_2'}{t_1\|t_2 \xrightarrow{\tau} t_1\|t_2'} & \dfrac{t_1 \xrightarrow{\tau} t_1'}{t_1\|t_2 \xrightarrow{\tau} t_1'\|t_2}
\end{array}$$

Observe that the rule $t.0 \xrightarrow{\tau} t$ may be omitted since no subtree of $t \in post^*(t_0)$ of the form $u.0$ exists. Moreover, the rule $t_1.(t_2.t_3) \xrightarrow{\tau} (t_1.t_2).t_3$ is never applicable since no subtree of $t \in post^*(t_0)$ of the form $t_1.(t_2.t_3)$ exists. Other rules are omitted because any subtree of t of the form $t_1.t_2$ must satisfy $t_2 \in W$, and that each $u \in W$ satisfies $u = u_\downarrow$ (which implies $u \not\xrightarrow{\tau}$).

Finally, in order to cast the system into GTRS framework, we will further restrict rules of the form $t\|0 \xrightarrow{\tau} t$ or $0\|t \xrightarrow{\tau} t$. Let l-prefix($\mathcal{P}$) be the set of all prefixes of words w appearing on the left hand side of the rules in $\mathcal{P}$ treated as left-associative terms. More formally, l-prefix($\mathcal{P}$) contains 0 (a term representation of the empty word) and all subterms u of a term appearing on the left hand side of a rule in $\mathcal{P}$ rooted at a node location of the form 1^*. We define $\rightsquigarrow_\tau$ to be the restriction of $\xrightarrow{\tau}$, where rules of the form $0\|t \xrightarrow{\tau} t$ and $t\|0 \xrightarrow{\tau} t$ are restricted to $t \in$ l-prefix($\mathcal{P}$). We let $\mathcal{T}_0'(\mathcal{P})$ to be $\mathcal{T}_0(\mathcal{P})$ with $\xrightarrow{\tau}$ replaced by $\rightsquigarrow_\tau$.

Lemma 8. *$(\mathcal{T}_0'(\mathcal{P}), t)$ is branching bisimilar to $(\mathcal{T}_0(\mathcal{P}), t)$.*

Constructions of the GTRS: It is now not difficult to cast $\mathcal{T}_0'(\mathcal{P})$ into GTRS framework. To construct the GTRS, we let A be the ranked alphabet containing: (i) a nullary symbol for each process variable occuring in $\mathcal{P}$, (ii) a binary symbol for the binary operator $\|$, and (iii) a unary symbol $\hat{t}$ for each term $t \in W$. Since each subtree u of a tree

$t \in post^*(t_0)$ of the form $t_1.t_2$ satisfies $t_2 \in W$, we may simply substitute u with the tree $\hat{t_2}(t_1)$ and perform this substitution recursively on t_1. Denote by $\lambda(t)$ the resulting tree over the new alphabet A after this substitution is performed on a process term t. The desired GTRS is $\mathcal{R} = (A, \mathbb{A}_\tau, R)$, where R is defined as follows. For each rule $t \mapsto_a t'$ in $\mathcal{P}$, where $a \in \mathbb{A}$, we add the rule $\lambda(t) \overset{a}{\hookrightarrow} \lambda(t')$ to R. For each $t \in$ 1-prefix$(\mathcal{P})$, we add $0\|t \overset{\tau}{\hookrightarrow} t$ and $t\|0 \overset{\tau}{\hookrightarrow} t$ to R. Finally, we add the transition rule $\hat{t}(0) \xrightarrow{\tau} t$ for each $t \in W$. It is now not difficult to show that $(\mathcal{T}_0'(\mathcal{P}), t) \simeq (\mathcal{T}(\mathcal{R}), \lambda(t))$, which immediately implies Theorem 1.

4.2 Further Containment Results

Theorem 9. BPP $\leq_\sim$ GTRS.

Proof (sketch). The idea is to construct from some BPP a GTRS, where each leaf corresponds to a process constant. A leaf is either marked or unmarked. An unmarked leaf X can become marked with the fresh symbol \$ via the action a in case the rule $X \mapsto_a 0$ is present in the BPP. Rules of the kind $X \mapsto_a Y_1\| \ldots \|Y_n$ are realized via $X \overset{a}{\hookrightarrow} \bullet(Y_1, \ldots, Y_n)$ in the GTRS. Moreover the GTRS does not contain any rules, where a marked leaf is on the left-hand side of a rule. □

Theorem 10. GTRS $\leq_\sim$ PRS.

Proof (sketch). Let k be the maximal rank of the alphabet of some GTRS. Although parallel composition is interpreted commutatively we can simulate order by using k additional symbols in a PRS. □

Theorem 11. RGTRS $\simeq$ GTRS.

Proof (sketch). A GTRS can simulate via τ-transitions the bottom-up computation of an NTA. In addition, one provides τ-transitions that allow to undo these transitions. □

In analogy to Theorem 11 one can prove the following.

Corollary 12. PDS $\simeq$ PREF.

5 Separation Results

In this section, we provide the separation results in the two refined hierarchies. We first note two known separation results: (1) BPA $\not\leq_\approx$ PN (e.g. see [10]), and (2) BPP $\not\leq_\approx$ PDS since there is a BPP trace language that is not context-free (e.g. see references in [4]) and trace equivalence is coarser than weak bisimulation equivalence.

5.1 PA $\not\leq_\sim$ GTRS

Some properties of GTRS**:** We introduce some notions that were also used in [15]. Let $\mathcal{R} = (A, \mathbb{A}, R)$ be an arbitrary GTRS. For each $t \in \mathsf{Trees}_A$, we define $\mathsf{height}(t) = \max\{|x| : x \in D_t\}$. We define the number $h_\mathcal{R} = \max\{\mathsf{height}(t) \mid \exists t' \in \mathsf{Trees}_A\ \exists \sigma \in \mathbb{A} : t \overset{\sigma}{\hookrightarrow} t' \in R \text{ or } t' \overset{\sigma}{\hookrightarrow} t \in R\}$ and $|\mathcal{R}| = |A| + |\mathbb{A}| + \sum_{t \overset{\sigma}{\hookrightarrow} t' \in R} |t| + |t'|$.

Lemma 13. *Let $\Lambda \subseteq \mathbb{A}$. For every $t_0 \in \mathsf{Trees}_A$ there is some $N = \exp(|\mathcal{R}| + height(t_0))$ such that $t_0 \xrightarrow{\Lambda}{}^{N}$ implies $t_0 \xrightarrow{\Lambda}{}^{n}$ for infinitely many $n \in \mathbb{N}$.*

The separating PA: Consider the PA $\mathcal{P} = (\Sigma, \mathbb{A}, \Delta)$ with $\Sigma = \{A, B, C, D\}$, $\mathbb{A} = \{a, b, c, d\}$ and where Δ consists of the following rewrite rules:

$$A \mapsto_a 0 \qquad B \mapsto_b 0 \qquad C \mapsto_c 0 \qquad D \mapsto_d 0 \qquad A \mapsto_a A\|B\|C$$

For the rest of this section, we wish to prove that the state $\alpha = A.D$ in $\mathcal{T}(\mathcal{P})$ is *not* strongly bisimilar to any pointed GTRS. So for the sake of contradiction, let us assume some GTRS $\mathcal{R} = (A, \mathbb{A}, R)$ and some $t_\alpha \in \mathsf{Trees}_A(\mathcal{R})$ with $t_\alpha \sim \alpha$. We note that e.g. by [15] it is known that the set of maximal sequences executable from α (the language of α when $\mathcal{P}$ is interpreted as a language acceptor) are recognizable by some GTRS [15].

We call $U[x]$ a *context* if $U \in \mathsf{Trees}_A$ and $x \in D_U$ is a leaf of U. Given a tree $t \in \mathsf{Trees}_A$ and a context $U[x]$, we write $U[t]$ for $U[x/t]$. We define $U^n[t]$ inductively as follows: $U^0[t] = t$ and $U^n = U[U^{n-1}[t]]$ for each $n > 0$.

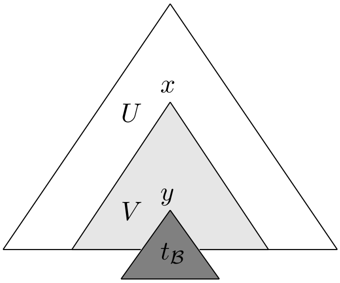


Fig. 2. The tree $T^1 = U[V[t_\mathcal{B}]]$

Let us consider $\mathsf{post}^*_{\{a\}}(t_\alpha)$. First, there is some NTA $\mathcal{A}$ with $L(\mathcal{A}) = \{t_\alpha\}$. A folklore result states that there is some NTA $\mathcal{B}$ with $L(\mathcal{B}) = \mathsf{post}^*_{\{a\}}(L(\mathcal{A})) = \mathsf{post}^*_{\{a\}}(t_\alpha)$, see e.g. [15]. Note that $L(\mathcal{B})$ is infinite since α can reach infinitely many pairwise non-bisimilar states and $t_\alpha \sim \alpha$ by assumption. By applying the Pumping Lemma for regular tree languages, there is some tree $t_\mathcal{B} \in \mathsf{Trees}_A$ and there are contexts $U[x], V[y] \in \mathsf{Trees}_A$ such that (i) $U[V[t_\mathcal{B}]] \in L(\mathcal{B})$, (ii) $\text{height}(U[V[t_\mathcal{B}]]) \leq 2 \cdot |\mathcal{B}|$, (iii) $\text{height}(V[t_\mathcal{B}]) \leq |\mathcal{B}|$, (iv) $|y| > 0$, i.e. V is not a singleton tree, and (v) $U[V^n[t_\mathcal{B}]] \in L(\mathcal{B})$ for each $n \geq 0$.

The tree $U[V[t_\mathcal{B}]]$ is displayed in Figure 2. We define the tree $T^n = U[V^n[t_\mathcal{B}]]$ for each $n \geq 0$. Moreover we define the consant $\gamma = \ell \cdot (h_\mathcal{R} + 1)$ with $\ell = 2^{|\{t \in \mathsf{Trees}_A | \text{height}(t) \leq h_\mathcal{R}\}|}$, i.e. ℓ denotes the number of different subsets of the set of all trees in Trees_A of height at most $h_\mathcal{R}$.

The following lemma states that if $V^\gamma[t_\mathcal{B}]$ can reach some tree of height at most $h_\mathcal{R}$ by only executing the action σ, then there is already some tree t_σ of height at most h_σ such that for all $i \geq 0$ we have $V^{\theta+i\cdot\delta}[t_\mathcal{B}] \xrightarrow[\mathcal{R}]{\sigma}{}^{*} t_\sigma$.

Lemma 14. *There exist $\theta, \delta \geq 1$ such that if $V^\gamma[t_\mathcal{B}] \xrightarrow{\sigma}{}^{*} t$ for some $t \in \mathsf{Trees}_A$ with height$(t) \leq h_\mathcal{R}$, then $V^{\theta+i\cdot\delta}[t_\mathcal{B}] \xrightarrow[\mathcal{R}]{\sigma}{}^{*} t_\sigma$ for all $i \geq 0$ for some $t_\sigma \in \mathsf{Trees}_A$.*

For the rest of this section, we fix θ and δ from Lemma 14. Note that due to $t_\alpha \sim \alpha$ we have that for every $t \in \mathsf{post}^*_{\{a\}}(t_\alpha)$ there is some unique $k \in \mathbb{N}$ with $t_\alpha \xrightarrow{a}{}^{k} t$. Thus, for each tree $t \in \mathsf{post}^*_{\{a\}}(t_\alpha)$ we define $k(t)$ to be the *unique* k with $t_\alpha \xrightarrow{a}{}^{k} t$.

Lemma 15. $\{k(T^n) \mid n \in \mathbb{N}\}$ *is an infinite set.*

Let us immediately apply Lemma 15. Let us fix some residue class r modulo δ such that there are infinitely many n with $n \equiv r \bmod \delta$ all having pairwise distinct $k(T^n)$ values. Among these infinitely many n we will choose a sufficiently large $N \geq \theta$ for the following arguments to work. The tree T_N is depicted in Figure 3. Recall that by definition $T^N \in \mathsf{post}^*_{\{a\}}(t_\alpha)$.

The following lemma states that one can never shrink the subtree $V^\gamma[t_\mathcal{B}]$ of T^N to some tree of height at most $h_\mathcal{R}$ by only executing b's or only executing c's.

Lemma 16. *If* $V^\gamma[t_\mathcal{B}] \xrightarrow{\sigma}{}^* t$*, then we have* height$(t) > h_\mathcal{R}$ *for each* $t \in \mathsf{Trees}_A$ *and each* $\sigma \in \{b, c\}$.

Let y_N denote the unique node of T^N where the subtree $t_\mathcal{B}$ is rooted at. We call a node $z \in D_{T^N}$ of T^N *off-path* if $z \not\preceq y_N$. For each tree $t \in \mathsf{Trees}_A$ and each $\sigma \in \mathbb{A}$, we define $\sup_\sigma(t) = \sup\left\{j \in \mathbb{N} \mid t \xrightarrow{\sigma}{}^j\right\}$.

Intuitively speaking, the following lemma states that from the subtree $V^\gamma[t_\mathcal{B}]$ of T^N and subtrees of T^N that are rooted at off-path nodes one only execute a constantly long sequences from b^*c^* or from c^*b^* (unless $t_\alpha \sim \alpha$ is violated). Let us define $\overline{b} = c$ and $\overline{c} = b$. We note that γ and $\mathcal{B}$ only depend on $\mathcal{R}$ and on t_α but not on N.

Lemma 17. *Let* $\sigma \in \{b, c\}$*. Then there is some constant* $J = J(\mathcal{R}, t_\alpha)$ *such that* $\sup_\sigma(t) \leq J$ *whenever either* $t = V^\gamma[t_\mathcal{B}]$ *or* $T^{N\downarrow z} \xrightarrow{\overline{\sigma}}{}^* t$ *for some off-path* z.

We can now prove the main result of this section.

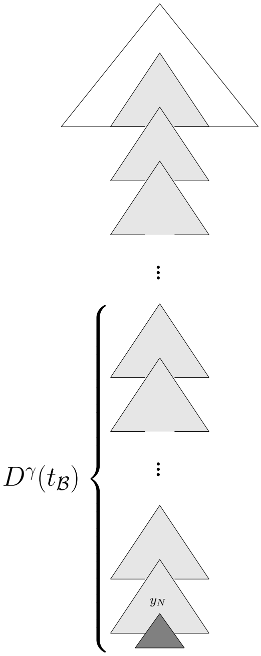


Fig. 3. The tree T^N

Theorem 18. $\mathsf{PA} \not\leq_\sim \mathsf{GTRS}$.

Proof. We give a simple winning strategy for Attacker that contradicts $t_\alpha \sim \alpha$. First Attacker plays $t_\alpha \xrightarrow{a}{}^{k(T^N)} T^N$. We remark since N is chosen sufficiently large, it follows that $k(T_N)$ is sufficiently large for the following arguments to work. It has to hold for some $s \in \{0, 1\}$

$$T^N \quad \sim \quad \left(A^{1-s} \| \underbrace{B\|B \cdots \|B}_{k(T^N)-s} \| \underbrace{C\|C \cdots \|C}_{k(T^N)-s} \right).D \qquad (\bigstar)$$

We only treat the case $s = 1$ (the case $s = 0$ can be proven analogously). Recall that γ is a constant that only depends on $\mathcal{R}$ and t_α. On the one hand we cannot modify the subtree $V^\gamma[t_\mathcal{B}]$ of T^N to any tree of height at most $h_\mathcal{R}$ by executing b's only by Lemma 16. On the other hand we cannot execute more than J many b's from the subtree $V^\gamma[t_\mathcal{B}]$, where J is the constant of Lemma 17. Thus, since $T^N \xrightarrow{b}{}^{k(T^N)-1}$ holds, Attacker can play $k(T^N) - 1 - J$ many b's outside the subtree $V^\gamma[t_\mathcal{B}]$. We recall that $k(T^N) - J$ can be arbitrarily large since J is a constant that only depends on $\mathcal{R}$ and t_α. By definition of T^N all of these $k(T^N) - 1 - J$ many b's can be played on subtrees initially rooted at off-path nodes of T^N *outside* the subtree $V^\gamma[t_\mathcal{B}]$. However from each of these subtrees that are initially rooted at off-path nodes outside the subtree $V^\gamma[t_\mathcal{B}]$, we can execute at most J many b's.

Analogously Attacker can execute $k(T^N) - 1 - J$ many c's from T^N all on subtrees initially rooted at off-path nodes of T_N *outside* the subtree $V^\gamma[t_\mathcal{B}]$.

Attacker now has the following winning strategy. First he plays $k(T^N) - 1 - J$ many b's on subtrees rooted at off-path nodes of T^N *outside* $V^\gamma[t_\mathcal{B}]$. After playing these b's the height each of these subtrees is bounded by a constant that only depends on $\mathcal{R}$ and t_α by Lemma 17. Next, Attacker plays $k(T^N) - 1 - J$ many c's at positions outside the subtree $V^\gamma[t_\mathcal{B}]$ and still, by Lemma 17, the height of all subtrees rooted at off-path nodes outside $V^\gamma[t_\mathcal{B}]$ have a height bounded by a constant that only depends on $\mathcal{R}$ and t_α. Let us call the resulting tree T'. We note that $T' \xrightarrow{b^J c^J d}$, i.e. from T' the sequence $b^J c^J$ is executable thus reaching a tree where a d-labeled rule is executable. But this implies that $T^N \xrightarrow{wd}$ for some $w \in \{b, c\}^*$ where $|w|$ is bounded by a constant that only depends on $\mathcal{R}$ and t_α, clearly contradicting (★). □

5.2 GTRS $\not\leq_\approx$ PAD

By Theorem 11 it suffices to prove that there is some RGTRS that is not weakly bisimilar to any PAD.

Consider the RGTRS $\mathcal{R} = (A, \mathbb{A}, R)$, with $A_0 = \{X_0, Y_0, Z_0\}$, $A_1 = \{X_1, Y_1\}$, $A_2 = \{\bullet\}$, and $\mathbb{A} = \{a, b, c, d, e, f\}$. First, we add to R the following singleton rewrite rules: (i) $X_0 \overset{a}{\hookrightarrow} X_1(X_0)$, (ii) $X_1(X_0) \overset{b}{\hookrightarrow} X_0$, (iii) $Y_1 \overset{c}{\hookrightarrow} Y_1(Y_0)$, (iv) $Y_1(Y_0) \overset{d}{\hookrightarrow} Y_0$, and (v) $\bullet(X_0, Y_0) \overset{e}{\hookrightarrow} Z_0$.

We note that so far all rewrite rules are standard ground tree rewrite rules. Also note that the singleton tree Z_0 is a dead-end. It is easy to see that for every tree in $t \in \mathsf{Trees}_A$ that is reachable from $\bullet(X_0, Y_0)$ we have $t = Z_0$ or t is of the form $t = \bullet(t_X, t_Y)$, where $t_X = X_1^m[X_0]$ and $t_Y = Y_1^n[Y_0]$ for some $m, n \geq 0$. In the latter case we denote t by $t(m, n)$. Finally, we add to R the regular tree rewrite rule $\{t(m, n) \mid n \geq 1 \text{ or } m \geq 1\} \overset{f}{\hookrightarrow} Z_0$. The transition system $\mathcal{T}(\mathcal{R})$ is depicted in Figure 4.

It is easy to see that the set of maximal sequences executable from $t(0, 0)$ is not a context-free language. We claim that there is no PAD that is weakly bisimilar to $t(0, 0) = \bullet(X_0, Y_0)$.

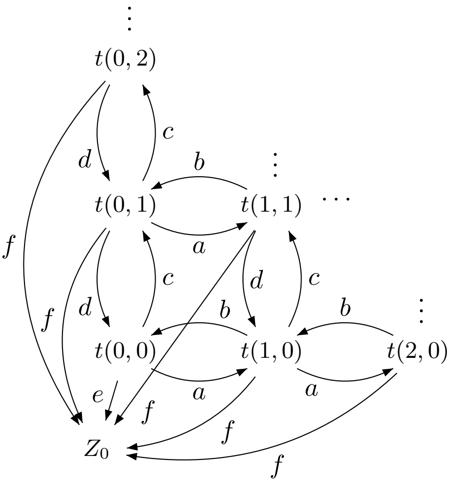


Fig. 4. The transition system $\mathcal{T}(\mathcal{R})$

Let us assume by contradiction that for some PAD $\mathcal{P} = (\Sigma, \mathcal{A}_\tau, \Delta)$ and for some term $\alpha_0 \in \mathbb{G}(\Sigma)$ we have $\alpha_0 \approx t(0,0)$. We call a term $\alpha \in \mathbb{G}(\Sigma)$ *inactive* if $\alpha \overset{\sigma}{\not\Longrightarrow}$ for all $\sigma \in \mathbb{A}$. We note that $\alpha \overset{\tau}{\Longrightarrow}$ might be possible even though α is inactive.

Lemma 19. *Assume some term α with $\alpha \approx t(m,n)$ for some $m, n \in \mathbb{N}$ and α contains an enabled subterm $\beta_1 \| \beta_2$. Then β_1 or β_2 is inactive.*

Theorem 20. GTRS $\not\leq_{\approx}$ PAD.

Proof (sketch). The proof idea is to show that any PAD that satisfies the property of Lemma 19 is already weakly bisimilar to a pushdown process.

5.3 PDS $\not\leq_{\approx}$ PAN and PN $\not\leq_{\approx}$ GTRS

Theorem 21. PDS $\not\leq_{\approx}$ PAN.

Proof (sketch). The proof idea is an adaption of an idea from [18] separating PAN from PDS with respect to strong bisimulation, but is technically more involved. The separating pushdown process behaves as follows: First, it executes a sequence of actions $w = \{a, b\}^*$ and then executes either of the following: (1) The action c, then the reverse of w and finally an e. (2) The action d, then the reverse of w and finally an f. □

Theorem 22. PN $\not\leq_{\approx}$ GTRS

The proof can be done by observing that $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is a PN language (e.g. see [22]), while this language is not a trace language of GTRS (e.g. see [15]).

6 Applications

In this section, we provide applications of the connections that we establish between GTRS and the PRS hierarchy. Instead of attempting to exhaust all possible applications, we shall only highlight a few of the key applications. In particular, Theorem 1 allows us to transfer decidability/complexity upper bounds on model checking over GTRS to model checking over PA/PAD-processes.

The first application is the decidability of EF-logic over PAD. The logic EF (e.g. see [13,23]) is the extension of Hennessy-Milner logic with reachability operators (possibly parameterized over subsets of all possible actions). The decidability of EF model checking over GTRS has been known for a long time, e.g., it follows from the results of [8,12]. Together with Theorem 1, this easily gives another proof of the following result of Mayr.

Theorem 23 ([19]). *Model checking EF-logic over* PAD *is decidable.*

The second application is the decidability/complexity of model checking the common fragments LTL$_{det}$ (called deterministic LTL) and LTL($\mathbf{F}_\mathrm{s}, \mathbf{G}_\mathrm{s}$) [7,17] of LTL over PAD. These fragments are suffciently powerful for expressing interesting properties like safety, fairness, liveness, and also some simple stuttering-invariant LTL properties. The following two theorems follow from the results for GTRS [23,24]; decidability with no upper bounds was initially proven in [7].

Theorem 24. *Model checking LTL_{det} over* PAD *is decidable in exponential time in the size of the formula and polynomial in the size of the system. Model checking $LTL(\boldsymbol{F}_s, \boldsymbol{G}_s)$ over* PAD *is decidable in time double exponential in the size of the formula and polynomial in the size of the system.*

Acknowledgment. We thank Taolue Chen, Matthew Hague, Jean Christoph Jung, Richard Mayr, Colin Stirling, and anonymous reviewers for their comments. Anthony W. Lin is supported by EPSRC (EP/H026878/1).

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 94–111. Springer, Heidelberg (1987)
3. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theor. Comput. Sci. 37, 77–121 (1985)
4. Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. In: LICS, pp. 123–133. IEEE Computer Society, Los Alamitos (1995)
5. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)
6. Bouajjani, A., Touili, T.: Reachability analysis of process rewrite systems. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 74–87. Springer, Heidelberg (2003)
7. Bozzelli, L., Kretínský, M., Rehák, V., Strejcek, J.: On decidability of LTL model checking for process rewrite systems. Acta Inf. 46(1), 1–28 (2009)
8. Brainerd, W.S.: Tree generating regular systems. Information and Control 14(2), 217–231 (1969)
9. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Handbook of Process Algebra, ch.9, pp. 545–623. Elsevier, North-Holland (2001)
10. Christensen, S.: Decidability and Decomposition in Process Algebras. PhD thesis, Department of Computer Science, The University of Edinburgh (1993)
11. Coquidé, J.-L., Dauchet, M., Gilleron, R., Vágvölgyi, S.: Bottom-up tree pushdown automata: Classification and connection with rewrite systems. Theor. Comput. Sci. 127(1), 69–98 (1994)
12. Dauchet, M., Tison, S.: The theory of ground rewrite systems is decidable. In: LICS, pp. 242–248. IEEE Computer Society, Los Alamitos (1990)
13. Göller, S., Lin, A.W.: The Complexity of Verifying Ground Tree Rewrite Systems. In: LICS, IEEE Computer Society, Los Alamitos (to appear, 2011)
14. Hack, M.H.T.: Decidability Questions for Petri Nets. PhD thesis, MIT (1976)
15. Löding, C.: Infinite Graphs Generated by Tree Rewriting. PhD thesis, RWTH Aachen (2003)
16. Lugiez, D., Schnoebelen, P.: The regular viewpoint on pa-processes. Theor. Comput. Sci. 274(1-2), 89–115 (2002)
17. Maidl, M.: The common fragment of CTL and LTL. In: FOCS, pp. 643–652 (2000)
18. Mayr, R.: Process rewrite systems. Inf. Comput. 156(1-2), 264–286 (2000)

19. Mayr, R.: Decidability of model checking with the temporal logic ef. Theor. Comput. Sci. 256(1-2), 31–62 (2001)
20. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
21. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theor. Comput. Sci. 37, 51–75 (1985)
22. Thomas, W.: Applied automata theory. Course Notes, RWTH Aachen (2005)
23. To, A.W.: Model Checking Infinite-State Systems: Generic and Specific Approaches. PhD thesis, LFCS, School of Informatics, University of Edinburgh (2010)
24. To, A.W., Libkin, L.: Algorithmic metatheorems for decidable LTL model checking over infinite systems. In: Ong, C.-H.L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 221–236. Springer, Heidelberg (2010)
25. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM 43(3), 555–600 (1996)

Author Index